

Francisco Durán (Ed.)

LNCSE 7571

Rewriting Logic and Its Applications

9th International Workshop, WRLA 2012

Held as a Satellite Event of ETAPS

Tallinn, Estonia, March 2012, Revised Selected Papers



Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

TU Dortmund University, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbruecken, Germany

Francisco Durán (Ed.)

Rewriting Logic and Its Applications

9th International Workshop, WRLA 2012
Held as a Satellite Event of ETAPS
Tallinn, Estonia, March 24-25, 2012
Revised Selected Papers

 Springer

Volume Editor

Francisco Durán
Universidad de Málaga
Departamento de Lenguajes y Ciencias de la Computación
E.T.S.I Informática
Campus de Teatinos
Boulevard Louis Pasteur, s/n
29071 Málaga, Spain
E-mail: duran@lcc.uma.es

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-34004-8 e-ISBN 978-3-642-34005-5
DOI 10.1007/978-3-642-34005-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012948196

CR Subject Classification (1998): F.3, D.2, D.3, D.1.3, F.1, F.4.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

© Springer-Verlag Berlin Heidelberg 2012

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, 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.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the proceedings of the 9th International Workshop on Rewriting Logic and Its Applications (WRLA 2012) that was held in Tallinn, Estonia, during March 24–25, 2012, as a satellite event of the European Joint Conferences on Theory and Practice of Software (ETAPS 2012).

Previous WRLA workshops were held in places all around the world, namely, at Asilomar (1996), Pont-à-Mousson (1998), Kanazawa (2000), Pisa (2002), Barcelona (2004), Vienna (2006), Budapest (2008), and Paphos (2010). Proceedings of these previous editions of the workshop have been published in Elsevier's *Electronic Notes in Theoretical Computer Science* series since its inception in 1996 until 2010, moving to Springer's *Lecture Notes in Computer Science* for that edition. In addition, extended versions of selected papers from WRLA 1996 were published in a special issue of *Theoretical Computer Science* (volume 285, 2002), extended versions of selected papers from WRLA 2004 appeared in a special issue of *Higher-Order and Symbolic Computation* (volume 21, 2008), and a special issue in the *Journal of Logic and Algebraic Programming* is currently under preparation for extended versions of selected papers of WRLA 2010.

The aim of the WRLA workshop series is to bring together researchers with a common interest in rewriting logic and its applications, and to give them the opportunity to present their recent works, to discuss future research directions, and to exchange ideas. Rewriting logic is a natural semantic framework for representing concurrency, parallelism, communication and interaction, as well as being an expressive (meta)logical framework for representing logics. It can then be used for specifying a wide range of systems and programming languages in various application fields.

WRLA 2012 had five invited speakers, namely, Saddek Bensalem, Santiago Escobar, Mark Hills, Grigore Rosu, and Martin Wirsing. The program was completed with regular papers and papers presenting work in progress, each of which was reviewed by four reviewers. Twelve papers were submitted, out of which eight were selected as regular papers. These regular papers are the ones included in these proceedings, together with papers from four of the five invited speakers.

I thank the authors who submitted their work to WRLA 2012 and who, through their contributions, made this workshop a high-quality event. I would also like to thank the Program Committee members and the external reviewers for their timely and insightful reviews as well as for their involvement in the post-reviewing discussions. I am also grateful to those who provided me with all kinds of help and useful advice, and to the invited speakers and the WRLA Steering Committee. I also thank A. Voronkov for the excellent EasyChair conference system, and the organizers of the ETAPS Conference in Tallinn, who provided an excellent environment for the development of the event.

Organization

Program Committee

Emilie Balland	UHP-LORIA, France
Artur Boronat	University of Leicester, UK
Roberto Bruni	Università di Pisa, Italy
Manuel Clavel	Universidad Complutense de Madrid, Spain
Grit Denker	SRI International, USA
Francisco Durán	University of Málaga, Spain
Steven Eker	SRI International, USA
Santiago Escobar	Technical University of Valencia, Spain
Kokichi Futatsugi	JAIST, Japan
Alexander Knapp	Universität Augsburg, Germany
Dorel Lucanu	Alexandru Ioan Cuza University, Romania
Salvador Lucas	Universidad Politécnica de Valencia, Spain
Narciso Martí-Oliet	Universidad Complutense de Madrid, Spain
José Meseguer	University of Illinois at Urbana-Champaign, USA
Ugo Montanari	Università di Pisa, Italy
Pierre-Etienne Moreau	INRIA-LORIA Nancy, France
Kazuhiro Ogata	JAIST, Japan
Miguel Palomino	Universidad Complutense de Madrid, Spain
Grigore Rosu	University of Illinois at Urbana-Champaign, USA
Vlad Rusu	Inria, France
Carolyn Talcott	SRI International, USA
Mark Van Den Brand	Eindhoven University of Technology, The Netherlands
Martin Wirsing	Ludwig-Maximilians-Universität, Germany
Peter Csaba Ølveczky	University of Oslo, Norway

Additional Reviewers

Abraham, Erika	Kuiper, Ruurd
Arusoae, Andrei	Lluch Lafuente, Alberto
Bach, Jean-Christophe	Meredith, Patrick
Brauner, Paul	Prugniel, Francois
Ellison, Chucky	Vandin, Andrea
Fantechi, Alessandro	Wijs, Anton

Table of Contents

Rigorous Component-Based System Design (Invited Paper)	1
<i>Ananda Basu, Saddek Bensalem, Marius Bozga, and Joseph Sifakis</i>	
Program Analysis Scenarios in Rascal	10
<i>Mark Hills, Paul Klint, and Jurgen J. Vinju</i>	
\mathbb{K} Framework Distilled	31
<i>Dorel Lucanu, Traian Florin Şerbănuţă, and Grigore Roşu</i>	
Design and Analysis of Cloud-Based Architectures with KLAIM and Maude	54
<i>Martin Wirsing, Jonas Eckhardt, Tobias Mühlbauer, and José Meseguer</i>	
Making Maude Definitions More Interactive	83
<i>Andrei Arusoaie, Traian Florin Şerbănuţă, Chucky Ellison, and Grigore Roşu</i>	
Model Checking LTLR Formulas under Localized Fairness	99
<i>Kyungmin Bae and José Meseguer</i>	
Modelling and Analyzing Adaptive Self-assembly Strategies with Maude	118
<i>Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin</i>	
Formal Modeling and Analysis of Human Body Exposure to Extreme Heat in HI-Maude	139
<i>Muhammad Fadlisyah, Peter Csaba Ölveczky, and Erika Ábrahám</i>	
Order-Sorted Equality Enrichments Modulo Axioms	162
<i>Raúl Gutiérrez, José Meseguer, and Camilo Rocha</i>	
Timed CTL Model Checking in Real-Time Maude	182
<i>Daniela Lepri, Erika Ábrahám, and Peter Csaba Ölveczky</i>	
Using Narrowing to Test Maude Specifications	201
<i>Adrián Riesco</i>	

A Rule-Based Framework for Building Superposition-Based Decision Procedures	221
<i>Elena Tushkanova, Alain Giorgetti, Christophe Ringeissen, and Olga Kouchnarenko</i>	
Author Index	241

Rigorous Component-Based System Design

(Invited Paper)

Ananda Basu, Saddek Bensalem, Marius Bozga, and Joseph Sifakis

VERIMAG Laboratory, Université Joseph Fourier Grenoble, CNRS

Abstract. Rigorous system design requires the use of a single powerful component framework allowing the representation of the designed system at different levels of detail, from application software to its implementation. This is essential for ensuring the overall coherency and correctness. The paper introduces a rigorous design flow based on the BIP (Behavior, Interaction, Priority) component framework [1]. This design flow relies on several, tool-supported, source-to-source transformations allowing to progressively and correctly transform high level application software towards efficient implementations for specific platforms.

1 Introduction

Traditional engineering disciplines such as civil or mechanical engineering are based on solid theory for building artefacts with predictable behaviour over their life-time. These follow laws established by simple Newtonian physics and recognized building codes and regulations. Their complexity is limited by these physical and normative factors.

In contrast, for systems engineering, we do not have an equivalent theoretical basis allowing to infer system properties from the properties of its components. Computer science provides only partial answers to particular system design problems. With few exceptions, in this domain predictability is impossible to guarantee at design time and therefore, a posteriori validation remains the only means for ensuring their correct operation over time.

The complexity of systems currently being built, the fast pace of technological innovation, and the harsh market conditions to which they are subjected, including in particular time-to-market, create many difficulties for system design. These difficulties can be traced in large part to our inability to predict the behaviour of an application's software running on a given platform. Usually, such systems are built by reusing and assembling components: simpler sub-systems. This is the only way to master the complexity and to ensure the correctness of the overall design, while maintaining or increasing productivity. However, system-level integration becomes extremely hard because components are usually highly heterogeneous: and have different characteristics, are often developed using different technologies, and highlight different features from different viewpoints. Other difficulties stem from current design approaches, often empirical and based on expertise and experience of design teams. Naturally, designers

attempt to solve new problems by reusing, extending and improving past solutions proven to be efficient and robust. This favors component reuse and avoids re-inventing and re-discovering design solutions every time. Nevertheless, on a longer term perspective, it may also be counter-productive: people are not always able to adapt in a satisfactory manner to new requirements and moreover, they tend to reject better solutions simply because they do not fit their design know-how. In this paper, we present the system design, the rigorous design flow and the BIP component framework.

2 System Design

System design is facing several difficulties, mainly due to our inability to predict the behavior of an application software running on a given platform.

System design is the process leading to a mixed software/hardware system meeting given specifications. It involves the development of application software taking into account features of an execution platform. The latter is defined by its architecture involving a set of processors equipped with hardware-dependent software such as operating systems as well as primitives for coordination of the computation and interaction with the external environment.

System design radically differs from pure software design in that it should take into account not only functional but also extra-functional specifications regarding the use of resources of the execution platform such as time, memory and energy. Meeting extra-functional specifications is essential for the design of embedded systems. It requires evaluation of the impact of design choices on the overall behavior of the system. It also implies a deep understanding of the interaction between application software and the underlying execution platform. We currently lack approaches for modelling mixed hardware/software systems. There are no rigorous techniques for deriving global models of a given system from models of its application software and its execution platform.

A system design flow consists of steps starting from specifications and leading to an implementation on a given execution platform. It involves the use of methods and tools for progressively deriving the implementation by making adequate design choices.

We consider that a system design flow must meet the following essential requirements:

- *Correctness*: This means that the designed system meets its specifications. Ensuring correctness requires that the design flow relies on models with well-defined semantics. The models should consistently encompass system description at different levels of abstraction from application software to its implementation. Correctness can be achieved by application of verification techniques. It is desirable that if some specifications are met at some step of the design flow, they are preserved in all the subsequent steps.
- *Productivity*: This can be achieved by system design flows.
 - providing high level domain-specific languages for ease of expression.

- allowing reuse of components and the development of component-based solutions.
- integrating tools for programming, validation and code generation.
- *Performance*: The design flow must allow the satisfaction of extra-functional properties regarding optimal resource management. This means that resources such as memory, time and energy are first class concepts encompassed by formal models. Moreover, it should be possible to analyze and evaluate efficiency in using resources as early as possible along the design flow. Unfortunately, most of the widely used modeling formalisms offer only syntactic sugar for expressing timing constraints and scheduling policies. Lack of adequate semantic models does not allow consistency checking for timing requirements, or meaningful composition of features.
- *Parcimony*: The design flow should not enforce any particular programming or execution model. Very often system designers privilege specific programming models or implementation principles that a priori exclude efficient solutions. They program in low level languages that do not help discover parallelism or non determinism and enforce strictly sequential execution. For instance, programming multimedia applications in plain C may lead to designs obscuring the inherent functional parallelism and involving built-in scheduling mechanisms that are not optimal. It is essential that designers use adequate programming models. Furthermore, design choices should be driven only by system specifications to obtain the best possible implementation.

3 Rigorous Design Flow

We call *rigorous* a design flow which allows guaranteeing essential properties of the specifications. Most of the rigorous design flows privilege a unique programming model together with an associated compilation chain adapted for a given execution model. For example, synchronous system design relies on synchronous programming models and usually targets hardware or sequential implementations on single processors [2]. Alternatively, real-time programming based on scheduling theory for periodic tasks, targets dedicated real-time multitasking platforms [3].

A rigorous design flow should be characterized by the following:

- It should be *model-based*, that is all the software and system descriptions used along the design flow should be based on a single semantic model. This is essential for maintaining the overall coherency of the flow by guaranteeing that a description at step n meets essential properties of a description at step $n - 1$. This means in particular that the semantic model is expressive enough to directly encompass various types of component heterogeneity arising along the design flow [4]:
- Heterogeneity of computation: The semantic model should encompass both synchronous and asynchronous computation by using adequate coordination mechanisms. This should allow in particular, modeling mixed hardware/software systems.

- Heterogeneity of interaction: The semantic model should enable natural and direct description of various mechanisms used to coordinate execution of components including semaphores, rendezvous, broadcast, method call, etc.
 - Heterogeneity of abstraction: The semantic model should support the description of a system at different abstraction levels from its application software to its implementation. This makes possible the definition of a clear correspondence between the description of an untimed platform-independent behavior and the corresponding timed and platform-dependent implementation.
- It should be *component-based*, that is it provides primitives for building composite components as the composition of simpler components. Existing theoretical frameworks for composition are based on a single operator e.g., product of automata, function call. Poor expressiveness of these frameworks may lead to complicated designs: achieving a given coordination between components often requires additional components to manage their interaction.
- For instance, if the composition is by strong synchronization (rendezvous) modelling broadcast requires an extra component to choose amongst the possible strong synchronizations a maximal one. We need frameworks providing families of composition operators for natural and direct description of coordination mechanisms such as protocols, schedulers and buses.
- It should rely on tractable theory for guaranteeing *correctness by construction* to avoid as much as possible monolithic a posteriori verification. Such a theory is based on two types of rules:
 - Compositionality rules for inferring global properties of composite components from the properties of composed components e.g. if a set of components are deadlock-free then for a certain type of composition the obtained composite components is deadlock-free too. A special and very useful case of compositionality is when a behavioral equivalence relation between components is a congruence [5]. In that case, substituting a component in a system model by a behaviorally equivalent component leads to an equivalent model.
 - Composability rules ensuring that essential properties of a component are preserved when it is used to build composite components.

4 The BIP Design Flow

BIP [1] (Behavior, Interaction, Priority) is a general framework encompassing rigorous design. It uses the BIP language and an associated toolset supporting the design flow. The BIP language is a notation which allows building complex systems by coordinating the behaviour of a set of atomic components. Behavior is described as a Petri net extended with data and functions described in C. The transitions of the Petri are labelled with guards (conditions on the state of a component and its environment) as well as functions that describe computations on local data. The description of coordination between components is layered. The first layer describes the interactions between components. The second layer

describes dynamic priorities between the interactions and is used to express scheduling policies. The combination of interactions and priorities characterizes the overall architecture of a component. It confers BIP strong expressiveness that cannot be matched by other languages [6]. BIP has clean operational semantics that describe the behaviour of a composite component as the composition of the behaviors of its atomic components. This allows a direct relation between the underlying semantic model (transition systems) and its implementation.

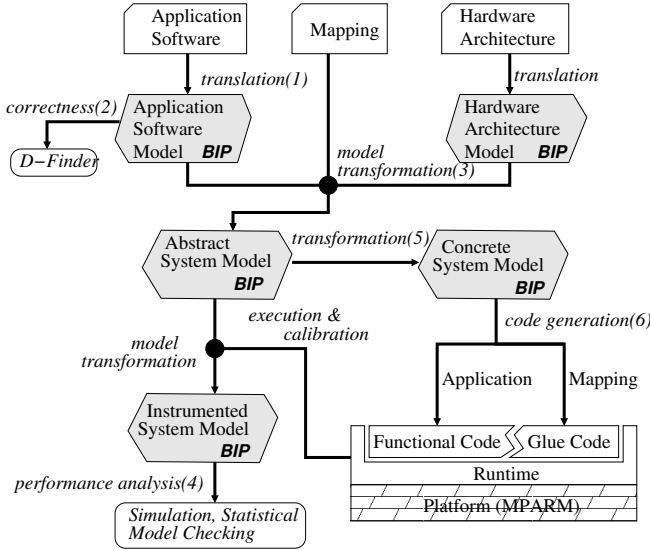


Fig. 1. BIP Design Flow for Manycore

The BIP design flow uses a single language to ensure consistency between the different design steps. This is mainly achieved by applying source-to-source transformations between refined system models. These transformations are proven correct-by-construction, that means, they preserve observational equivalence and consequently essential safety properties. The design flow involves several distinct steps, as illustrated in figure 1:

1. The translation of the application software into a BIP model. This allows its representation in a rigorous semantic framework. Translations for several programming models (including synchronous, data-flow and event-driven) into BIP are already implemented.
2. Correctness checking of the functional aspects of the application software. Functional verification needs to be done only at high level models since safety properties and deadlock-freedom are preserved by different transformations applied along the design flow. To avoid inherent complexity limitations, the verification method rely on compositionality and incremental techniques.

3. The generation of an abstract system model from the BIP model representing the application software, a model of the target execution platform as well as a mapping of the atomic components of the application software model into processing elements of the platform. The obtained model takes into account hardware architecture constraints and execution times of atomic actions. Architecture constraints include mutual exclusion induced from sharing physical resources such as buses, memories and processors as well as scheduling policies seeking optimal use of these resources.
4. Performance analysis on the system model using simulation-based models combined with statistical model checking.
5. The generation of a concrete system model obtained from the abstract model by expressing high level coordination mechanisms e.g., interactions and priorities by using primitives of the execution platform. This transformation involves the replacement of atomic multiparty interactions and/or dynamic priorities by protocols using asynchronous message passing (send/receive primitives) and arbiters ensuring. These transformations are proved correct-by-construction as well.
6. The generation of platform dependent code, including both functional and glue code needed to deploy and run the application on the target multi-core. In particular, components mapped on the same core can be statically composed thus avoiding extra overhead for (local) coordination at runtime.

5 The BIP Framework

The BIP framework allows building complex systems by coordinating the behavior of a set of atomic components. Coordination in BIP uses connectors, to specify possible interaction patterns between components, and priorities, to select amongst possible interactions.

Atomic components are finite-state automata or Petri nets that are extended with arbitrary data and ports. Ports are action names, and may be associated with data. They are used for interaction with other components. States denote control locations at which the components await for interaction.

A transition is an execution step, labeled by a port, from a control location to another. It has associated a guard and an action, that are respectively a boolean condition and a function defined on local data. In BIP complex data and their transformations are written in C/C++. A transition can be executed if its guard evaluates to true and some interaction involving its port is enabled. The execution is an atomic sequence of two microsteps: (i) execution of the interaction involving the port, which is a synchronization between several components, with possible exchange of data, followed by (ii) execution of the action associated with the transition.

In the design flow, BIP is used as a unifying semantic model to ensure consistency between the different design steps. The design flow involves four distinct steps. They consist in translating the application software into a BIP model and deriving progressively an implementation by application of source-to-source transformations. These transformations are correct-by-construction as the obtained

BIP models are observationally equivalent. In particular, they preserve safety properties of the application software. Furthermore, the D- Finder verification tool is used to check essential safety properties of the application software.

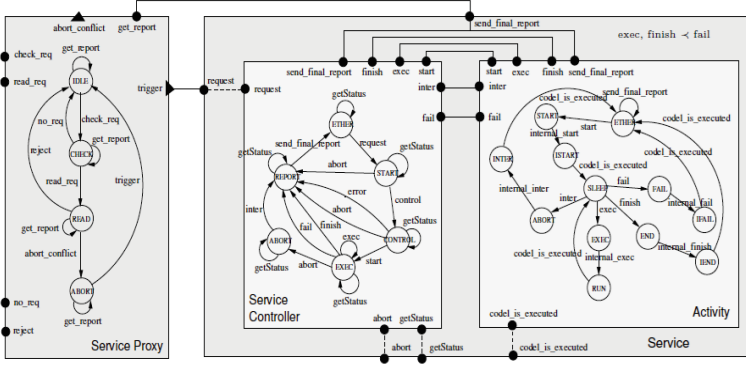


Fig. 2. BIP Components: a service of the DALA Robot

Example 1. Figure 2 shows two atomic components, Service-Controller and Activity of a larger application named DALA robot controller. Activity wraps the long-time computation of some specific applicative function. Service-Controller provides execution control (i.e., triggering, canceling, error control, etc) over the associated Activity component. For sake of simplicity, the figure presents only the skeleton control behavior (i.e., ports and transitions) whereas the data and associated code is omitted. For example, Activity is initialized (start transition) and then it executes its associated functions (exec, internal exec transitions). The execution may finish normally (finish transition), may fail (fail transition) or may be interrupted (inter transition).

Composite components are defined by assembling constituent components (atomic or composite) using connectors. Connectors relate ports of interacting components. They represent sets of interactions, that is, non-empty sets of ports that have to be jointly executed. Within a connector, an interaction can take place in two situations: either all involved ports are ready to participate (strong synchronization), or some subset of ports triggers the interaction without waiting for other ports (broadcast). The set of valid interactions within connectors are formally defined using algebraic expressions on ports using a binary fusion operator and a unary typing operator. Typing associates with connector-ends (ports or connectors) synchronization types: trigger (active port, initiates broadcast) or synchron (passive port). Moreover, with every interaction of a connector there is associated a guard and a data transfer function. An interaction may be executed only when its guard is true. Its execution consists in computing the data transfer function and notifying the components involved in the interaction.

Finally, priorities are used to arbitrate between simultaneously enabled interactions within a BIP component. These are rules, each consisting of an ordered pair of interactions associated with a condition. When the condition holds and both interactions of the corresponding pair are enabled, only the one with higher-priority can be executed.

Example 2. Figure 2 also presents the Service composite component obtained by the composition of Activity and Service-Controller through six connectors. They enforce strong synchronizations of several actions and allow the Service-Controller to initiate and follow the computation performed within the Activity. Priorities are used to privilege the execution of fail interaction that is error handling, over finish and exec interactions, which correspond to normal behavior. The example also illustrates the principle of encapsulation used in BIP: the Service component is further composed with the Service-Proxy component by using the ports available on its interface, which are explicitly re-directed either to ports of subcomponents or to inner connectors. The trigger request connector between the ServiceProxy and the Service illustrates a broadcast initiated by the trigger port, that is, trigger actions are either executed alone, or synchronized with request actions, whenever enabled.

The design flow is entirely supported by the BIP language and an extensible toolset, see Figure 3. This includes translators from various programming models, verification tools, source- to-source transformers as well as a compiler for generating code executable by a dedicated engine.

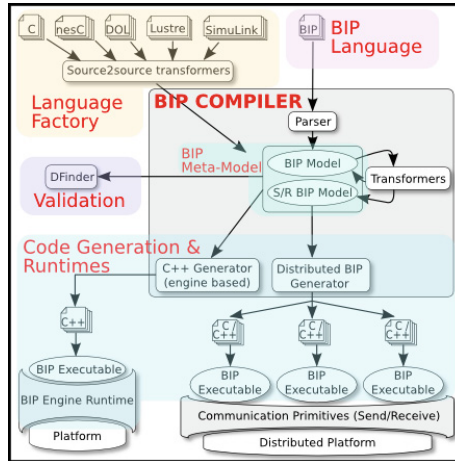


Fig. 3. The BIP Toolset

6 Conclusion

In this work the key motivations are facilitation of the design of advanced heterogeneous embedded systems with distributed SW. At the same time it can provide a secure design flow with performance analysis and validation for multiple programming models to target hardware architectures including multiple heterogeneous smart subsystems and components. The contribution of the paper is twofold.

- First, we developed the BIP (Behavior, Interaction, Priority) component framework that encompasses an expressive notion of composition for heterogeneous components by combining interactions and priorities. This allows description at different levels of abstraction from application software to mixed hardware/software systems.
- Second, we developed a rigorous design flow that uses BIP as a unifying semantic model to derive from application software, a model of the target architecture and a mapping, a correct implementation. Correctness of implementation is ensured by application of source-to-source transformations in BIP, which preserve correctness of essential design properties. The design is fully automated and supported by a toolset including a compiler, code generators, the D-Finder verification tool and model transformers.

References

1. Basu, A., Bozga, M., Sifakis, J.: Modeling Heterogeneous Real-time Systems in BIP. In: Software Engineering and Formal Methods SEFM 2006 Proceedings, pp. 3–12. IEEE Computer Society Press (2006)
2. Halbwachs, N.: Synchronous Programming of Reactive Systems. Kluwer Academic Publishers (1993)
3. Burns, A., Welling, A.: Real-Time Systems and Programming Languages, 3rd edn. Addison-Wesley (2001)
4. Henzinger, T.A., Sifakis, J.: The Embedded Systems Design Challenge. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 1–15. Springer, Heidelberg (2006)
5. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
6. Bliudze, S., Sifakis, J.: A Notion of Glue Expressiveness for Component-Based Systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 508–522. Springer, Heidelberg (2008)

Program Analysis Scenarios in Rascal

Mark Hills¹, Paul Klint^{1,2}, and Jurgen J. Vinju^{1,2}

¹ [Centrum Wiskunde & Informatica](#), Amsterdam, The Netherlands

² [INRIA Lille Nord Europe](#), France

Abstract. Rascal is a meta programming language focused on the implementation of domain-specific languages and on the rapid construction of tools for software analysis and software transformation. In this paper we focus on the use of Rascal for software analysis. We illustrate a range of scenarios for building new software analysis tools through a number of examples, including one showing integration with an existing Maude-based analysis. We then focus on ongoing work on alias analysis and type inference for PHP, showing how Rascal is being used, and sketching a hypothetical solution in Maude. We conclude with a high-level discussion on the commonalities and differences between Rascal and Maude when applied to program analysis.

1 Introduction

Rascal [32,33] is a meta programming language focused on the implementation of domain-specific languages and on the rapid construction of tools for software analysis and software transformation. Rascal is the successor to both ASF [4] and ASF+SDF [48,46], providing features for defining grammars, parsing programs, analyzing program code, generating new programs, interacting with external tools (through Java), and visualizing the results of these operations.

In this paper we focus on software analysis, exploring the design space of Rascal analysis solutions. We begin this in Section 2 by providing a brief introduction to Rascal, focusing on the design of the language and how this design is realized by Rascal's language features. We also show several small examples of Rascal code. We continue this in Section 3, presenting several analyses developed using Rascal: a hybrid Rascal/Maude [11] analysis for finding type and units of measurement errors; a Rascal analysis of Java code that uses information extracted from the Eclipse Java Development Tools; and the Rascal name resolver and type checker, which works as part of the Rascal development environment and is implemented completely in Rascal. These examples highlight different solution scenarios in the design space, including (at one extreme) using Rascal as a coordination language for existing analysis tools and (at the other) building an analysis solely in Rascal.

In Section 4 we then describe ongoing work, written mostly in Rascal but with some integration of external tools, on defining a set of analyses for the PHP language. We focus here on two specific analyses: alias analysis and type inference. We start this description by setting out a number of tasks that need to be performed for these analyses – for instance, parsing the source program, or defining an internal representation for storing analysis facts. We then show how Rascal is used to provide support for each

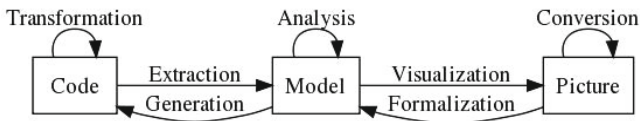


Fig. 1. Meta-programming domain: 3 layers of software representation with transitions

of these tasks. Another possibility for defining these analyses would be to use rewriting logic [34], specifically techniques developed as part of the rewriting logic semantics [35] project. Therefore, to end the section, we also show how these analysis tasks could be supported with a rewriting logic semantics-based analysis in Maude.

This paper touches on several areas with extensive related work. Section 5 focuses on existing research related directly to Rascal, the analysis of PHP programs, and program analysis using Maude. Section 6 then offers observations and discussion, highlighting what we believe to be the advantages and disadvantages of Rascal and Maude for developing program analysis tools.

2 Rascal

Rascal was designed to cover the entire domain of meta-programming, shown pictorially in Figure 1. The language itself is designed with unofficial “language layers”. This allows Rascal developers to start with just the core language features, adding more advanced features as they become more comfortable with the language. This language core contains basic data-types (booleans, integers, reals, source locations, date-time, lists, sets, tuples, maps, relations), structured control flow (if, while, switch, for), and exception handling (try, catch). The syntax of these constructs is designed to be familiar to programmers: for instance, if statements and try/catch blocks look like those found in C and Java, respectively. All data in Rascal is immutable (i.e., no references are ever created or taken), and all code is statically typed. At this level, Rascal looks like a standard general purpose programming language with immutable data structures.

Rascal’s type system is organized as a lattice, with bottom (`void`) and top (`value`) elements. The Rascal `node` type is the parent of all user-defined datatypes, including the types of concrete syntax elements (`Stmt`, `Expr`, etc). Numeric types also have a parent type, `num`, but are not themselves in a subtype relation: i.e., `real` is not a parent of `int`. The basic types available in Rascal, including examples, are shown in Table 1.

Beyond the type system and the language core, Rascal also includes a number of more advanced features. These features can be progressively added to create more complex programs, and are needed in Rascal to enable the full range of meta-programming capabilities. These more advanced features include:

- Algebraic data type definitions, with optional type parameters, allow the user to define new data types for use in the analysis. These data types are similar to sum types in functional languages like ML or (possibly parameterized) sort and operator definitions in algebraic systems like Maude.

Table 1. Basic Rascal Types

Type	Example literal
bool	true, false
int	1, 0, -1, 123456789
real	1.0, 1.0232e20, -25.5
rat	1r4, 22r7, -3r8
str	"abc", "first\nnext"
loc	file:///etc/passwd
datetime	\$2012-05-08T22:09:04.120+0200
tuple[t_1, \dots, t_n]	$\langle 1, 2 \rangle, \langle \text{"john"}, 43, \text{true} \rangle$
list[t]	$[], [1], [1,2,3], [\text{true}, 2, \text{"abc"}]$
set[t]	$\{\}, \{1, 2, 3, 5, 7\}, \{\text{"john"}, 4.0\}$
rel[t_1, \dots, t_n]	$\{\langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 1, 3 \rangle\}, \{\langle 1, 10, 100 \rangle, \langle 2, 20, 200 \rangle\}$
map[t, u]	$()$, $(1 : \text{true}, 2 : \text{true})$, $(6 : \{1, 2, 3, 6\}, 7 : \{1, 7\})$
node	f, add(x, y), g("abc"), [2, 3, 4]

- A built-in grammar formalism allows the definition of context-free grammars. These grammars are used to generate a scannerless generalized parser, which allows for modular syntax definitions (i.e., unions of defined grammars) and the parsing of programs in real programming languages. The syntax formalism is EBNF-like and includes disambiguation facilities, such as the ability to indicate associativity and precedence, add follow restrictions, and even provide arbitrary code to disallow specific parses.
- Pattern matching is provided over all Rascal data types: matches can be performed against numbers, strings, nodes, etc. A number of advanced pattern matching operators, such as deep match ($/$, matching values nested at an arbitrary depth inside other values), negative match ($!$), set matching, and list matching are also provided. Given the importance of concrete syntax for some meta-programming tasks, it is also possible to match against concrete syntax fragments, e.g., matching a while loop and binding variables to syntax fragments representing the loop condition and the loop body.
- Additionally, pattern matching is used in the formal parameters of functions, allowing function dispatch to be based on the pattern matching mechanism. This provides for more extensible code, since new constructors of a user-defined datatype can be handled by using new variants of an existing function, instead of requiring a single function with a large switch/case statement. As an equivalent to the switch/case default case, a default function provides the default behavior for the function when none of the other cases match.
- In cases where there are multiple matches for a pattern, backtracking happens from right to left in a pattern, enforcing lexical scope (names bind starting at the left, and can be used in the pattern to the right of the binding site) and providing a natural order on matches. A successful match can be explicitly discarded by the user with the `fail` keyword.

- List, set, and map comprehensions, in combination with pattern matching and other Rascal expressions, allow new lists, sets, and maps to be constructed based on complex conditions. For instance, one could use a deep match to find all while loops in a set of program files that contain a condition with a less than comparison. Also provided is the `<-` element generation operator, which can enumerate the elements of all container data-types, e.g. lists, sets, maps, and trees, and can be used inside comprehensions and in for loops.
- String templates with margins and an auto-indent feature provide a straightforward way to generate formatted code in multi-line source code templates.
- `visit` statements, with a syntax similar to that of `switch` statements, perform *structure-shy* traversals of Rascal data types, allowing one to match only those cases of interest. Visit cases can execute arbitrary code, for instance to keep track of statistics or analysis information, or can directly replace the matched node with one of the same type. Visits are parameterized by a traversal strategy (e.g., top-down) to allow different traversal orders.
- `solve` statements allow fixed-point computations to be expressed directly as a language construct. The statement continues to iterate as long as the result of the condition expression continues to change.

A number of Rascal features focus on the safety and modularity of Rascal code. While local variable types can be inferred, parameter and return types in functions must be provided. This allows better error messages to be generated, since errors detected by the inferencer can be localized within a function, and also provides documentation (through type annotations) on function signatures. Also, the only casting mechanism is pattern matching, which prevents the problems with casts found in C (lack of safety) and Java (runtime casting exceptions). Finally, the use of persistent data structures eliminates a number of standard problems with using references which can leak out of the current scope or be captured by other variables.

Example: As a simple example, imagine that we want to work with the Peano representation of natural numbers. In Maude, these could be defined as follows:

```
fmod PEANO is
  sort Nat .
  op z : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .

  vars N M : Nat .

  op plus : Nat Nat -> Nat .
  eq plus(s(N),M) = s(plus(N,M)) .
  eq plus(z,M) = M .
endfm
```

In Rascal, this same functionality would be defined as follows:

```

module Nat

data Nat = z() | s(Nat);

Nat plus(s(Nat n), Nat m) = s(plus(n,m));
Nat plus(z(), Nat m) = m;

```

Function `plus` could also be defined using a switch/case statement, as follows:

```

Nat plussc(Nat n, Nat m) {
  switch(n) {
    case s(j) : return s(plussc(j,m)) ;
    case z()  : return m;
  }
}

```

As a more complex example, take the case where we have colored binary trees: trees with an integer in the leaves, but with a color (given as a string) defined at each composite node. This would be defined as follows:

```

data ColoredTree
  = leaf(int n)
  | composite(str color, ColoredTree left, ColoredTree right);

```

Suppose we want to analyze a `ColoredTree`, computing how often each color appears at each node. The Rascal code is shown in Listing 1. In this code, we use a map, held in a local variable `counts` with inferred type `map[str, int]`, to maintain the counts. A visit statement is used to traverse the binary tree, matching only the composite nodes, and binding the color stored in the node to the string variable `color`. The statement `counts[color]?0 += 1` then increments the current frequency count for the given color if it exists, or it initializes this count to 0 first and then increments, assigning the result back into the map entry for the color.

Listing 1. Counting frequencies of colors in a `ColoredTree`.

```

public map[str, int] colorDistribution(ColoredTree t) {
  counts = ();
  visit(t) {
    case composite(str color, -, -): counts[color] ? 0 += 1;
  }
  return counts;
}

```

3 Scenarios for Program Analysis in Rascal

Scenarios for program analysis using Rascal can be viewed as a spectrum: at one end, Rascal acts just as a coordination language, with all the analysis work done using external tools; at the other, all work needed for the analysis, from parsing, through all

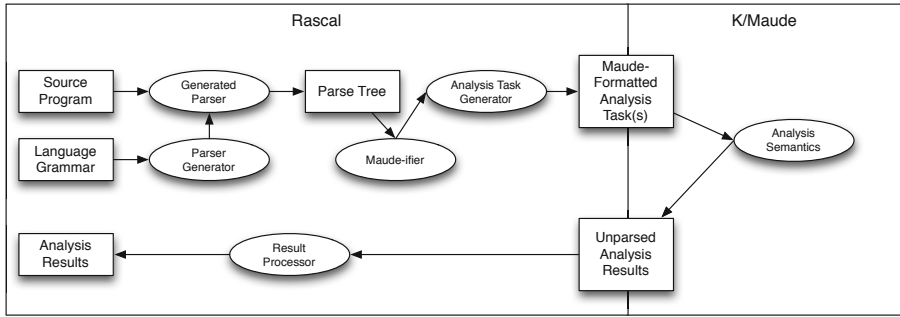


Fig. 2. Integrating Rascal and K

the analysis tasks, to the display of the results, is done within Rascal. Many solutions are somewhere in between, with Rascal providing significant functionality while also interacting with existing tools. This section presents three examples exemplifying these alternatives.

3.1 Integrating Rascal with RLS-Based Analysis Tools

One approach to program analysis in Rascal, near the “coordination language” end of the spectrum, is to use a program analysis tool based on rewriting logic semantics (RLS) and Maude while leveraging the support for parsing and IDE integration provided by Rascal. This is supported in Rascal through the RLSRunner library [22].

RLSRunner provides components in both Rascal and Maude for linking Rascal language definitions with RLS analysis semantics. In Rascal, functions and data types are provided both to perform the analysis in Maude using the analysis semantics and to process the results to yield information about the error and warning messages to display in Eclipse. In Maude, sorts and operations are defined to model and use Rascal source locations, allowing the locations of errors to be tracked by the analysis and reported accurately to the user in Eclipse. These locations are added to an analysis semantics by extending abstract syntax sorts with new operations to represent located versions of terms, with additional equations provided to keep track of the locations in the configuration and to give back the original (unlocated) terms.

Figure 2 provides an overview of the RLSRunner process. The two initial inputs are a grammar for the language being analyzed, created using the Rascal grammar formalism, and a source program. The grammar is processed using the Rascal parser generator, generating a parser for the language under analysis. This parser is then used to parse the source program, as well as to provide features used by the Eclipse-based program user interface such as code folding, code outlining, etc. Using the parse tree emitted by the parser, a Rascal program dubbed the “Maude-ifier” is then run. In conjunction with the analysis task generator, this generates individual Maude terms from the parsed program, with one term per analysis task – in some cases a task represents the entire program, while in others tasks may be generated for smaller units, such as for individual functions. Each analysis task is then evaluated in the analysis semantics, yielding the analysis results,

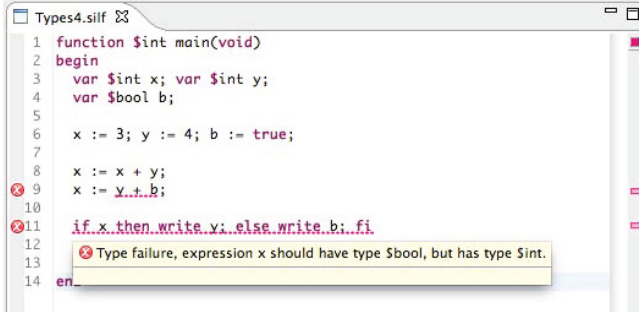


Fig. 3. Type Errors in SILF Programs shown in Rascal’s Eclipse-based IDE

which are processed by the result processor to yield analysis information shown to the user in the IDE (e.g., error messages).

An example of the result of this process is shown in Figure 3, which shows type errors identified in a SILF [20] program. The type errors are identified using the types policy in the SILF Policy Framework [24], a rewriting logic semantics-based framework for defining program analyses. The SILF units policy generates similar messages, but based instead on annotations provided for units of measurement.

3.2 Refactoring Analysis with the Eclipse JDT

As an experiment in measuring the maintainability of large software systems, we decided to investigate the difference in maintenance complexity for an interpreter written using either the Visitor [17, page 331] or the Interpreter [17, page 243] design patterns. We did this by creating a refactoring [38,37,16], dubbed V2I [23], which transforms an interpreter written using the Visitor pattern into one written using the Interpreter pattern, holding all else constant. We then measured the difficulty of performing various maintenance scenarios on the two systems [21].

As part of the work in developing V2I, we had to develop an analysis that would identify the code to be refactored. Normally, this would require creating a grammar for Java, parsing the Java code used in the interpreter, and performing analyses to bind name and type declaration information to entities in the code (while also taking account of information provided by external libraries, which could be in binary form), all before developing the analysis needed to perform the refactoring. Instead of doing this in Rascal, we opted to instead reuse the information computed by the Eclipse Java Development Tools (JDT), which computes all of these facts as part of its IDE support for Java development. These facts are extracted from the JDT using the Rascal JDT library, which communicates with Eclipse to build Rascal representations of a number of Java program facts. Some of the facts used by the V2I analysis are shown in Table 2, with both entities (types, classes, fields, methods) and relationships between entities (the remaining four items) shown.

Using these entities, the analysis performs a number of computations to find all methods that must be refactored. This is done by computing a number of intermediary relations – for instance, from visitor interfaces to implementing classes – with the

Table 2. Rascal JDT Interface: Extracted Entities and Relationships

Extracted Fact	Description
types	classes, interfaces, enums
classes	classes
fields	fields
methods	methods
modifiers	modifiers on definitions (e.g., public, final)
implements	interface \times implementer
extends	class or interface \times extender
declaredMethods	class or interface \times method declaration

final relation containing all identified methods, the method locations, and the method source code. Additional relations indicate any dependencies of the refactored methods that must also be modified. For example, in cases where method code is relocated to a different class, uses of private fields in this code are changed first to uses of public getter and setter methods. As in the prior example, Rascal acts partly as a coordination language, but here also performs all the V2I-specific analysis using Rascal code.

3.3 Type-Checking Rascal in Rascal

The Rascal type checker (referred to hereafter as the RTC for conciseness) enforces the static typing rules of the Rascal language. It is fully implemented in Rascal and uses no external tools.

The parsing of Rascal code precedes RTC, naturally. Rascal includes a syntax definition formalism which is bootstrapped, and supports embedded concrete syntax fragments. When parsing Rascal modules with concrete syntax, such as in the implementation of RTC, an automatically generated parser for Rascal embedded in Rascal is used. The input for RTC are parse trees of any Rascal modules, which are also processed by similarly generated parsers.

Several key principles are used in the RTC:

- Checking occurs at the level of individual modules. Imported modules are assumed to be correct, and supply a module signature with type information for all declarations.
- Checking with a module is performed by checking each function individually. Function signatures must be given explicitly for each declared function, and are not inferred.
- Function bodies are checked by statically evaluating the code in the function body using a recursive interpreter which implements the type semantics. Values in the interpreter represent types and abstract entities, such as variables, functions, etc.
- Local inference is handled by checking for stabilization across multiple (static) evaluations of iterating constructs. For instance, loops that assign new values to variables are evaluated twice. Types that fail to stabilize are assumed to be `value`, the top of the type lattice.

The result of running the checker is an assignment of types (including error types) to all names and expressions in a Rascal module. This information is then used by the Rascal IDE to provide type documentation (visible by hovering over a name or expression in the IDE), type error annotations, and documentation links allowing the user to jump to the definition of a name, including definitions of variables, constructors, functions, and user-defined data types.

3.4 Discussion

These three tools exemplify the diversity of solution strategies when using Rascal. In some cases we use external tools, while in others we do not. The RTC is self-contained and bootstrapped, the refactoring emphasizes reuse of the Eclipse JDT and the SILF checker reuses a version of K in Maude. These examples also use different intermediate data-structures and different types of analysis algorithms.

The design of Rascal is intended to leave many choices to the meta programmer. It provides a kaleidoscopic set of solution scenarios. Users are not forced to use the language for all components, and are actively helped by the system to connect to other systems. This is one reason Rascal has been designed as a programming language rather than as a specification formalism [45].

4 Analyzing PHP

In this section we describe analyses of PHP code using Rascal. We also describe the same analyses as they could be implemented in Maude. Our goal is to give the reader insight into how these two systems compare in terms of functionality and style.

PHP¹ is a dynamically-typed server-side scripting language. According to the TIOBE Index², as of May 2012 PHP is the 6th most popular programming language. PHP5³ is an object-oriented language with an imperative core. The object system is based around a single-inheritance model with multiple inheritance of interfaces. The visibility mechanism uses the familiar keywords `public`, `private`, and `protected`, and works in conjunction with the inheritance mechanism (e.g., protected methods are visible in subclasses). As in C++, namespaces provide a mechanism for grouping user-defined names. Newer features include closures and traits [44]. PHP also includes extensive libraries, including standard functions for working with arrays, manipulating strings, and interacting with databases, and a number of third-party application frameworks and utility libraries are available.

We are interested in analyzing PHP code for several reasons. Since PHP has a weak typing model (“duck typing”) it allows many common errors to go undetected. One question this raises is: even with this weak typing model, how many of these errors can be detected, and with what accuracy? Another goal of this research is to detect possible errors caused by changes in the PHP semantics, especially in cases where syntactically updated PHP4 code is running on a PHP5 engine. Tools which analyze

¹ <http://www.php.net>

² <http://www.tiobe.com/index.php/content/paperinfo/tpci/>

³ As of May 2012, the current version is 5.4.3.

PHP and detect such problems are typically in the application domain of Rascal, as well as in the application domain of Maude.

First we describe some of the challenges, with possible solutions, in analyzing PHP. These include resolving includes, alias analysis and type inference.

4.1 Analyzing Includes, Aliases and Types for PHP

As part of our work on creating analysis tools for PHP, we are creating two analyses that will be used in many of the other tools: an alias analysis, and a type inferencer. These both work on individual PHP scripts, which can include other scripts as well as references to PHP library functions.

Includes. The first challenge in our analysis is to actually get a script’s complete source. Unlike in languages such as C, includes in PHP are resolved at runtime, with include paths that can be based on arbitrary expressions. In a worst-case scenario, this means that an include could refer to any other PHP file in the system. In practice, it is possible in many cases to statically resolve the include path using a number of techniques (constant propagation, algebraic simplification, path matching). Thus, for these analyses we assume we have a “fully inlined” script in which all includes have been merged in.

Type Inference. The type inferencer is based on the Cartesian product algorithm [1]. This algorithm assigns a set of types from the universe of all possible program types (classes, interfaces, and built-in types) to the expressions and *access paths* (paths made up of names, field accesses, and array element accesses) in the script. This type assignment is initially seeded with those cases where an invariant type can be determined. For instance, the expression `new T` always yields type $\{T\}$, and the literal `5` is always given type $\{int\}$. Using this seed, other types are derived using typing rules. The algorithm gets its name through its treatment of method calls: given the type sets assigned to the invocation target and the actual parameters, the Cartesian product of these sets is formed. For each element of this Cartesian product, the proper method (based on the target type) is selected, types are bound to the formal parameters, and the method body is then typed. Overloaded expressions, such as the arithmetic expressions, can be treated as special versions of methods and typed similarly.

Alias Analysis. The alias analysis is based on an interprocedural alias analysis algorithm that can handle function pointers, recursive calls, and references [25]. The alias analysis yields a relation between names at each program point, where the relation contains pairs of names which are may-aliases (i.e., which *may* refer to the same memory location). Aliases in PHP are created directly through reference assignments, the use of reference arguments and reference returns in functions and methods, and potentially through the use of the `global` statement. Aliases can also be created indirectly through the use of variable-variables, where (for instance) the name of a variable to access is stored as a string in another variable and “dereferenced” using a double dollar sign. Finally, indirect aliases are created through object references, which act as pointers to the referenced objects. Several of these cases are shown in Figure 4.

```

1 $g = 10;
2 function f1(&$p1) { $p1++; } // $p1 is a reference argument
3 function &f2() { global $g; return $g; } // reference return
4 class C1 { public $v1 = 5; }
5
6 $a = 3;
7 f1($a); // $a is now 4, $p1 aliases $a in f1
8 $b =& f2(); // $b now aliases $g
9 $c = new C1();
10 $d = $c; // $d and $c are not aliases, $c->v1 and $d->v1 are
11 $d =& $c; // $d and $c now are aliases
12 $vv = "a";
13 $$vv = 6; // $$vv aliases $a

```

Fig. 4. Creating Aliases in PHP

In order to get the correct results, the analyses need to be run in tandem, first running one, then the other, until a fixpoint is reached. This is because each analysis provides new information that can be used in the other. For instance, discovering that two names are aliased can add to the set of types assigned to a name, which could then add additional elements to the Cartesian products calculated for method calls. Also, expanding the set of types of an invocation target can expand the set of methods invoked at a specific call site, which can then lead to the generation of more alias pairs. While convergence of this process can be slow, the type inference and alias analysis algorithms work over finite sets of values and are monotonic, guaranteeing that they will terminate.

4.2 Required Analysis Tasks

A number of standard tasks are required for creating any PHP analysis. We discuss four of these below: parsing, maintaining internal representations needed in the analysis, writing the rules for the analysis, and reporting the analysis results. After this we show how we implemented these four tasks in Rascal and then we sketch out how we would implement them in Maude.

Parsing PHP Scripts. The purpose of executing a server-side PHP script (the standard mode of execution) is to generate an HTML page to send to a client. To make this easier for developers, PHP scripts are often a mixture of PHP code and fragments of HTML. Because of this, the parser needs to be capable of parsing intermingled PHP code and HTML markup. Whether to keep the HTML fragments is a decision based on the analysis – some analyses try to ensure that sensible HTML code is generated, while for other analyses it may be possible to discard the HTML. Since the analysis also needs to be able to handle includes properly (discussed above), parsing and later steps may also work in tandem, with new scripts parsed as new information on includes is discovered. The end result of parsing should be an internal representation of the parsed script(s) that can be used in the analysis.

Developing Internal Representations. Each analysis uses a (potentially large) number of intermediate representations. Some of these can be shared between different analyses, such as the representation of the script to analyze, while some are unique to each analysis. For the analyses here, this would include the representations of the results: sets of types assigned to expressions and access paths for the type inference analysis, and sets of alias pairs assigned to program points for the alias analysis. This would also include the intermediate representations, used during computation of the results, but often containing information that is not needed once the analysis is complete.

Writing Analysis Rules. The analysis rules interact with the internal program representation and the various supporting data structures to analyze the various language constructs, computing (for instance) the inferred type of a concatenation expression, the aliases created in a reference assignment, or the set of all alias pairs at a given program point.

Reporting Analysis Results. The analysis needs to provide a way to report the final analysis results. Based on the needs of the analysis clients, results could be provided using internal data structures (e.g., sets of alias pairs associated with a specific program point), external messages (visual indicators to show which types have been inferred for an expression), or some combination of the two.

4.3 Analyzing PHP in Rascal

Here we describe, in terms of the analysis tasks listed above, the current implementation of the PHP type and alias analyses we have developed in Rascal.

Parsing PHP Scripts. We are currently parsing PHP using a fork of an open-source PHP parser⁴. Our parsing script pretty-prints the parse tree as a term conforming to the AST representation we have in Rascal for PHP, with location information provided as Rascal annotations. This PHP script is called directly from Rascal using a library for interacting with the command shell. Parsing the abstract syntax tree representation to Rascal's internal algebraic data-types is also a standard library function.

We are also converting an SDF parser for PHP, written as part of the PHP-front project⁵, which will allow us to parse PHP code directly in Rascal. This will also create terms conforming to the Rascal PHP AST definition so we will not have to change existing code.

Developing Internal Representations. The PHP AST is defined as a mutually recursive collection of Rascal datatype declarations, with base types and collection types used to represent strings, integers, lists of parameters, etc. The internal configuration of the analysis is represented as an element of a Rascal algebraic datatype, with different fields holding different pieces of analysis information. Relations are used to represent intermediate results (e.g., the relation between an access path and possible types); final results are often instead given in maps, since these provide for quicker lookup performance, and can be stored directly on the abstract syntax tree using Rascal annotations.

⁴ <https://github.com/nikic/PHP-Parser/>

⁵ <http://www.program-transformation.org/PHP/PhpFront>

Writing Analysis Rules. Analysis rules are written as Rascal functions using a combination of switch statements and parameter-based dispatch. Each function takes at least the current configuration and a piece of abstract syntax, returning the updated configuration and (often) a partial analysis result, such as the type set computed for an expression.

The functions that use dynamic dispatch look like rewrite rules from a certain perspective, while the code that uses switch has a more procedural style. For example:

```
Infer plus(Cfg c, int(), int()) = <c, {int()}>;
Infer plus(Cfg c, int(), float()) { return <c, {float()}>; }
Infer plus(Cfg c, int(), str()) = <warn(c, msg), {int(), float()}>;
default Infer plus(Cfg c, Type l, Type r)
  = <err(c, ...), {int(), float()}>
```

This code defines the `plus` function in four independent declarations. Each definition is overloaded—mutually exclusively—using pattern matching. The second definition is written using a block of statements to demonstrate the two different styles of function definition. The last definition has the **default** keyword, which indicates that it will be tried only after the others have failed to match the parameters passed into the call. The definitions use different kinds of data-types, namely algebraic data types (the updated configuration returned from a call to `warn(c, msg)`), tuples (`<a, b>`), and sets (using the `{ . . . }` brackets).

The current implementation of the analysis rules has several limitations:

- The handling of “variable” constructs (variable-variables, variable functions, accesses to variable properties, etc) is currently too conservative, weakening the precision of the analysis. For instance, assigning a new type to a variable accessed through a variable-variable assigns this type to all variables currently in scope.
- The analysis can be overwhelmed as PHP scripts get larger, with both memory usage and processor usage growing to make the analysis infeasible. The imprecision in the handling of the variable constructs makes this worse, since this dramatically increases the size and number of type sets and alias pairs.
- Since the initial focus was on checking for upgrade problems between PHP4 and PHP5, some PHP5 features are not yet analyzed. This includes interfaces, traits, closures, and `gotos`, although some similar constructs (`break` and `continue` statements, essentially structured `gotos`) are currently supported.

Because of these limitations, we are currently reimplementing the analyses with a focus on performance. We are also working towards supporting the newer features of PHP that we do not yet support.

Reporting Analysis Results. As mentioned above, analysis results are recorded in annotations on the AST, and are also given back as part of the final configuration. The current model of sharing the results is to use annotations, since this allows arbitrary information to be added to each node in the AST.

We do not yet show types for PHP expressions and access paths in a graphical fashion (e.g., as hovers within the Eclipse IDE), but that would be the next step if we were constructing an IDE for PHP. To display errors and warnings, the easiest way is to register them via the Rascal standard library module that gives access to the Eclipse Problem

View. A typical low-brow way of producing readable results from the annotated AST would be to use Rascal’s string templates to produce a readable list of error messages and warnings.

4.4 Analyzing PHP in Maude

Here we describe what would be needed to build rewriting logic semantics versions of the type and alias analyses described above. We assume that these analyses would be run using Maude.

Parsing PHP Scripts. Since the Maude parser is not capable of parsing normal PHP scripts, we instead would need to use an external parser. We may use the same external parser that Rascal used before: the parser would generate terms, in prefix form, using an algebraic signature defined to represent the abstract syntax of PHP. Some of these terms would be “located”, as described in Section 3 with the RLSRunner tool, allowing source location information to be reflected in any generated error messages. Connecting the PHP front-end with Maude is done most straightforwardly by creating a shell script wrapping a call to the parser and sending the result over a pipe to Maude.

Developing Internal Representations. Using Maude, all internal representations are based around terms formed over an algebraic signature. The abstract syntax for PHP would most likely be given in mixfix, allowing rules⁶ to be written over an abstract syntax that looks similar to the concrete syntax. Intermediate results and the final results would also be defined as terms, representing (for instance) sets of types or maps from program points to alias pairs.

In a computation-based rewriting logic semantics [35], the current configuration – i.e., the current state of the analysis, including all intermediate results – would also be defined algebraically, as a multiset of nested *cells*. These cells can contain information such as the current computation (i.e., the remaining steps of the analysis), the current map of program names to values or storage locations, or the set of available class definitions.

Writing Analysis Rules. Using a computation-based RLS, analysis rules are written as transformations of the configuration, generally involving the computation. Most language features are handled by several rules: one rule breaks apart the language construct to evaluate its pieces, while the others use the evaluation results to compute a result for the entire construct. As an example, a type inferencer may include rules such as:

```
eq k(exp(E + E') -> K) = k(exp(E,E') -> + -> K) .
eq k(val(int,int) -> + -> K) = k(val(int) -> K) .
eq k(val(int,float) -> + -> K) = k(val(float) -> K) .
eq k(val(int,str)-> + -> K) = k(warn(...)-> val(int float)-> K) .
```

The first rule indicates that we need to evaluate expressions E and E' before we can compute the result of $E + E'$; E and E' are put at the front of the computation (k) to

⁶ We speak here of rules in the generic sense, including both equations and rewriting logic rules.

indicate they should be evaluated next, while `+` is put in the computation as a marker to indicate the operation being performed. The second, third, and fourth rules then give several possible behaviors, based on the results of evaluating the operands. The second rule says that, if both operands are of type `int`, so is the result. The third rule says that, if the first is `int` and the second is `float`, the overall result is `float`. Finally, the last rule shown says that, if we are trying to add an `int` to a `string` (as occurs in examples in Figure 4), we should issue a warning (the text is elided as `. . .`), because this may not be the operation we were intending to perform. We should also then return a set (represented by juxtaposition) containing both `int` and `float`, since, based on the contents of the string, both are possible result types.

Reporting Analysis Results. There are several options for reporting the results of the analysis. The most basic is to just examine the final configuration, which will have the type assignments, alias pairs, etc. inside it. We could also use the semantics to perform a final “pretty printing” step to provide the results in string form. Other options include the use of external tools to view the results, such as was done with the RLSRunner tool in Section 3. Finally, we could return just a term with the results, not the entire configuration. This term could be used as input into another RLS-based tool which needed the computed results.

5 Related Work

In earlier work [45] we discussed the evolution of Rascal from its origins in the ASF+SDF and RSCRIPT systems. Some of the material in this paper, especially in Section 2, is based on this work, including the Rascal examples shown for illustration. Below we list other related work for Rascal, for PHP program analysis, and for analysis using rewriting logic.

Rascal: The design of Rascal is based on inspiration from many earlier languages and systems. The syntax features (grammar definition and parsing) are directly based on SDF [18], but the notation has changed and the expressivity has been increased. The features related to analysis are mostly based on relational calculus, relational algebra and logic programming systems such as Crocopat [5], Grok [26] and RSCRIPT [31], with some influence from CodeSurfer [2]. Rascal has strongly simplified backtracking and fixed point computation features reminiscent of constraint programming and logic programming systems like Moreau’s Choice Point Library [36], Prolog and Datalog [8]. Rascal’s program transformation and manipulation features are most directly inspired by term rewriting/functional languages such as ASF+SDF [48], Stratego [6], TOM [3], and TXL [12]. The ATerm library [47] inspired Rascal’s immutable values, while the ANTLR tool-set [39], Eclipse IMP [9] and TOM [3] have been an inspiration because of their integration with mainstream programming environments.

PHP Analysis: Most research on PHP analysis has focused on detecting security vulnerabilities, including SQL injection attacks, cross-site scripting, and the use of tainted data (data that comes from outside the program, such as from a user form,

and that is not checked before being used in file writes, database queries, etc). This is the main focus of both the WebSSARI [27][28] and the Pixy [30][29] systems and of a number of individual analyses [41]. The PHP-sat⁷ and PHP-tools⁸ projects extend this security validation research by also adding support for additional analyses, including detecting a variety of common bug patterns (e.g., assigning the result of a function call where the body of the called function does not include a return statement), and by finding some PHP4 to PHP5 migration errors (e.g., functions with names that match new PHP5 functions). Another tool, the prototype PHP Validator [7], uses a type inferencer as part of a number of possible analyses. For instance, one example given is an analysis to detect the accidental use of + instead of . in string concatenation. However, not all of the analyses listed are actually implemented, and the PHP Validator tool does not support the object-oriented features of PHP.

Analysis in Rewriting Logic: Rewriting logic has been used extensively for program analysis. The work most similar to that discussed here is the work on policy frameworks for C [19] and SILF [24]. This work, in turn, was based on earlier work on detecting units of measurement errors in C [42] and BC [10] programs. Taking another approach, JavaFAN [15] uses Maude’s state space search and LTL model checking facilities [13] to find program errors, including possible deadlocks in concurrent programs. A semantics of C [14], developed using K [43], uses standard Maude rewriting to run C programs and look for undefined behavior; state space search and model checking are used to explore the nondeterminism introduced by constructs with undefined evaluation orders.

6 Summary and Discussion

We have shown a range of scenarios for building new software analysis tools in Rascal: from a purely Rascal-based solution to solutions that make use of external tools. This has been illustrated through a number of examples, including one showing the integration of Rascal with an existing rewriting logic-based analysis. The most substantial example has been the analysis of PHP code, in which we outlined the necessary steps for such an analysis, their implementation in Rascal, and a speculative implementation in Maude. We now summarize our observations and conclusions.

6.1 Observations Regarding Rascal

Rascal is a programming language that provides all the features and libraries needed to create end-to-end software analysis tools. Although it has its roots in algebraic specification [4] and can be used to write programs that strictly follow the rewriting paradigm, it shows its major strengths in developing Rascal-based or hybrid solutions that cooperate with an IDE or with external tools.

⁷ <http://www.program-transformation.org/PHP/PhpSat>

⁸ <http://www.program-transformation.org/PHP/PhpTools>

Table 3. Comparison of Maude and Rascal

Aspect	Maude	Rascal
Computational Model	Simpler	More complex
Semantics	Formal	Informal
Extensibility (language level)	Yes	No
Integration external tools	Only sockets, pipes	Java & Eclipse
Grammar definition & parsing	Limited	Fully integrated
Data types	Limited	Rich
Libraries	Limited	Rich
Maturity	Mature framework	Young language
Efficiency	Optimized	Not yet optimized

6.2 Comparing Maude and Rascal: General Observations

Our global experience with the two languages makes clear that both Maude and Rascal have competing benefits and shortcomings, which we summarize in Table 3.

Maude is closer to semantic foundations, with a formal semantics and a rewriting-based computational model. This makes the analysis itself amenable to formal techniques and provides a simpler conceptual core. Maude provides flexible extensibility mechanisms and (by treating computations as first class entities) powerful ways to manipulate partial computations. This can be especially useful when analyzing features that “jump”, such as `gotos`, exceptions, and loop break and continue statements. Maude is a mature framework, with an optimized implementation, but it can still be challenging to write and maintain large specifications. Debugging tools for Maude [40] have made great progress but still have trouble scaling to large specifications, which often leads to debugging by reading through rewriting traces.

Rascal is more pragmatic, with a more complex, unformalized computational model. The Rascal language definition framework provides very strong integrated grammar definition and parsing facilities, rich data types that provide element generation, and traversal and pattern matching constructs. For integration, Rascal also provides rich libraries (e.g., supporting visualization, statistics, and data formats like HTML, XML, CSV, and SVN) and seamless extension and integration facilities with Java, Eclipse and external tools. Since Rascal is a young language, its design is not yet completely finalized and its implementation is not yet optimized, leading to potential performance problems when tackling large analysis problems.

6.3 Comparing Maude and Rascal: Observations for PHP Analysis

More specific differences can be identified on the basis of the PHP analysis.

Parsing PHP Scripts. Here Rascal is clearly superior since both a completely Rascal-based parser and an externally implemented parser can be used. Also, more control is possible over the shape of the resulting trees.

Developing Internal Representations. Rascal provides many more constructs, like maps, trees, and n-ary tuples and relations, “out of the box”, making it easier to quickly develop the needed internal representations and to treat them as black boxes.

Writing Analysis Rules. One perspective is that the same term and rule-based style can be used for writing specifications in both approaches. Specifications written in this style (this would be a functional style of programming in Rascal) are more amenable to formal analysis. This could be useful if, for instance, one wanted to prove that the type inference algorithm does not infer incorrect types. However, given the informal nature of PHP, it is not clear how possible it would be to conduct such proofs. Another perspective is that, if this level of rigor isn’t needed, Rascal provides a number of programming language features that enable a richer variety of programming styles. Jumps can be modeled in Maude using K-style computations and in Rascal using either edges in graph-like program representations or using higher-order functions.

Reporting Analysis Results. Maude and Rascal can both return an annotated term or string representing the analysis results. Rascal also includes facilities for error reporting and IDE integration built directly into the language.

6.4 Final Observations

We can draw several conclusions from this comparison. The bottom-line is that Maude is focused on formal specification while Rascal is focused on programming.

Maude is a better choice when the formal properties of the implemented tools are also important. Maude may be a better choice when a language has a number of jump-like constructs; these can be handled in Rascal, but require making the control context more explicit. Rascal is a better choice for end to end solutions, for instance real language environments, where parsing, integration with IDEs, and integration with external tools becomes important. With more standard control flow mechanisms, richer built-in data types, and a provided unit test definition and execution mechanism, large Rascal programs should also be easier to debug than large rewriting logic specifications for the same analysis. Finally, as shown with the RLSRunner tool, in some cases it may make sense to write analyses partly in Rascal and partly in Maude, such as when an analysis semantics already exists, or can be derived as an extension of an existing semantics, and can then be used as part of a Rascal-developed language environment.

References

1. Agesen, O.: The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 2–26. Springer, Heidelberg (1995)
2. Anderson, P., Zarins, M.: The CodeSurfer Software Understanding Platform. In: Proceedings of IWPC 2005, pp. 147–148. IEEE (2005)
3. 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)

4. Bergstra, J., Heering, J., Klint, P.: Algebraic Specification. ACM Press (1989)
5. Beyer, D.: Relational programming with CrocoPat. In: Proceedings of ICSE 2006, pp. 807–810. ACM Press (2006)
6. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming* 72(1-2), 52–70 (2008)
7. Camphuijsen, P.: Soft typing and analyses of PHP programs. Master's thesis, Universiteit Utrecht (2007)
8. Ceri, S., Gottlob, G., Tanca, L.: Logic programming and databases. Springer-Verlag New York, Inc. (1990)
9. Charles, P., Fuhrer, R.M., Sutton Jr., S.M., Duesterwald, E., Vinju, J.J.: Accelerating the Creation of Customized, Language-Specific IDEs in Eclipse. In: Proceedings of OOPSLA 2009, pp. 191–206. ACM Press (2009)
10. Chen, F., Roşu, G., Venkatesan, R.P.: Rule-Based Analysis of Dimensional Safety. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 197–207. Springer, Heidelberg (2003)
11. 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)
12. Cordy, J.R.: The TXL source transformation language. *Science of Computer Programming* 61(3), 190–210 (2006)
13. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL Model Checker. In: Proceedings of WRLA 2002. ENTCS, vol. 71. Elsevier (2002)
14. Ellison, C., Roşu, G.: An Executable Formal Semantics of C with Applications. In: Proceedings of POPL 2012, pp. 533–544. ACM Press (2012)
15. Farzan, A., Chen, F., Meseguer, J., Roşu, G.: Formal Analysis of Java Programs in JavaFAN. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 501–505. Springer, Heidelberg (2004)
16. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (2000)
17. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
18. Heering, J., Hendriks, P., Klint, P., Rekers, J.: The syntax definition formalism SDF - reference manual. *SIGPLAN Notices* 24(11), 43–75 (1989)
19. Hills, M., Chen, F., Roşu, G.: A Rewriting Logic Approach to Static Checking of Units of Measurement in C. In: Proceedings of RULE 2008. Elsevier (2008) (to appear)
20. Hills, M., Şerbănuţă, T.F., Roşu, G.: A Rewrite Framework for Language Definitions and for Generation of Efficient Interpreters. In: Proceedings of WRLA 2006. ENTCS, vol. 176, pp. 215–231. Elsevier (2007)
21. Hills, M., Klint, P., van der Storm, T., Vinju, J.J.: A Case of Visitor versus Interpreter Pattern. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 228–243. Springer, Heidelberg (2011)
22. Hills, M., Klint, P., Vinju, J.J.: RLSRunner: Linking Rascal with K for Program Analysis. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 344–353. Springer, Heidelberg (2012)
23. Hills, M., Klint, P., Vinju, J.J.: Scripting a Refactoring with Rascal and Eclipse. In: Proceedings of WDT 2012. ACM Press (to appear, 2012)
24. Hills, M., Roşu, G.: A Rewriting Logic Semantics Approach To Modular Program Analysis. In: Proceedings of RTA 2010. Leibniz International Proceedings in Informatics, vol. 6, pp. 151–160. Schloss Dagstuhl - Leibniz Center of Informatics (2010)

25. Hind, M., Burke, M.G., Carini, P.R., Choi, J.-D.: Interprocedural Pointer Alias Analysis. *ACM TOPLAS* 21(4), 848–894 (1999)
26. Holt, R.C.: Grokking Software Architecture. In: *Proceedings of WCRE 2008*, pp. 5–14. IEEE (2008)
27. Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., Kuo, S.-Y.: Securing Web Application Code by Static Analysis and Runtime Protection. In: *Proceedings of WWW 2004*, pp. 40–52. ACM Press (2004)
28. Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.T., Kuo, S.-Y.: Verifying Web Applications Using Bounded Model Checking. In: *Proceedings of DSN 2004*, pp. 199–208. IEEE (2004)
29. Jovanovic, N., Kruegel, C., Kirda, E.: Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In: *Proceedings of PLAS 2006*, pp. 27–36. ACM Press (2006)
30. Jovanovic, N., Krügel, C., Kirda, E.: Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In: *IEEE Symposium on Security and Privacy*, pp. 258–263 (2006)
31. Klint, P.: Using Rscript for Software Analysis. In: *Working Session on Query Technologies and Applications for Program Comprehension, QTAPC 2008* (2008)
32. Klint, P., van der Storm, T., Vinju, J.: RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In: *Proceedings of SCAM 2009*, pp. 168–177. IEEE (2009)
33. Klint, P., van der Storm, T., Vinju, J.: EASY Meta-programming with Rascal. In: *Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491*, pp. 222–289. Springer, Heidelberg (2011)
34. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science* 96(1), 73–155 (1992)
35. Meseguer, J., Roşu, G.: The rewriting logic semantics project. *Theoretical Computer Science* 373(3), 213–237 (2007)
36. Moreau, P.-E.: A choice-point library for backtrack programming. In: *JICSLP 1998 Post-Conference Workshop on Implementation Technologies for Programming Languages based on Logic* (1998)
37. Opdyke, W.F.: *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign (1992)
38. Opdyke, W.F., Johnson, R.E.: Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In: *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications* (1990)
39. Parr, T.: *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf (2007)
40. Riesco, A., Verdejo, A., Martí-Oliet, N.: A Complete Declarative Debugger for Maude. In: *Johnson, M., Pavlovic, D. (eds.) AMAST 2010. LNCS, vol. 6486*, pp. 216–225. Springer, Heidelberg (2011)
41. Rimsa, A., d’Amorim, M., Pereira, F.M.Q.: Tainted Flow Analysis on e-SSA-Form Programs. In: *Knoop, J. (ed.) CC 2011. LNCS, vol. 6601*, pp. 124–143. Springer, Heidelberg (2011)
42. Roşu, G., Chen, F.: Certifying Measurement Unit Safety Policy. In: *Proceedings of ASE 2003*, pp. 304–309. IEEE (2003)
43. Roşu, G., Şerbănuţă, T.F.: An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79(6), 397–434 (2010)
44. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable Units of Behaviour. In: *Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743*, pp. 248–274. Springer, Heidelberg (2003)
45. van den Bos, J., Hills, M., Klint, P., van der Storm, T., Vinju, J.J.: Rascal: From Algebraic Specification to Meta-Programming. In: *Proceedings of AMMSE 2011. EPTCS, vol. 56*, pp. 15–32 (2011)

46. van den Brand, M., Bruntink, M., Economopoulos, G., de Jong, H., Klint, P., Kooiker, T., van der Storm, T., Vinju, J.: Using The Meta-environment for Maintenance and Renovation. In: Proceedings of CSMR 2007, pp. 331–332. IEEE (2007)
47. van den Brand, M., de Jong, H., Klint, P., Olivier, P.: Efficient Annotated Terms. *Software, Practice & Experience* 30, 259–291 (2000)
48. den van Brand, M.G.J., van Deursen, A., Heering, J., de Jong, H.A., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 365–370. Springer, Heidelberg (2001)

℔ Framework Distilled*

Dorel Lucanu¹, Traian Florin Șerbănuță^{1,2}, and Grigore Roșu^{1,2}

¹ Faculty of Computer Science

Alexandru Ioan Cuza University, Iași, Romania

`dlucanu@info.uaic.ro`

² Department of Computer Science

University of Illinois at Urbana-Champaign, USA

`grosu@illinois.edu`

Abstract. ℔ is a rewrite-based executable semantic framework in which programming languages, type systems, and formal analysis tools can be defined using configurations, computations and rules. Configurations organize the state in units called cells, which are labeled and can be nested. Computations are special nested list structures sequentializing computational tasks, such as fragments of program. ℔ (rewrite) rules make it explicit which parts of the term they read-only, write-only, read-write, or do not care about. This makes ℔ suitable for defining truly concurrent languages even in the presence of sharing. Computations are like any other terms in a rewriting environment: they can be matched, moved from one place to another, modified, or deleted. This makes ℔ suitable for defining control-intensive features such as abrupt termination, exceptions or call/cc. This paper presents an overview of ℔ Framework and the ℔ tool, focusing on the interaction between the ℔ tool and Maude.

1 Introduction

Introduced by the second author in 2003 for teaching programming languages [1], and continuously refined and developed ever since (see, e.g., [2,3]), ℔ is a programming language definitional framework which aims to bring together the collective strengths of existing frameworks (expressiveness, modularity, concurrency, and simplicity) while avoiding their weaknesses. The ℔ framework has already been used to define real-life programming languages, such as C, Java, Scheme, and several program analysis tools (see Section 7 for references). ℔ is representable in rewriting logic, and this representation has been automated in the ℔ tool for execution, testing and analysis purposes using Maude [4].

This paper gives a brief overview of the ℔ framework and its current implementation, focusing on: (1) its place in the rewriting logic semantics [5,6,7] project; (2) its main features; (3) how easy it is to define programming language features in ℔; (4) how to use the ℔ tool to compile ℔ definitions into Maude, and to execute and analyze programs against these definitions.

* This work is supported by Contract 161/15.06.2010, SMISCSNR 602-12516 (DAK).

The remainder of this paper is organized as follows. Section 2 motivates \mathbb{K} and places it in the general programming language semantics research context, and in particular within the rewriting logic semantics project. Section 3 gives an overview of the main features of \mathbb{K} . Section 4 shows \mathbb{K} at work, by giving a compact semantics for a combination of the call-by-value and call-by-reference parameter passing styles. Section 5 discusses the transition semantics associated to a \mathbb{K} definition and its relation to our current embedding of \mathbb{K} into rewriting logic. Section 6 shows how the embedding of \mathbb{K} into Maude through the \mathbb{K} tool can be used to execute, explore, and model check programs. Section 7 concludes.

The didactic language CinK [8] is used as a running example.

2 Rewriting Logic Semantics, Related Work, Motivation

The research presented in this paper is part of the rewriting logic semantics project [5,6,7], an international collaborative effort to advance the use of rewriting logic for defining programming languages and for analyzing programs.

Rewriting is an intuitive and simple mathematical paradigm which specifies the evolution of a system by matching and replacing parts of the system state according to *rewrite rules*. Besides being formal, rewriting is also executable, by simply repeating the process of rewriting the state. Additionally, an initial state together with a set of rules yields not only a formal execution of the system, but also a transition system comprising all the system behaviors, which can be thus formally analyzed. Moreover, a rewriting semantic definition of a language can also be used to (semi-)automate the verification of programs written in that language, by using the semantic rules to perform symbolic execution of the program and hereby discharging (some of) the proof obligations.

Rewriting logic [9] combines term rewriting and equational logic in a formalism suitable to define truly concurrent systems. Equations typically define structural identities between states, and rewrite rules apply modulo equational rearrangements of the state. The benefits of using rewriting logic in defining the behavior of systems are multiple. First, one directly gains executability, and thus the ability to directly use formal definitions as interpreters. Second, it allows to capture the intended concurrency of the defined system directly in the definition, rather than relying on subsequent abstractions. Furthermore, by encoding the deterministic rules of a rewrite system as equations [10], the state-space of the resulting transition systems is drastically reduced, thus making its exploration more feasible and practical. The Maude rewrite system [4] offers a suite of tools for rewrite theories: debugger, execution tracer, state-space explorer, explicit-state LTL model checker, inductive theorem prover, etc. For example, model checking Java programs in Maude using a definition of Java, following the \mathbb{K} technique presented here, was shown to compare favorably [11] with Java PathFinder, the state-of-art explicit-state model checker for Java [12].

When defining a language semantics in rewriting logic, the program state is typically represented as a configuration term. Equations represent structural rearrangements of the configuration or behaviorally irrelevant computational steps.

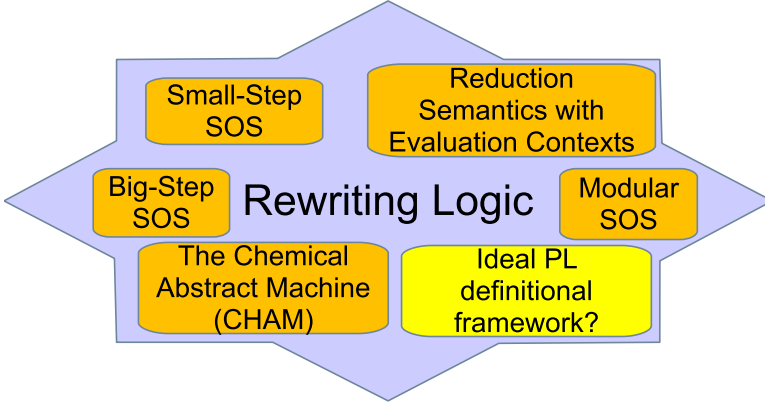


Fig. 1. Rewriting logic as a meta-logical framework for defining programming languages

Rewrite rules capture the relevant computational steps, namely those which we want to count as actual transitions between states. This way, a program execution is captured as a sequence of transitions between equivalence classes of configuration terms, and the state-space of executions is captured as the transition system defined by the rewrite rules. Several paradigmatic languages have been given a faithful rewriting logic semantics this way [9,13,14], and even some small programming languages, following different styles and methodologies. In fact, [15] shows how various operational semantics approaches can be framed as methodological fragments of rewriting logic, including big-step (or natural) semantics [16], (small-step) structural operational semantics (SOS) [17], Modular SOS (MSOS) [18], reduction semantics (with evaluation contexts) [19], continuation-based semantics [20], and the chemical abstract machine (CHAM) [21].

Thus, we can regard rewriting logic as a meta-framework for defining programming language semantics, as illustrated in Figure 1. Once a language is defined in rewriting logic, the arsenal of generic tools of the latter can then be used to formally analyze both the programming language itself as well as its programs. An advantage that rewriting logic offers over other similar powerful meta-frameworks, such as e.g., higher-order logic, is that it allows us to tune the computational granularity of the defined language, both in depth (when we want several small steps to count as one step) and in breadth (when we want several non-overlapping steps to proceed concurrency), with little or no effort.

Unfortunately, one pragmatic problem that all meta-frameworks share is that they do not tell us *how* to define a language. They only give us powerful means to faithfully and uniformly represent any semantics, following any approach, using the same formalism. This desirable faithfulness actually also implies that a meta-framework, no matter how powerful it is, cannot magically eliminate the inherent limitations of the chosen semantic approach. For example, existing approaches have problems with control-intensive features (except for evaluation contexts), with modularity (except for MSOS, and except for evaluation contexts in some cases), with true concurrency (except for the CHAM), and so on.

Ideally, we would like a semantic approach, or framework, which has at least the union of all the strengths of the existing approaches, and which at the same time avoids all their weaknesses. Such a framework would also likely be representable in powerful meta-frameworks such as rewriting logic or higher-order logic, but that is not the point here. The point is that it is not clear whether such a framework is possible. In particular, such a framework should be at least as expressive as reduction semantics with evaluation contexts, but should also allow non-syntactic, environment/store style definitions; it should be at least as modular as MSOS, but should also give us access to the execution/evaluation context; it should be at least as concurrent as the CHAM, but should not force us artificially encode everything in molecules and solutions; and so on. Whether \mathbb{K} has all these desirable features is and probably will always be open for debate. Nevertheless, \mathbb{K} has been from the very beginning designed in a bottom-up fashion, striving to incorporate the positive aspects of the existing approaches and to avoid their negative aspects, at the same time being based on a rigorous mathematical foundation and offering an intuitive notation to its users.

3 The \mathbb{K} Framework

In a nutshell, the \mathbb{K} framework consists of computations, configurations, and rules. *Computations* are special sequences of tasks, where a task can be, e.g., a fragment of program that needs to be processed. *Configurations* are organized as nested soups of cells that hold syntactic and semantic information. \mathbb{K} *rules* distinguish themselves by specifying only what is needed from a configuration, and by clearly identifying what changes, and thus, being more concise, more modular, and more concurrent than regular rewrite rules.

The running example of this paper is CinK [8], an overly-simplified kernel of the C++ language including integer and boolean expressions, functions, and basic imperative statements. Without modifying anything but the configuration, the language is extended with the following concurrency constructs: thread creation, lock-based synchronization and thread join.

Configurations. The initial running configuration of CinK is presented in Figure 2. The configuration is a nested multiset of labeled cells, in which each cell can contain either a list, a set, a bag, a map, or a computation. The initial CinK configuration consists of a top cell, labeled “T”, holding a bag of cells, among which a map cell, labeled “store”, to map locations to values, a list cell, labeled “in”, to hold input values, and a bag cell, labeled “threads”, which can hold any number of “thread” cells (signaled by the star “*” attached to the label of the cell). The thread cell is itself a bag of cells, among which the “k” cell holds a computation structure, which plays the role of directing the execution.

Syntax and Computations. Computations extend the user-defined language syntax with a task sequentialization operation, “ \curvearrowright ”. The basic unit of computation is a task, which can be either a fragment of syntax, possibly with holes in it, or a semantic task, such as an environment recovery. Most of the manipulation of the

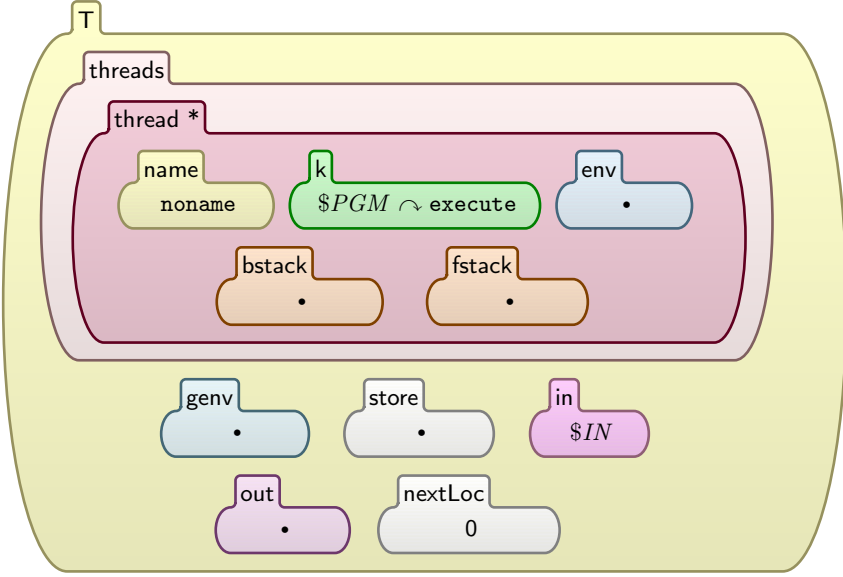


Fig. 2. The initial configuration of the CinK language

computation is abstracted away from the language designer via intuitive PL syntax annotations like strictness constraints which, when declaring the syntax of a construct also specify the order of evaluation for its arguments. Similar decompositions of computations happen in abstract machines by means of stacks [20], and also in the refocusing techniques for implementing reduction semantics with evaluation contexts [22]. However, what is different here is that \mathbb{K} achieves the same thing *formally*, by means of rules (there are special rules behind the strictness annotations, as explained below), not as an implementation means.

The \mathbb{K} BNF syntax specified below suffices to parse the program fragment “ $t = *x; *x = *y; *y = t;$ ” specifying a sequence of statements for swapping the values at two memory locations:

```

SYNTAX  Exp ::= Id
          | * Exp [strict]
          | Exp = Exp [strict(2)]
SYNTAX  Stmt ::= Exp ; [strict]
           | Stmt Stmt [seqstrict]
    
```

Strictness annotations add semantic information to syntax by specifying the order of evaluation of arguments. The special rules corresponding to these strictness annotations are a special case of *structural rules*, metaphorically called heating/cooling rules like in the CHAM with the one going from left to right called a heating rule and the one from right to left called a cooling rule, are:

$$\begin{aligned}
* ERed &\rightleftharpoons ERed \curvearrowright * \square \\
E = ERed &\rightleftharpoons ERed \curvearrowright E = \square \\
ERed ; &\rightleftharpoons ERed \curvearrowright \square ; \\
SRed S &\rightleftharpoons SRed \curvearrowright \square S \\
Val SRed &\rightleftharpoons SRed \curvearrowright Val \square
\end{aligned}$$

The heating/cooling rules specify that the arguments mentioned in the strictness constraint can be taken out for evaluation at any time and plugged back into their original context. Note that statement composition generates two such rules (as, by default, strictness applies to each argument); however, since the constraint specifies *sequential strictness*, the second statement can be evaluated only once the first statement was completely evaluated (specified by the *Val* variable which should match a value) and its side effects were propagated.

By successively applying the heating/cooling rules above on the statement sequence above, we obtain the following (structurally equivalent) computations:

$$\begin{aligned}
& t = * x ; * x = * y ; * y = t ; \rightleftharpoons \\
& t = * x ; \curvearrowright \square * x = * y ; * y = t ; \rightleftharpoons \\
& t = * x \curvearrowright \square ; \curvearrowright \square * x = * y ; * y = t ; \rightleftharpoons \\
& * x \curvearrowright t = \square \curvearrowright \square ; \curvearrowright \square * x = * y ; * y = t ; \rightleftharpoons \\
& x \curvearrowright * \square \curvearrowright t = \square \curvearrowright \square ; \curvearrowright \square * x = * y ; * y = t ;
\end{aligned}$$

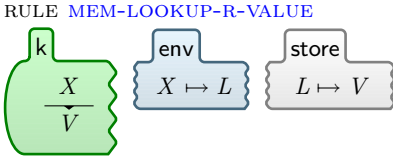
The heating rules thus pull redexes out from their context for evaluation according to the desired evaluation strategy of the corresponding constructs, leaving holes as placemarkers for where to plug their results or intermediate computations back using the cooling rules. Above, the heating rules eventually singled out the variable x at the top of the computation. As seen shortly, other rules can now match it and replace it with its corresponding value from the store. The cooling rules can then plug that value back into its place in context.

Implementations can choose to keep computations heated as an optimization, and only cool by need and only as much as necessary. Nevertheless, from a theoretical perspective, heating/cooling rules can be applied at any time and as many times as they match, thus yielding a potentially exponential number of structurally equivalent computations. As seen in Section 6, these can lead to non-deterministic behaviors of programs.

\mathbb{K} rules. As discussed above, \mathbb{K} has a particular kind of rules, called structural rules, which allow us to rearrange the configuration. Heating/cooling rules are a special kind of structural rules, which are typically bidirectional. \mathbb{K} also allows standalone structural rules, for example a rule desugaring a “for” loop into a “while”, which need not be reversible. The distinction between heating/cooling rules and other structural rules is purely methodological, with no semantic implications. Because of that, one should feel free to call other pairs of structural rules, which do not necessarily capture evaluation strategies, also heating/cooling. For example, pairs of structural rules corresponding to intended equations (“heat” $A*(B+C)$ into $A*B+A*C$, and “cool” $A*B+A*C$ into $A*(B+C)$).

In addition to structural rules, ℕ also has *computational rules*. The distinction between structural and computational rules is purely semantic, and will be clarified in Section 5. Intuitively, only the computational rules yield transitions in the transition system associated to a program. The role of the structural rules is to only rearrange the configuration so that computational rules can match.

The computational rule below succinctly describes the intuitive semantics for reading the value of a variable: if variable X is the next thing to be evaluated and if X is mapped to a location L in the environment, and that location is mapped to a value V in the store, then replace that occurrence of X by V . Moreover, note that the rule only specifies what is needed from the configuration, which is essential for obtaining modular definitions, and by precisely identifying what changes, which significantly enhances modularity and concurrency.



There are several ways in which ℕ rules differ from regular rewrite rules. First, *in-place rewriting* allows one to specify small changes into a bigger context, by underlining the part that needs to change and writing its replacement under the line, instead of repeating the context in both sides of a rewrite rule. This additionally gives us the ability of using anonymous variables for the unused variables in the context, and, furthermore, the use of *cell comprehension* for focusing only on the parts of the cells which are relevant for this rule. Our metaphorical notation for cell comprehension is the jagged cell edge, which thus specifies that there could be more items in the cell, in the corresponding side, in addition to what is explicitly specified. Finally, the process of *configuration abstraction* allows for only the relevant cells to be mentioned in a rule, relying on the static structure of the declared configuration to infer the rest.

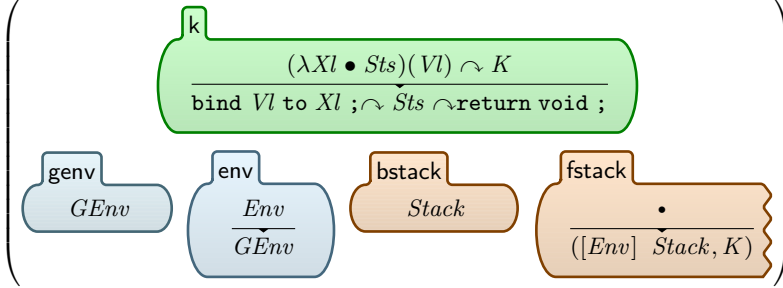
Modularity. Configuration abstraction is crucial for modularity. Relying on the initial configuration to be specified by the designer, and on the fact that usually the structure of such a configuration does not change during the execution of a program, the ℕ rules are essentially invariant under change of configuration structure. This effectively means that the same rule can be re-used in different definitions as long as the required cells are present, regardless of the additional context, which can be automatically inferred from the initial configuration.

Expressiveness. The particular structure of ℕ computations, and the fact that the current task is always at the top of the computation, greatly enhances the expressiveness of the ℕ framework. Next paragraphs show how easy it is to use ℕ to define constructs which are known to be hard to define in other frameworks.

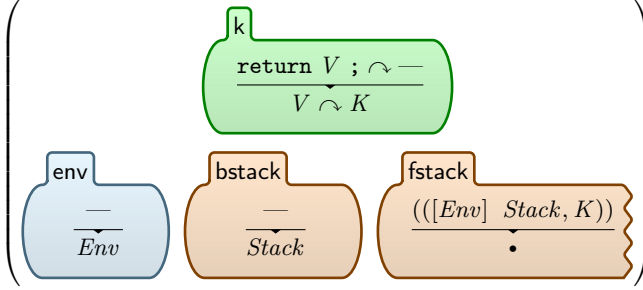
Abrupt returning from a function is hard to define in many frameworks (except for reduction semantics with evaluation contexts), due to the lack of explicit

access to the execution context. Having the entire remainder of computation always following the current redex allows the \mathbb{K} definition of CinK to capture this construct in a simple and succinct manner by the following two rules:

RULE FUNCTION-CALL



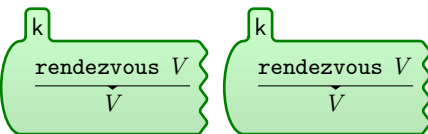
RULE RETURN



The function name is evaluated to its value, which is a lambda abstraction: Xl is the list of parameters, Sts is body of the function. The FUNCTION-CALL rule pushes the calling context, i.e., the remainder of the computation K and environment stack (including the current environment) on top of the function stack, while the RETURN rule uses the information there to restore the environment and computation of the caller. The evaluation of the arguments Vl and their binding to the formal parameters is described in Section 4.

Another feature which is hard to represent in other frameworks is handling multiple tasks at the same time, as when defining synchronous communication, for example. Although SOS-based frameworks can capture specific versions of this feature for languages like CCS or the π -calculus, they can only do it there because the communication commands are always at the top of their processes. \mathbb{K} computation's structure is again instrumental here, as it allows to easily match two redexes at the same time, as shown by the following rule, defining the semantics of a **rendezvous** expression used for synchronizing two threads:

RULE RENDEZVOUS



Reading this rule one can easily get the intended semantics: a thread requesting a `rendezvous` has to wait until another thread makes a request with the same value V ; once that happens, both threads can continue with V as their result.

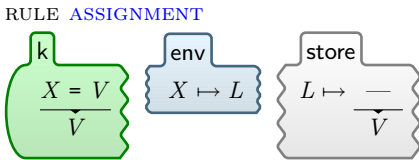
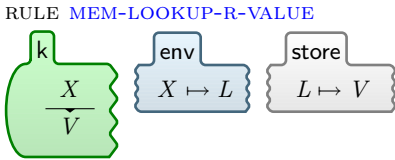
ℕ has a reflective view of syntax. Although it allows us to use concrete syntax in definitions as a convenience, it regards all syntactic terms as abstract syntax trees (AST). Thus, language constructs are regarded as AST labels. For example, $a + 3$ is represented in ℕ as $_ + _ (a(\bullet_{List\{K\}}), 3(\bullet_{List\{K\}}))$. This abstract view of syntax allows reducing the computation constructs to the following core:

$$\begin{array}{l}
 \text{SYNTAX } K ::= KLabel(List\{K\}) \\
 \quad \quad \quad \left| \begin{array}{l} \bullet_K \\ K \curvearrowright K \end{array} \right. \\
 \text{SYNTAX } List\{K\} ::= K \\
 \quad \quad \quad \left| \begin{array}{l} \bullet_{List\{K\}} \\ List\{K\}, List\{K\} \end{array} \right.
 \end{array}$$

We won't go into details here, but the ability of referring to the ℕ AST in a definition allows ℕ to define powerful reflective rules for AST manipulation such as generic AST visitor patterns, code generation, or generic substitution [23].

Concurrency. An aspect that makes ℕ appropriate for defining programming languages is its natural way to capture concurrency. Besides being truly concurrent (like CHAM), ℕ also allows capturing concurrency with resource sharing.

Let us exemplify this concurrency power. The two rules below specify the semantics for accessing/updating the value at a memory location:



As the semantics of the ℕ rules specify that the parts of the configuration which are only read by the rule can be shared by concurrent applications, the read rule can match simultaneously for two threads attempting to read from the same location, and they can both advance one step concurrently. A similar thing happens for concurrent updates. As long as the threads attempt to update distinct locations, the update rules can match at the same time and the threads can advance concurrently. Moreover, by disallowing rule instances to overlap on the parts they change, the ℕ semantics enforces sequentialization of data-races.

4 Case Study: Parameter Passing Styles in CinK

In this section we exhibit the definitional power of \mathbb{K} , by using it to compactly and naturally define a non-trivial language feature, namely the combination between call-by-value and call-by-reference as mechanisms for binding the formal parameters of a function to the arguments passed during a function call.

For call-by-value, the arguments passed to a function call are first evaluated in the context of the caller, then their values are stored into fresh memory locations, which are then bound to the corresponding formal parameters of the function. The lifetime of these fresh memory locations is limited to the execution of the called function's body, which guarantees that they are not accessible by the caller function after the callee's return.

For call-by-reference, the arguments must evaluate to l-values, and the formal parameters are directly bound to the locations designated by the resulting l-values. Therefore, any updates to the formal parameters during the execution of the function body is reflected onto the arguments passed to the call.

In this section we show how the two mechanisms are being combined in the \mathbb{K} definition of CinK. CinK uses a C++-like notation for the two mechanisms. For instance, the code below declares a function `f` with two parameters `x` and `y`, `x` being called-by-value, while `y` being called-by-reference:

```
int f(int x, int &y) {
  y = ++x;
  return x;
}
```

L-value and R-value Expressions. CinK expressions can be evaluated to either *l-values* or *r-values*. Historically, the names of these two categories come from the fact that an l-value can be used in the left hand side of an assignment, i.e., it can be assigned to, while an r-value corresponds to the right hand side of an assignment (it can be assigned). Semantically, an l-value expression is evaluated to a location, while an r-value expression is evaluated to a value that can be stored into a location. Locations in our \mathbb{K} definition of CinK are modeled by non-negative integers; we write `loc(L)` whenever the value L designates a location. This is achieved with the following syntax declaration:

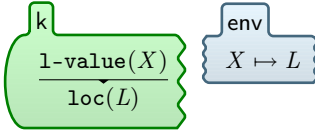
SYNTAX $Val ::= \text{loc}(Int)$

To distinguish expressions which must evaluate to l-values we introduce a special wrapper for them (the other expressions are considered r-values by default):

SYNTAX $Exp ::= \text{l-value}(K)$

The rule evaluating a program variable to an r-value is given on page [39](#). It replaces a variable (at the top of the computation) with the value stored in the location associated to that variable. In contrast, the rule that evaluates a program variable to its l-value replaces the variable with the location associated to it:

RULE MEM-LOOKUP-L-VALUE



Function call. The rule defining the evaluation of a function call expression is described on page 38. It assumes that the arguments have already been evaluated, binds their values to the formal parameters, and schedules the body for execution, while saving the calling context to be restored when returning from the call. This rule is rather plain, similar to languages with a single evaluation strategy; therefore, it does not explain how the actual parameters are evaluated. If the language included only the call-by-value mechanism, then it would be enough to declare the function call expression strict in both arguments. However, since the evaluation strategy for the second argument is depending on the binding specification in the function signature, the function call expression is declared strict only in its first argument:

SYNTAX $Exp ::= Exp (Exps) [strict(1)]$

and a more complex mechanism for evaluating the parameters is required.

Parameter passing styles. To evaluate the arguments of a function call according to the strategy specified by the function parameters, we use ℕ’s special support for evaluation contexts. A context declaration for the function call specifies that the evaluation of arguments needs to consider their declared strategy:

CONTEXT: $(\lambda Xl \bullet Sts)(\frac{\square}{\text{evaluate } \square \text{ following } Xl};)$

This context says not only that the actual parameters must be evaluated when passed to a function value, but also that they need to be evaluated using the **evaluate** construct and **following** the list of formal parameters.

SYNTAX $Exps ::= \text{evaluate } Exps \text{ following } Decls ;$

For a call-by-value formal parameter, the corresponding argument must be evaluated normally, to an r-value:

CONTEXT: $\text{evaluate } \square, - \text{ following int } X, - ;$

For a call-by-reference formal parameter, the corresponding argument must be evaluated as an l-value expression:

CONTEXT: $\text{evaluate } \frac{\square}{\text{l-value } (\square)}, - \text{ following int } \& X, - ;$

This second context uses again the special type of context used above for `evaluate`, by requesting that the expression on position \square be evaluated as an l-value.

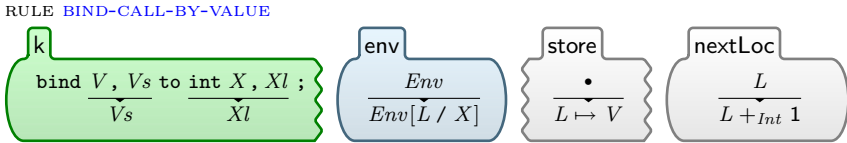
The following two rules, together with the strict evaluation strategy for lists of expressions complete the semantics of `evaluate` by recursing into the lists:

$$\begin{array}{c} \text{RULE} \\ \frac{\text{evaluate } V, El \text{ following } _ , Xl ;}{V, \text{evaluate } El \text{ following } Xl ;} \end{array} \qquad \begin{array}{c} \text{RULE} \\ \frac{\text{evaluate } \bullet \text{ following } \bullet ;}{\bullet} \end{array}$$

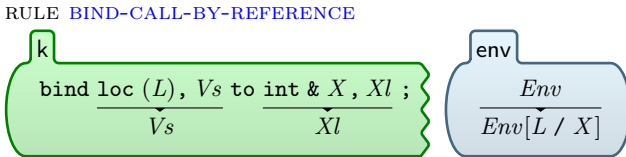
Binding mechanisms. Similarly to the evaluation rules, the binding rules are also different for the two parameter passing styles. The binding is performed using an auxiliary construction:

SYNTAX $K ::= \text{bind } Vals \text{ to } Decls ;$

For call-by-value, the passed value V is stored into a new memory location which is bound to the formal parameter:



For call-by-reference, the location pointed to by the l-value is directly bound to the formal parameter:



Finally, once all parameters have been bound, the binding construct dissolves:

$$\text{RULE} \frac{\text{bind } \bullet \text{ to } \bullet ;}{\bullet}$$

5 On the Semantics of a \mathbb{K} Definition

This section briefly presents the transition semantics of a \mathbb{K} definition. Understanding this semantics is essential for understanding the differences between \mathbb{K} and rewriting logic and thus making correct use of the Maude tools to analyze the behavior of programs against the executable semantics of a language.

As pointed out in the previous section, a \mathbb{K} definition consists of several components: a language syntax (which is a set of *KLabel* constants) possibly annotated with strictness and other attributes and possibly extended with additional

syntactic constructs needed for semantic reasons, an initial configuration, and a set of rewrite rules. As seen, several of \mathbb{K} 's features are in fact just notations, allowing users to define more compact or more modular semantics. For example, the strictness annotations can be desugared in pairs of special heating/cooling rules, the configuration abstraction can complete the cell structure of rules to match that of the initial configurations, and so on. Then a natural question is what is \mathbb{K} , after all, from a theoretical, minimalistic perspective, and what is its semantics. In this section we address this question at an informal level, referring the interested reader to [2,23] for more technical details.

A \mathbb{K} *definition* (or \mathbb{K} *rewrite theory*, or \mathbb{K} *rewrite system*, or even just a \mathbb{K} *system*) is a triple (Σ, S, C) , where Σ is an algebraic signature and where S and C are sets of \mathbb{K} rewrite rules, the former called structural rules and the latter called computational rules. Σ includes operation symbols for all the desired language constructs, builtin data types and values, auxiliary operations needed for the semantics (e.g., `bind_to_`), operations corresponding to cells, operations corresponding to \mathbb{K} -tool-provided data-structures such as lists, sets, maps, etc.

The formal definition of a \mathbb{K} *rule* is rather technical [2,23]. Intuitively, a \mathbb{K} rule consists of a shared *pattern*, which is a multi-context with a distinguished hole for each underlined sub-term in the rule (i.e., sub-term that rewrites), together with two mappings of these special holes: one corresponding to the sub-terms above the line, and another to the sub-terms underneath the line. Any regular rewrite rule is a particular \mathbb{K} rule with an empty pattern (i.e., just a hole). As shown in [2,23], a set R of \mathbb{K} rules yields a *concurrent rewrite relation* \Rightarrow_R on Σ -terms. As expected, \Rightarrow_R can be serialized into sequences of ordinary rewrite steps obtained by turning each \mathbb{K} rule into a regular rewrite rule by forgetting the shared pattern information, that is, by infusing the pattern into both the left-hand-side and the right-hand-side terms. Thus, \Rightarrow_R can and should be regarded as a more concurrent variant of rewriting, one which takes into account the specifics of the \mathbb{K} -rules, namely their capability to share resources.

The split of rules into structural S and computational C in a \mathbb{K} definition (Σ, S, C) is purely methodological; there are no hard requirements on what should be structural and what should be computational. In general, we think of structural rules as rearranging the configuration before or after a computational rule applies. Besides heating/cooling rules and language-specific syntax desugarings, S typically also includes rules telling how the underlying mathematical domains or builtin libraries operate; since an equation can be regarded as two opposite rules, usual algebraic data types can also be captured by means of structural rules, and usual equational deduction can be mimicked with structural rewrites using S . As their name indicates, computational rules are the ones that count as computations. In terms of the generated transition system, the structural rules are not observable while the computational rules are observable.

Formally, given a \mathbb{K} definition (Σ, S, C) , we let \Rightarrow denote the relation $\Rightarrow_S^* \circ \Rightarrow_C \circ \Rightarrow_S^*$. In other words, $\gamma \Rightarrow \gamma'$ if and only if γ can be structurally rearranged into a term which is computationally transformed into a term which can be structurally rearranged into γ' . Or even simpler, γ rewrites to γ' using precisely one

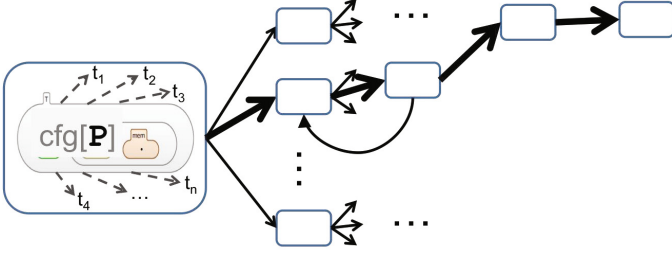


Fig. 3. The \mathbb{K} transition system for the execution of a program P

computational rule. The relation \Rightarrow associates a transition system to any term γ , which we can think of as the behavior of γ under the given \mathbb{K} definition. Consider, for example, the initial configuration of CinK, say $\text{cfg}[PGM]$ (assume IN instantiated to some arbitrary input), and some CinK program P . Then *the \mathbb{K} semantics of P* is the transition system associated to the configuration term $\text{cfg}[P]$, depicted in Figure 3: boxes enclose the structural rearrangements via \Rightarrow_S^* , which appear as dashed arrows in the figure, and full arrows between boxes depict the relation \Rightarrow_C . We can even let $\llbracket P \rrbracket$ denote this transition system.

Therefore, in addition to allowing rules that explicitly specify what can be concurrently shared with other rules, another major difference between \mathbb{K} and rewriting logic is that \mathbb{K} has no equations. Equations can be expressed in \mathbb{K} as two opposite structural \mathbb{K} rules with zero sharing. A question then is how to develop \mathbb{K} tools that can effectively execute and formally analyze \mathbb{K} definitions. Ideally, an implementation would statically analyze the rules in S and C , and make use of efficient decision procedures for common fragments of S (e.g., when it contains rules corresponding to equations such as associativity, commutativity, identity, etc.) and of specialized data-structures and even decision procedures for heating/cooling rules, and so on. Unfortunately, these seem hard. Our current approach in our \mathbb{K} tool prototype is to (automatically) compile \mathbb{K} definitions to Maude, allowing the user to intervene in the process. Specifically, the current version of the \mathbb{K} tool compiles a \mathbb{K} definition (Σ, S, C) into a Maude system module $(\Sigma^{\text{Maude}}, A, E, R)$, following the following rules:

- Each ground Σ -configuration is represented by a ground Σ^{Maude} -term;
- Each structural rule in S is compiled either into an axiom in A (e.g., associativity, commutativity, identity) or into a Maude equation in E ;
- Each computational rule in C is compiled either into a Maude equation in E or into a Maude rewrite rule in R .

The \mathbb{K} tool provides an annotation system by which the user can instruct the tool which computational rules are to be compiled into Maude equations and which into Maude rewrite rules (see Section 6 for examples).

An immediate advantage of compiling \mathbb{K} definitions into Maude rewrite theories is that the \mathbb{K} tool can be used as an interpreter, i.e., given a program P it can execute it according to the semantics of its language; the execution describes a

path from the initial configuration to a normal form (irreducible configuration). Such an execution is intuitively represented by the thick arrows in Figure 3.

6 Executing and Analyzing ℔ Definitions in Maude

In this section we describe how the ℔ tool, taking advantage of the generic Maude tool suite, can be used to execute programs against the ℔ definition of their language and to analyze their behavior.

The following command asks the ℔ tool to compile the definition of CinK into a Maude module; we assume that the command is executed in a directory containing the definition of CinK in the `cink.k` file:

```
$ kcompile cink
$ ls *.maude
cink-compiled.maude
```

6.1 Executing Programs

Consider the following program:

```
int r;                                int main() {
int f(int x) {                          r = 5;
    return (r = x);                      return f(1) + f(2), r;
}                                        }
```

Assuming this program is contained into a file `nondet.cink` in the `programs` directory, its execution can be obtained with the following command:

```
$ krun programs/nondet.cink
<T>
  <k> 1;  </k>
  ...
  <genv> ... r |-> 0 </genv>
  <store> 0 |-> 1 ... </store>
</T>
```

The tool displays the final configuration reached on one of the execution paths, which contains the result of the computation. The information stored in cells is very useful when, e.g., the normal form is a dead-lock configuration. However, `krun` can be also used as an interpreter. For example, if we replace the last line from `main` with “`cout << f(1) + f(2) << r << "\n";`” and execute the program with the `no-config` option, we obtain:

```
$ krun --no-config programs/nondet.cink
3 1
```

The mechanism that connects a cell (here the out cell) to the standard input/output is described in [24].

6.2 Analyzing Executions

The transition system associated to a ℔ definition can be explored using two Maude tools: the search command and the LTL model checker.

Search. `krun` provides the `search` option to explore all execution paths starting from the initial configuration and display the final configurations obtained along these paths. The implementation of this options uses Maude’s search engine. The command below displays all possible outcomes for the `nondet` program above:

```
$ krun nondet.cink --search
Search results:
Solution 1, state 0:
<T>
  <k> 1; </k>
  ...
  <genv>... r |-> 0 </genv>
  <store> 0 |-> 1 ...</store>
</T>
```

Although the behavior of this program is non-deterministic, only one solution is reported. This is a consequence of how the Maude module is generated from the \mathbb{K} definition. To obtain the full behavior of the above program, the tool must be instructed to compile the \mathbb{K} rules that are the source of non-determinism into Maude rewrite rules. Here the non-determinism is given by the evaluation order of the addition operator. We can use the annotation `superheat` for the addition operator to specify that its heating rules must generate Maude rewrite rules:

```
SYNTAX Exp ::= Exp + Exp [superheat strict]
```

Now the above command displays all executions paths:

```
$ krun nondet.cink --search
Search results:
Solution 1, state 1:
<T>
  <k> 1; </k>
  ...
  <genv>... r |-> 0 </genv>
  <store> 0 |-> 1 ...</store>
</T>
Solution 2, state 2:
<T>
  <k> 2; </k>
  ...
  <genv>... r |-> 0 </genv>
  <store> 0 |-> 2 ...</store>
</T>
```

This example shows that members of the same structural class can generate distinct, non-joinable transitions.

The cooling rules could have a similar effect. Using the \mathbb{K} tool to explore the behaviors of program

```
int print(int n) {
    cout << n;
    return n;
}

int main() {
    (print(1) + print(2) + print(3));
    return 0;
}
```

only displays four solutions. The cause is that, by default, cooling rules (e.g., those for the operands of $+$) are only applied when their arguments are reduced to values. However, if we annotate the rule for `return` on page 38 with the tag `supercool`, then the cooling rules will apply eagerly after the application of this rule, cooling the entire computation before attempting to heat it again, and thus allowing all 6 possible solutions to be observed.

The `superheat` and `supercool` tags described above are compiled to Maude so that they offer the ℕ tool user the following intuition: when a `superheat` operation is reached during the execution of the program, an “exhaustive non-determinism” mode is entered; when a `supercool` rule is applied, the next non-deterministic behavior is explored. This way, `superheat/supercool` act as user-defined begin/end brackets within the non-deterministic state-space of the program where exhaustive non-determinism is desired to be explored.

Another way to specify that certain rules should generate transitions into the transition system generated by Maude is to annotate them with the `transition` tag. If the definition of CinK is compiled without any `transition` tags, then the tool will explore only one execution for the following multi-threaded program:

```

int r;
int f(int x) {
    return (r = x);
}

int main() {
    std::thread t1(f, 1);
    std::thread t1(f, 2);
    return r;
}

```

However, if we annotate the rules for memory lookup and memory update with the `transition` tag, the tool will display all 8 possible execution outcomes.

At this stage, the reader may wonder why don't we automatically tag all the operations with `superheat` and all the rules, both structural and computational, with `supercool`, and all the computational rules also with `transition`. While this would indeed guarantee that no behaviors are lost in compilation, in our experience doing so typically yields impractical Maude definitions, whose state-space is too large to search. In general, most of the users of ℕ are interested in fast execution first place, and only then, potentially, in searching. Thus, we decided that the default compilation of the ℕ tool optimizes execution. Searching is considered expert use of the tool. Even experts typically start conservatively, by adding only one or two tags, and then increase their number depending on the complexity of the tested program, too, and only if performance is acceptable. It would be interesting to develop automatic criteria or techniques that provide guarantees of exhaustive behavior exploration with a limited number of tags, but this is beyond our scope here. The ℕ tool currently provides no such criteria.

Model Checking. The ℕ tool also includes a hook to Maude's LTL model-checker, where the latter's sorts and operations are renamed to avoid name clashes and to follow the ℕ tool's convention for naming builtin items. For instance, the sort name for the transition system states is `#ModelCheckerState` and that for the atomic propositions is `#Prop`. For similar reasons, the operators for LTL formulas are prefixed with "LTL". As any builtin sort is subsorted to `K`, `#ModelCheckerState` is also a subsort of `K`.

The Maude module obtained by compiling a \mathbb{K} definition is based on the abstract syntax tree (AST) representation of both the language constructs and the \mathbb{K} constructs. Hence the direct use of Maude to define properties and to call the model-checker for a given initial configuration and a given LTL formula is not quite user-friendly. We created an interface to facilitate the use of the model-checker. We describe the use of this interface by means of a program describing the Dekker's algorithm (see Figure 4). Let us assume that we want to show that this program satisfies the LTL formula

$$\text{LTL}[](\text{eqTo}(\text{critical1}, 1) \text{LTL}\rightarrow \text{eqTo}(\text{critical2}, 0)),$$

where $\text{LTL}[]$ denotes the always modal operator, $\text{LTL}\rightarrow$ the implication, and the atomic proposition $\text{eqTo}(X, I)$ is satisfied by the current configuration if and only if the value of the variable X is equal to I . This property says that for each configuration reachable from the initial one the value of global variable `critical2` is equal to 0 whenever the value of `critical1` is equal to 1, representing half of the mutual exclusion property (the other half is symmetrical).

```

int flag1 = 0,  flag2 = 0;
int critical1 = 0, critical2 = 0;
int turn = 1;
int dekker1() {
  while (true) {
    flag1 = 1; turn = 2;
    while((flag2 == 1) &&
          (turn == 2)) { }
    // Enter critical section
    critical1 = 1;
    // Critical stuff ...
    // Leave critical section
    critical1 = 0;
    flag1 = 0;
  }
}

int dekker2() {
  while (true) {
    flag2 = 1; turn = 1;
    while((flag1 == 1) &&
          (turn == 1)) { }
    // Enter critical section
    critical2 = 1;
    // Critical stuff ...
    // Leave critical section
    critical2 = 0;  flag2 = 0;
  }
}

int main() {
  std::thread t1(dekker1);
  std::thread t2(dekker2);
}

```

Fig. 4. Dekker's algorithm in CinK

We propose the following solution for model-checking this property. The LTL formulas are similar to programming languages: they have syntax and semantics. Therefore we define the syntax and the semantics in separate modules. For this example, the syntax module defines the atomic proposition `eqTo`:

```

MODULE CINK-PROP-SYNTAX
  IMPORTS CINK-SYNTAX
  IMPORTS LTL-HOOKS
  SYNTAX #Prop ::= eqTo(Id, Val)
END MODULE

```

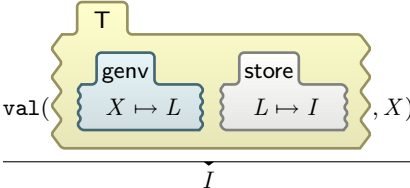

Note the simplicity of this module. Generally, such a module should define the languages for properties to be checked for the defined language. The module LTL-HOOKS provides a ℕ interface to the Maude module defining the syntax of LTL formulas.

The module for semantics has a simple structure, too:

```

MODULE CINK-PROP-SEMANTICS
  IMPORTS MODEL-CHECKER-HOOKS
  IMPORTS CINK-PROP-SYNTAX
  IMPORTS CINK-SEMANTICS
  SYNTAX #ModelCheckerState ::= KItem(Bag)

  SYNTAX Int ::= val(Bag, Id)

  RULE
    
    
$$\frac{\text{val}( \text{genv } X \mapsto L \text{ store } L \mapsto I, X )}{I}$$

  RULE
    
$$\frac{\text{KItem}(B) \text{ LTL} \mid = \# \text{eqTo}(X, I)}{\text{true}}$$

    when  $\text{val}(B, X) ==_{\kappa} I$ 
END MODULE

```

The module MODEL-CHECKER-HOOKS provides a ℕ interface to the Maude module implementing the model-checker algorithm. The sort for configurations is Bag and therefore it is injected as a subsort of #ModelCheckerState. The auxiliary function val (C, X) returns the value of the variable X in the configuration C. Note that its definition is given by just one rule. The last rule in the module gives the LTL semantics to the atomic proposition eqTo.

The definition of CinK together with these new modules is compiled, and then the krun command with the --check option is executed:

```

$ krun dekker.cink --check LTL[] (eqTo(critical1, 1) LTL->
                                eqTo(critical2, 0))

```

Remark 1. The full implementation of this command is in progress. For instance, when the formula is false the counter-example is huge. Currently, the output obtained from the corresponding Maude model checking command is displayed unformatted—we are working on finding a nicer way to represent it.

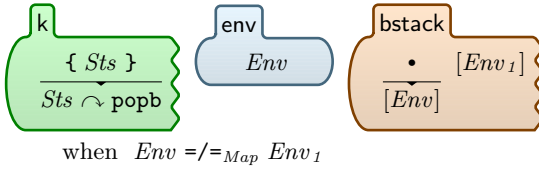
The above command works fine if the following two conditions are fulfilled:

1. the definition includes enough annotations to generate a transition system representing a faithful abstraction of the intended ℕ semantics;

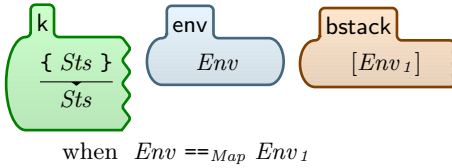
2. the set of configurations reachable from the initial configuration is finite.

Unfortunately, the second condition is not satisfied by the initial configuration of the program describing the Dekker’s algorithm. Since in CinK we may have variable declarations inside of blocks, each time the execution enters a block, the environment is saved in the cell `bstack`. Hence the two infinite `while` loops will infinitely increase the size of this cell during the exploration process. A small change of this rule re-establishes the needed property. The `while` loop does not include declarations of variables, so saving the environment is useless. We modify the semantics of the block statement such that the environment is saved only if it differs from the one stored in the top of the cell:

RULE BLOCK



RULE BLOCK



Remark 2. The search command and the model checker must be carefully used since, as we already mentioned, the Maude modules produced by the \mathbb{K} tool are not always faithfully representing the intended \mathbb{K} transition system. The main motivation for this choice is given by efficiency. The user can use the annotations (tags) to guide the compilation process into obtaining good abstraction of the \mathbb{K} transition system. However, even if the Maude transition system is a good abstraction of the \mathbb{K} one, it often is a (strict) subsystem of that giving the transition semantics to the original \mathbb{K} definition.

7 Conclusions

This paper gave a high-level overview of the \mathbb{K} framework: its motivation and objective, what it is and how it works, and its relationship to rewriting logic and Maude. The \mathbb{K} framework and the \mathbb{K} tool have by now reached maturity, and are currently being actively used for defining real programming languages and experimenting with various language features. Besides didactic and prototypical languages (such as lambda calculus, System F, and Agents), the \mathbb{K} tool was used to formalize C [25] (and to analyze C programs [26]) and Scheme [27]; additionally, definitions of Haskell, Javascript, X10, a framework for domain specific languages [28,29] or P-Systems [30], a RISC assembly language [31], and

LLVM are underway. With respect to analysis tools, the ℔ tool was used for tools like type checkers and type inferencers [32], and in the development of a new deductive program verification tool using program assertions based on matching logic [33,34], model checking tools [35,36], symbolic execution [37,38], computing worst case execution times [39], or researching runtime verification techniques [40,23]. All these definitions and analysis tools can be found on the ℔ framework website at <http://k-framework.org>

References

1. Roşu, G.: CS322, Fall 2003 - Programming Language Design: Lecture Notes. Technical Report UIUCDCS-R-2003-2897, University of Illinois at Urbana Champaign. Lecture notes of a course taught at UIUC (December 2003)
2. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. *Journal of Logic and Algebraic Programming* 79(6), 397–434 (2010)
3. Serbanuta, T.F., Arusoae, A., Lazar, D., Ellison, C., Lucanu, D., Rosu, G.: The K primer (version 2.5). In: Hills, M. (ed.) *Proceedings of the Second International K Workshop, K 2011*. *Electronic Notes in Theoretical Computer Science* (to appear, 2012)
4. 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)
5. Meseguer, J., Roşu, G.: Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools. In: Basin, D., Rusinowitch, M. (eds.) *IJCAR 2004*. LNCS (LNAI), vol. 3097, pp. 1–44. Springer, Heidelberg (2004)
6. Meseguer, J., Roşu, G.: The rewriting logic semantics project. *The rewriting logic semantics project* 373(3), 213–237 (2007)
7. Meseguer, J., Roşu, G.: The Rewriting Logic Semantics Project: A Progress Report. In: Owe, O., Steffen, M., Telle, J.A. (eds.) *FCT 2011*. LNCS, vol. 6914, pp. 1–37. Springer, Heidelberg (2011)
8. Lucanu, D., Şerbănuţă, T.F.: Cink - an exercise of thinking in K. Technical Report TR12-03, Department of Computer Science, Alexandru Ioan Cuza University of Iaşi (2012), [http://thor.info.uaic.ro/\\$\sim\\$str/tr.pl.cgi](http://thor.info.uaic.ro/\simstr/tr.pl.cgi)
9. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science* 96(1), 73–155 (1992)
10. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. *Theoretical Computer Science* 403(2-3), 239–264 (2008)
11. Farzan, A., Chen, F., Meseguer, J., Roşu, G.: Formal Analysis of Java Programs in JavaFAN. In: Alur, R., Peled, D.A. (eds.) *CAV 2004*. LNCS, vol. 3114, pp. 501–505. Springer, Heidelberg (2004)
12. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. *STTT* 2(4), 366–381 (2000)
13. Meseguer, J.: Rewriting Logic as a Semantic Framework for Concurrency: a Progress Report. In: Montanari, U., Sassone, V. (eds.) *CONCUR 1996*. LNCS, vol. 1119, pp. 331–372. Springer, Heidelberg (1996)
14. Verdejo, A., Martí-Oliet, N.: Implementing CCS in Maude 2. In: Gadducci, F., Montanari, U. (eds.) *Proceedings of the Forth International Workshop on Rewriting Logic and its Applications (WRLA 2002)*. *Electronic Notes in Theoretical Computer Science*, vol. 71. Elsevier (2002)

15. Şerbănuţă, T.F., Roşu, G., Meseguer, J.: A rewriting logic approach to operational semantics. *Information and Computation* 207, 305–340 (2009)
16. Kahn, G.: Natural Semantics. In: Brandenburg, F.J., Vidal-Naquet, G., Wirsing, M. (eds.) *STACS 1987*. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987)
17. Plotkin, G.D.: A structural approach to operational semantics. *Journal of Logic and Algebraic Programming* 60-61, 17–139 (2004); Original version: University of Aarhus Technical Report DAIMI FN-19 (1981)
18. Mosses, P.D.: Modular structural operational semantics. *Journal of Logic and Algebraic Programming* 60-61, 195–228 (2004)
19. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. *Information and Computation* 115(1), 38–94 (1994)
20. Friedman, D.P., Wand, M., Haynes, C.T.: *Essentials of Programming Languages*, 2nd edn. MIT Press, Cambridge (2001)
21. Berry, G., Boudol, G.: The chemical abstract machine. *Theoretical Computer Science* 96(1), 217–248 (1992)
22. Danvy, O., Nielsen, L.R.: Refocusing in reduction semantics. RS RS-04-26, BRICS, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark (November 2004); This report supersedes BRICS report RS-02-04. A preliminary version appears in the *Informal Proceedings of the Second International Workshop on Rule-Based Programming, RULE 2001*. *Electronic Notes in Theoretical Computer Science*, vol. 59.4
23. Şerbănuţă, T.F.: *A Rewriting Approach to Concurrent Programming Language Design and Semantics*. PhD thesis, University of Illinois at Urbana-Champaign (December 2010), <https://www.ideals.illinois.edu/handle/2142/18252>
24. Arusoaie, A., Şerbănuţă, T.F., Ellison, C., Roşu, G.: Making Maude Definitions More Interactive. In: Durán, F. (ed.) *WRLA 2012*. LNCS, vol. 7571, pp. 83–98. Springer, Heidelberg (2012)
25. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL 2012)*, pp. 533–544. ACM (2012)
26. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: *33rd Conference on Programming Language Design and Implementation (PLDI 2012)*. ACM (to appear, 2012)
27. Meredith, P., Hills, M., Roşu, G.: An executable rewriting logic semantics of k-scheme. In: Dube, D. (ed.) *Proceedings of the 2007 Workshop on Scheme and Functional Programming, SCHEME 2007*, Technical Report DIUL-RT-0701, Laval University, pp. 91–103 (2007)
28. Rusu, V., Lucanu, D.: A K-Based Formal Framework for Domain-Specific Modelling Languages. In: Beckert, B., Damiani, F., Gurov, D. (eds.) *FoVeOOS 2011*. LNCS, vol. 7421, pp. 214–231. Springer, Heidelberg (2012)
29. Rusu, V., Lucanu, D.: K semantics for OCL—a proposal for a formal definition for OCL. In: Hills, M. (ed.) *Proceedings of the Second International K Workshop, K 2011*. *Electronic Notes in Theoretical Computer Science* (to appear, 2012)
30. Şerbănuţă, T., Ştefănescu, G., Roşu, G.: Defining and Executing P Systems with Structured Data in K. In: Corne, D.W., Frisco, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) *WMC 2008*. LNCS, vol. 5391, pp. 374–393. Springer, Heidelberg (2009)
31. Asavaoe, M.: K semantics for assembly languages: A case study. In: Hills, M. (ed.) *Proceedings of the Second International K Workshop, K 2011*. *Electronic Notes in Theoretical Computer Science* (to appear, 2012)

32. Ellison, C., Şerbănuță, T.F., Roşu, G.: A Rewriting Logic Approach to Type Inference. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 135–151. Springer, Heidelberg (2009)
33. Roşu, G., Ellison, C., Schulte, W.: Matching Logic: An Alternative to Hoare/Floyd Logic. In: Johnson, M., Pavlovic, D. (eds.) AMAST 2010. LNCS, vol. 6486, pp. 142–162. Springer, Heidelberg (2011)
34. Roşu, G., Ştefănescu, A.: Matching logic: A new program verification approach (NIER track). In: 30th International Conference on Software Engineering (ICSE 2011), pp. 868–871 (2011)
35. Asăvoae, I.M., Asăvoae, M.: Collecting Semantics under Predicate Abstraction in the K Framework. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 123–139. Springer, Heidelberg (2010)
36. Asavoaie, I.M., de Boer, F., Bonsangue, M.M., Lucanu, D., Rot, J.: Bounded model checking of recursive programs with pointers in k. In: WRLA (2012); as extended abstract in the ETAPS preproceedings
37. Asavoaie, I.M., Asavoaie, M., Lucanu, D.: Path directed symbolic execution in the k framework. In: Ida, T., Negru, V., Jelebean, T., Petcu, D., Watt, S.M., Zaharie, D. (eds.) SYNASC, pp. 133–141. IEEE Computer Society (2010)
38. Asavoaie, I.M.: Abstract semantics for alias analysis in K. In: Hills, M. (ed.) K 2011. Electronic Notes in Theoretical Computer Science (to appear, 2012)
39. Asăvoae, M., Lucanu, D., Roşu, G.: Towards semantics-based WCET analysis. In: WCET (2011)
40. Roşu, G., Schulte, W., Şerbănuță, T.F.: Runtime Verification of C Memory Safety. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 132–151. Springer, Heidelberg (2009)

Design and Analysis of Cloud-Based Architectures with KLAIM and Maude^{*}

Martin Wirsing^{1,4}, Jonas Eckhardt², Tobias Mühlbauer², and José Meseguer³

¹ LMU Munich

² Technical University of Munich

³ University of Illinois at Urbana-Champaign

⁴ IMDEA Software

Abstract. Cloud computing is a modern paradigm for offering and utilizing distributed infrastructure resources in a dynamic way. Cloud-based systems are safety- and security-critical; they need to satisfy time-critical performance-based quality of service properties and to dynamically adapt to changes in the potentially hostile and uncertain environment they operate in. In this paper we propose the coordination language KLAIM and a composite actor approach for modelling Cloud-based architectures whereas for formally analyzing such architectures we use a rewriting-based approach. We specify the operational semantics of KLAIM in Maude, show how to realize KLAIM programs in a distributed implementation of Maude, and simulate and analyze three simple Cloud architectures with Maude and the Maude LTL model checker. Moreover, we report shortly on the Maude specification and analysis of three larger Cloud case studies using the composite actor model, where statistical model checking with the Maude-based tool PVeStA is successfully used for detecting bugs and performance issues and for analyzing a defense mechanism against distributed denial-of-service attacks.

Keywords: rewriting logic, distributed systems, cloud computing, formal analysis, coordination languages, composite actor model.

1 Introduction

In 1961 at the MIT Centennial, John McCarthy predicted that "Computing may someday be organized as a public utility just as the telephone system is a public utility [...] The computer utility could become the basis of a new and important industry" [13]. Fifty years later this dream is becoming a reality. Cloud Computing provides global access to data, software services and infrastructure through the internet, typically utilizing a pay for use model. Main characteristics of Cloud Computing are virtualization, enabling elasticity and the illusion of infinite capacity, multiple customers consuming the same software service, and service-level agreements ensuring concrete levels of quality of service. Examples for Cloud services are mail, calendar, credit card services or any kind of data storage. According to Gartner [19], by 2012, 20 percent of businesses will have

^{*} This work has been partially sponsored by the EU-funded projects FP7-257414 ASCENS and FP7-256980 NESSoS, and AFOSR Grant FA8750-11-2-0084.

no ownership of IT assets; similarly, US federal agencies are told “to default to Cloud-based solutions whenever a secure, reliable, cost-effective Cloud option exists” [30].

However, for current Cloud solutions security and reliability are still major concerns [26]. E.g., the open foundation project Cloutage (see cloutage.org) enumerates more than 60 Cloud incidents in 2011; the Berkeley view on Cloud Computing lists ten main obstacles for the growth and adoption of Cloud Computing, among them: availability of service, data transfer bottlenecks, performance unpredictability, and bugs in large distributed systems. As thus, Cloud Computing-based systems (i) are safety- and security-critical systems which have strong qualitative and quantitative formal requirements, (ii) have equally important time-critical performance-based quality of service properties (e.g., availability), and (iii) need to dynamically adapt to changes in the potentially hostile (e.g., distributed denial of service attacks) and often probabilistic environment they operate in.

To tackle these challenges, we propose in this work [1] the coordination language KLAIM and a composite actor approach for modelling Cloud-based architectures and a rewriting-based approach [21] for formally analyzing such architectures. KLAIM [8] is a process algebra-based formalism for designing distributed applications. It supports explicit localities and multiple tuple spaces and permits exchanging data and processes and retrieving information over the net. As examples for the design of Cloud-based architectures in KLAIM we study a simple (Fibonacci-) server consumer application, a load balancing algorithm, and a mutual exclusion algorithm. For analyzing KLAIM specifications we specify KLAIM in object-oriented Maude and show how to realize distributed KLAIM programs by providing multiple instances of Maude which communicate via sockets. We use the Maude system and the Maude LTL model checker for simulating the Cloud Computing examples and for verifying appropriate safety properties. These examples show that it is possible to formally specify and analyze Cloud-based architectures based on KLAIM and Maude, but also that we face a state space explosion which makes it hard to deal with more complex scenarios.

To tackle this scalability issue we propose a composite actor model as an alternative approach. An actor is a concurrent object that encapsulates a state and can be addressed using a unique name. Actors communicate with each other using asynchronous messages. The composite actor model reflects the so-called “Russian Dolls” model [23] and supports an arbitrary hierarchical composition of entities. This approach is well-suited for statistical model checking with the Maude-based tool PVeStA and can be used to overcome some of the above-mentioned obstacles to the growth of Cloud Computing. To illustrate this we report on three case studies: (i) the detection of bugs in the design of a key distribution mechanism based on the Zookeeper distributed coordination, (ii) performance prediction of a distributed broker-based a publish/subscribe service for stock exchange events, and (iii) improving the availability of services using a Cloud-based denial-of-service prevention mechanism.

¹ A full description of the technical results can be found in [9,27].

Outline. The paper is structured as follows: Sect. 2 shortly presents the KLAIM language and its structured operational semantics. In Sect. 3 we present three Maude-based formal executable specifications of KLAIM: M-KLAIM, which maps the SOS style semantics to a rewriting logic-based semantics; OO-KLAIM, which slightly modifies the semantics of KLAIM by defining inter-node communication as asynchronous message passing; and finally D-KLAIM, which extends OO-KLAIM and allows specifications to be executed in a distributed environment. Sect. 4 shows how the Maude tools can be used for formally analyzing KLAIM designs of three simple Cloud-based systems; in particular, we show how to model check distributed D-KLAIM specifications by using an appropriate socket abstraction. In Sect. 5 we shortly present the composite actor approach as an alternative for specifying Cloud-based architectures and use statistical model checking for formally analyzing the three case studies mentioned above. We conclude by summarizing our results and discussing the scalability of formal analysis.

2 The KLAIM Coordination Language

KLAIM (**K**ernel **L**anguage for **A**gents **I**nteraction and **M**obility) [8] is a kernel programming language for mobile computing. The language offers the aspects of a computation as well as a coordination language. The language’s basic operators were influenced by process algebras like CSP, CCS [24] and the π -calculus [25]. Additionally, Linda’s tuple space primitives [14,5] provide a coordination mechanism. These primitives are enriched with explicit localities which allow to distinguish between multiple computing sites and the distribution of the tuple spaces across such sites. A locality can be either a *physical* or a *logical* locality. This separation allows a program to be defined independently from the underlying network’s physical setup. Again, the network structure, the mapping between logical and physical localities, and the distribution of processes, can be rearranged without any changes to the program. The specification of KLAIM also includes a type system that statically checks security properties, i.e., whether the intended operations of a process comply with its access rights. In the following, we give a brief overview of the syntactic elements as well as the operational semantics of KLAIM. For an in-depth description of KLAIM and its syntax and semantics we refer to [8].

2.1 Syntactic Elements of KLAIM

At the highest level of abstraction, the KLAIM model specifies a soup of nodes, called a *net*. A net can be either a single *node* or a composition of nets N_1 and N_2 , $N_1 \parallel N_2$. A *node* is a triple $s ::_{\rho} P$ where s is a *site*, P is a *process*, and ρ defines an *allocation environment*. A *site* can be thought of as a globally valid identifier for a node. Logical localities, i.e., symbolic names for a site, allow programs to reference nodes while ignoring the precise allocation between these names and actual sites.

² For the sake of brevity, we omit the type system in the following Maude-based specifications of KLAIM.

The distinguished logical locality *self* refers to the current execution site. These localities are considered to be first-order data which can be created dynamically and shared using the tuple space. Each node has a specific *allocation environment*, which is a (partial) function from logical localities to sites. $[s/l]$ denotes the environment that maps the logical locality l to the site s . $\rho_1 \bullet \rho_2$ denotes the allocation environment that combines the environments ρ_1 and ρ_2 and is defined by:

$$\rho_1 \bullet \rho_2(l) = \begin{cases} \rho_1(l) & \text{if } \rho_1(l) \text{ is defined} \\ \rho_2(l) & \text{otherwise} \end{cases}$$

In KLAIM, sites are also considered to be logical localities for which the allocation environment acts as the identity function. Additionally, it is assumed that for an allocation environment ρ_s at site s the equation $\rho_s(\text{self}) = s$ holds. Nodes that fulfil this property are said to be well-formed. A KLAIM net is said to be *legal* if each node is well-formed and is assigned a distinct site in the net.

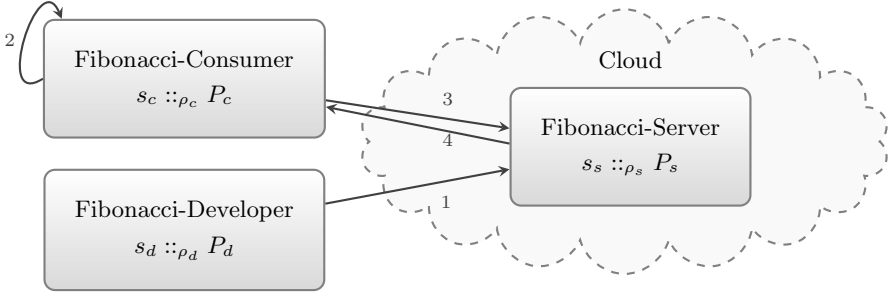
KLAIM processes are built using operators borrowed from Milner's CCS [24]. Additionally, a process can be a process variable or a process invocation $A\langle \vec{P}, \vec{l}, \vec{e} \rangle$, where \vec{P} is a sequence of processes, \vec{l} a sequence of localities, and \vec{e} a sequence of expressions. KLAIM assumes that a process identifier A has a unique defining equation $A(\vec{X}, \vec{u}, \vec{x}) =_{\text{def}} P$, with \vec{X} a sequence of process variables, \vec{u} a sequence of locality variables, \vec{x} a sequence of expression variables, and P being a process. The KLAIM actions $\text{out}(t)@l$, $\text{eval}(P)@l$, $\text{read}(t)@l$, and $\text{in}(t)@l$ correspond to the Linda operations to generate tuples (**out**), spawn processes (**eval**), read tuples (**rd**), and consume tuples (**in**). In KLAIM, the operations have logical localities as a postfix, which denote the sites the actions address. t stands for a tuple, which is a list of expressions, processes, localities (including locality variables) and formal fields. Formal fields are of the form $!v$, where v is either an expression variable, a process variable, or a locality variable. In addition to the operations borrowed from Linda, the *newloc*(u) action is used to create fresh sites. The locality variable u refers to that fresh site in the prefixed process.

Example 1 (A Cloud-based architecture specification based on KLAIM). In the following, we consider the example: a service developer builds a Fibonacci service. The service should provide high scalability and availability and is started in the Cloud. A consumer calls the service in the Cloud and stores the incoming Fibonacci numbers in a local storage.

In KLAIM, each participating entity, the developer, the consumer, and the server in the Cloud can be modeled as KLAIM nodes and their behavior as KLAIM processes. The KLAIM net

$$s_c ::_{\rho_c} P_c \parallel s_d ::_{\rho_d} P_d \parallel s_s ::_{\rho_s} P_s$$

specifies the example setup, where s_c, s_d, s_s are the sites, ρ_c, ρ_d, ρ_s are the allocation environments, and P_c, P_d, P_s are the processes of the consumer, the



- 1: Start Fibonacci service on a server in the Cloud
- 2: Create storage node
- 3: Request Fibonacci number
- 4: Send Fibonacci number

Fig. 1. Overview of the Fibonacci Cloud service architecture

developer, and the server of the Fibonacci service, respectively. For example, the process of the Fibonacci server, P_s , can be specified as

$$in(start)@self.out(0)@c.out(0, 1)@self.FibRec(\emptyset, \emptyset, \emptyset)$$

where $start$ is the tuple that is sent from the developer to start the service, $\rho_s(c) = s_c$ and $FibRec$ is defined by

$$FibRec(\emptyset, \emptyset, \emptyset) =_{\text{def}} in(f_1, f_2)@self.out(f_1 + f_2)@c.out(f_2, f_1 + f_2)@self.FibRec(\emptyset, \emptyset, \emptyset)$$

The $FibRec$ process definition blocks until it receives two previous Fibonacci numbers, calculates and sends the new Fibonacci number to the consumer, and finally sends the two last Fibonacci numbers to itself and invokes $FibRec$ again.

Figure 1 illustrates the example. For the complete specification we refer to [9, 27].

2.2 KLAIM's Operational Semantics

KLAIM's operational semantics is given in the structural operational semantics (SOS) style and differentiates between two semantics: the *symbolic semantics* and the *reduction relation*. The semantics proceeds in two steps. First, the *symbolic semantics* specifies the effects of actions on the tuple space which, in KLAIM, is reflected at the process level and defines the process commitments related to localities and the allocation environment. In a second step, the *reduction relation* fully describes the process behavior in a net.

The structural rules of the the *symbolic semantics* specify the possible transitions of KLAIM processes. The resulting labeled transition system does not

take the physical location of processes and the tuple space into account. In the transition system, the labeled transition

$$P \xrightarrow[\rho]{\mu} P'$$

describes how process P evolves to process P' . The label μ gives an abstract description of what activity is performed and the label ρ stands for the allocation environment that records the local bindings that must be taken into account to evaluate μ . For example, the rules to send and consume a tuple

$$out(t)@l.P \xrightarrow[\phi]{s(t)@l} P$$

$$in(t)@l.P \xrightarrow[\phi]{i(t)@l} P$$

specify that a process P with the prefix $out(t)@l$ or $in(t)@l$ is able to evaluate to process P with the side effect of sending the tuple t to l ($\mu = s(t)@l$) or consuming the tuple t from l ($\mu = i(t)@l$). Both rules also state that the allocation environment does not have to be taken into account to evaluate the activities, i.e., the empty allocation environment has to be taken into account ($\rho = \phi$).

KLAIM reflects the tuple space at the process level, where tuples are modeled as processes. The auxiliary process $out(et)$, whose symbolic semantics is given by the structural rule

$$out(et) \xrightarrow[\phi]{o(et)@self} nil$$

denotes the presence of the evaluated tuple et in the tuple space. Tuples are evaluated using the tuple evaluation function $\mathcal{T}[\cdot, \cdot]_{\rho}$, which exploits the allocation environment to resolve locality names. The evaluation of a process P , $\mathcal{T}[P]_{\rho}$ introduces the concept of the process closure $P\{\rho\}$, which combines the process P with the allocation environment ρ .

Nets are identified up to the smallest congruence such that the net composition \parallel is associative and commutative. The *reduction relation* describes the process behavior in a net and provides rules for actions that affect the local node and rules for actions that affect a remote node. Syntactically, the reduction transition

$$N \rightsquigarrow N'$$

describes the evolution of net N to N' . The local and remote rules for the out operation

$$(1) \frac{P \xrightarrow[\rho']{s(t)@l} P' \quad s = \rho' \bullet \rho(l) \quad et = \mathcal{T}[t]_{\rho' \bullet \rho}}{s ::_{\rho} P \rightsquigarrow s ::_{\rho} P' \mid out(et)}$$

$$(2) \frac{P_1 \xrightarrow[\rho]{s(t)@l} P'_1 \quad s_2 = \rho \bullet \rho_1(l) \quad et = \mathcal{T}[t]_{\rho \bullet \rho_1}}{s_1 ::_{\rho_1} P_1 \parallel s_2 ::_{\rho_2} P_2 \rightsquigarrow s_1 ::_{\rho_1} P'_1 \parallel s_2 ::_{\rho_2} P_2 \mid out(et)}$$

add a new auxiliary process to the local (rule (1)) or to a remote (rule (2)) process and thereby put a new tuple into the tuple space. In rule (2), the tuple t is evaluated using the allocation environment $\rho \bullet \rho_1$, which means that if the process has a closure $P\{\rho\}$, its closure is used in conjunction with the local allocation environment ρ_1 to evaluate the tuple. If the process has no closure, the equation $\rho = \phi$ holds, and the tuple is evaluated using only the local allocation environment $\rho_1 = \phi \bullet \rho_1$. Finally, if a tuple is sent to a remote node, the sending process' closure and the sending node's allocation environment are used to evaluate the tuple.

Pattern matching is used to identify appropriate tuples for an *in* or *read* operation. For example, the rule to consume a tuple from a remote node

$$(6) \frac{P_1 \xrightarrow[\rho]{i(t)@l} P'_1 \quad s_2 = \rho \bullet \rho_1(l) \quad P_2 \xrightarrow[\phi]{o(et)@self} P'_2 \quad \text{match}(\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}, et)}{s_1 ::_{\rho_1} P_1 \parallel s_2 ::_{\rho_2} P_2 \rightsquigarrow s_1 ::_{\rho_1} P'_1[et/\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}] \parallel s_2 ::_{\rho_2} P'_2}$$

uses pattern matching to match the remotely available evaluated tuple with the evaluation of the tuple that is the argument of the *in* operation.

Example 2 (Communication between nodes in KLAIM). Let us conclude this overview of the KLAIM language with an example that shows how nodes communicate in KLAIM: a net consists of three nodes that are located at the sites s_1, s_2 and s_3 . The nodes are all well-formed, since their allocation environments ρ_i , $i \in \{1, 2, 3\}$ are well-defined ($\rho_i(self) = s_i$). Furthermore, the logical locality l_2 is mapped to s_2 in ρ_1 and ρ_3 . An *out* operation at site s_1 first triggers a transition as described in the *symbolic semantics*:

$$\begin{array}{c} s_1 ::_{\rho_1} \text{out}(7)@l_2.nil \quad \parallel \quad s_2 ::_{\rho_2} nil \quad \parallel \quad s_3 ::_{\rho_3} \text{in}(!x)@l_2.P \\ \downarrow s(7)@l_2 \\ nil \end{array}$$

Now the corresponding rule of the reduction relation adds the evaluated tuple 7 to the process at s_2 , which is the site that the logical locality l_2 maps to in the allocation environment at site s_1 . Simultaneously, the *symbolic semantics* allows for transitions to be made by the action at site s_3 and the auxiliary process at site s_2 :

$$\begin{array}{c} s_1 ::_{\rho_1} nil \quad \parallel \quad s_2 ::_{\rho_2} \text{out}(7).nil \quad \parallel \quad s_3 ::_{\rho_3} \text{in}(!x)@l_2.P \\ \downarrow o(7)@self \quad \quad \quad \downarrow i(!x)@l_2 \\ nil \quad \quad \quad P \end{array}$$

In a last step, the rule for a remote consumption of a tuple allows the tuple 7 of site s_2 to be consumed by the *in* operation at site s_3 since the expression variable $!x$ matches with any value:

$$s_1 ::_{\rho_1} nil \quad \parallel \quad s_2 ::_{\rho_2} nil \quad \parallel \quad s_3 ::_{\rho_3} P[7/!x]$$

3 Maude-Based Implementations of KLAIM (*-KLAIM)

Rewriting logic [21] supports the executable specification of KLAIM's syntax and structural operational semantics. This can be done in several definitional styles [29], which can exactly mirror any desired SOS style. In this work, we aim at an efficient executable specification of KLAIM and therefore do *not* follow the SOS style of KLAIM's semantics *au pied de la lettre*. That is, the inference rules of the structural operational semantics are not specified as given, but are transformed to rewrite rules that allow for better executability. In the following, we present three Maude-based formal executable specifications of KLAIM: M-KLAIM, which maps the SOS style semantics to a rewriting logic-based semantics in a straightforward fashion; OO-KLAIM, which slightly modifies the semantics of KLAIM by defining inter-node communication as asynchronous message passing; and finally D-KLAIM, which extends OO-KLAIM and allows specifications to be executed in a distributed environment.

3.1 M-KLAIM

In terms of syntax, the Maude-based specification KLAIM, M-KLAIM, was designed to be as close as possible to KLAIM's notation. For example, the KLAIM net

$$s_1 ::_{[s_1/\text{self}] \bullet [s_2/l_2]} \text{out}(1)@l_2.\text{nil} \parallel s_2 ::_{[s_2/\text{self}]} \text{in}(1)@\text{self}.\text{nil}$$

is represented as the following Maude term:

$$\begin{aligned} &(\text{site}'1)::\{[\text{site}'1]/\text{self}] * [\text{site}'2)'/12]\} \text{out}(1)@'12.\text{nil} \parallel \\ &(\text{site}'2)::\{[\text{site}'2)/\text{self}]\} \text{in}(1)@\text{self}.\text{nil} \end{aligned}$$

For the sake of brevity we do not show an additional counter that is part of the M-KLAIM syntax in the examples in this work. This counter represents the number of child nodes that the node created and is used to process `newloc` operations. A full description of the syntactic elements of the M-KLAIM specification can be found in [27,9].

Works by Braga and Meseguer [4], Verdejo et al. [34], and Serbanuta et al. [29] have shown that SOS rules can naturally be mapped to rewrite rules with different styles. Basically, inference rules of the form

$$\frac{P_1 \rightarrow Q_1 \dots P_n \rightarrow Q_n}{P_0 \rightarrow Q_0}$$

become conditional rewrite rules of the form

$$P_0 \rightarrow Q_0 \text{ if } P_1 \rightarrow Q_1 \wedge \dots \wedge P_n \rightarrow Q_n$$

where the condition may include rewrite conditions. Some technical details may be added to capture the one-step semantics of some SOS rules. In our approach defining the rewrite semantics of KLAIM, $\mathcal{R}_{\text{KLAIM}}$, we combine the rules of the symbolic semantics and the reduction relation. Transitions in $\mathcal{R}_{\text{KLAIM}}$ only happen at the net level. The structural rules for action prefixes are not reflected by

the rewriting semantics. Inference rules of the reduction relation with premises that require a process to perform a transition according to the symbolic semantics are mapped to inference rules with no such condition. The symbolic semantics transition for the action prefix is built-in into the rewrite rule. Using this approach, the conditional rewrite rules obtained by the transformation of the reduction relation's rules include no rewrite conditions as the remaining premises of these inference rules are expressed by equational and matching conditions. One advantage of this approach is that the specification can be executed with higher performance, because equational and matching conditions are evaluated faster than rewrite conditions and therefore avoid expensive non-deterministic rewrite searches in conditions. As examples, we show the corresponding rewriting logic rules for the local and remote rules for the *out* operation (rule (1) and (2)) and the *in* operation (rule (6)). In these Maude rules the variables *RHO*, *RHO1*, and *RHO2* represent allocation environments; *T* a tuple; *S*, *S1*, and *S2* sites; *L* a locality; and finally *SP* and *PP* processes.

The rules

```

cr1 [out-self] :
  (S::{RHO} (out(T) @ L) . SP | PP )
=>
  (S::{RHO} SP | PP | out(T[| T |]RHO) )
if S := RHO(L) .

```

and

```

cr1 [out-remote] :
  (S1::{RHO1} (out(T) @ L) . SP | PP) || (S2::{RHO2} P )
=>
  (S1::{RHO1} SP | PP) || (S2::{RHO2} P | out(T[| T |]RHO1) )
if S2 := RHO1(L) .

```

correspond to rule (1) and rule (2) of KLAIM's reduction relation. *out-self* specifies that if the process of the node at site *S* contains a process which can perform an *out* action and the action's locality evaluates to *S*, then a new auxiliary process is added to the process at that node. Otherwise, if the action's locality evaluates to a different site than the node's site and the remote node at that site exists, the rule *out-remote* adds the auxiliary process to the process at that node. In both rules, the process *PP* reflects the possible existence of a parallel process to the process that is prefixed with the *out* action.

The rule

```

cr1 [in-remote] :
  (S1::{RHO1} (in(T) @ L) .SP | PP) || (S2::{RHO2} P | out(ET1) )
=>
  (S1::{RHO1} (SP [ ET1 / ET2 ]) | PP) || (S2::{RHO2} P )
if ET2 := T[| T |]RHO1 /\ S2 := RHO1(L) /\ match(ET1, ET2) .

```

corresponds to rule (6) of KLAIM's reduction relation. *in-remote* specifies that if the process of the node at site *S1* contains a process which can perform an *in* action and the action's locality evaluates to *S2* and the node at site *S2* contains an

auxiliary process which matches with the requested tuple, then, the evaluated tuple ET_1 is replaced by the evaluated tuple ET_2 and the auxiliary process is removed.

The complete set of rules of the specification of the semantics of M-KLAIM can be found in [27,9].

3.2 OO-KLAIM

Maude supports modeling of distributed object-based systems in which objects communicate via message passing. In the following, we extend M-KLAIM by adding the object-oriented paradigm to the specification. This extensions allows a more natural specification of Cloud-based architectures.

The predefined Maude module *CONFIGURATION* supports modelling of object-based systems in Core Maude. Terms of sort **Configuration** consist of a multitude of objects and messages, which can be thought of as a soup. More specifically, a configuration is a multiset of objects (defined by the sort **Object**) and messages (defined by the sort **Msg**) that describe a possible system state. To address objects, an object's first argument is usually an object identifier that is unique in the system. Object identifiers are terms of the sort **Oid**. Messages usually contain such an identifier to address specific objects. An object's state is described by terms of the sort **Attribute**. These attributes are usually, as a set of attributes (defined by the sort **AttributeSet**), part of an object and determine the object's current state.

In OO-KLAIM, we decide to introduce a slightly modified syntax for objects and configurations. To better demonstrate that KLAIM nodes describe the entities of a distributed system, we extend the notion of a site to be an IPv4 address together with a port.

The OO-KLAIM specification uses messages for the communication between nodes. These messages are syntactically reflected by terms of the sort **Msg**, which are made up of an object identifier which represents the addressed object and a message content of sort **MsgContent**. In OO-KLAIM, the configuration that forms a KLAIM net does not only contain KLAIM nodes but also the messages that nodes use to communicate with each other. As mentioned above, this configuration can be thought of as a soup of objects (the nodes) and messages.

```
sort MsgContent .
op msg(.,_) : Oid MsgContent -> Msg [ctor message] .
```

Message contents exist for the *out*, *eval*, *read*, and *in* actions, respectively. The message contents to request a tuple using a *read* or *in* action include the evaluated tuple to match with and an object identifier to send the response to. The response contains a matched tuple in addition to the object identifier the response is from.

```
op remote-out(_) : EvaluatedTuple -> MsgContent [ctor] .
op remote-eval(_) : SyntacticProcess -> MsgContent [ctor] .
op readRequest(.,_) : Oid EvaluatedTuple -> MsgContent [ctor] .
op readResponse(.,_) : Oid EvaluatedTuple -> MsgContent [ctor] .
```

```

op inRequest (_,_) : Oid EvaluatedTuple -> MsgContent [ctor] .
op inResponse (_,_) : Oid EvaluatedTuple -> MsgContent [ctor] .

```

At the process level, the OO-KLAIM specification adds two auxiliary actions, `blockRead` and `blockIn`. These actions are placeholders to block the continuation of a process that is waiting for a response from `read` or `in` actions that address remote KLAIM nodes. Both auxiliary actions carry the tuple the process is waiting for, and an object identifier, which represents the object the response is expected to arrive from.

In the following we show the corresponding rewriting rules for the local and remote rules for the `out` operation (rule (1) and (2)) and the `in` operation (rule (6)). It is of note that the process behavior of M-KLAIM and OO-KLAIM processes differs in the way inter-node communication is handled. In M-KLAIM, if a remote `out` or `eval` action address a node which is not in the KLAIM net, the process that is prefixed by this action cannot proceed. In OO-KLAIM, however, the definition of the rules for the remote `out` and `eval` actions allow a process to proceed even in the case when the action prefixing the process addresses a node that does not exist in the net. This semantic variation is not due to technical limitations, since an object-oriented specification that exactly captures the original semantics could be given. However, we decided to introduce this semantic variation from the original specification to achieve greater flexibility with regard to the design of distributed systems.

Since we now use asynchronous message passing, rules from the M-KLAIM specification are translated to pairs of rules; one rule that produces a message and adds it to the soup, and another rule that actually consumes the message and performs the action.

The semantic rule for an `out` action which addresses a remote node consumes the `out` action and puts a new message into the configuration. The message contains the receiving node's site and the tuple of the `out` action.³

```

cr1 [out-remote-produce] :
  (S1::{RHO} (out(T) @ L) . SP | PP)
=>
  (S1::{RHO} SP | PP) || msg(S2, remote-out(T[| T |]RHO))
if S2 := RHO(L) /\ S2 /= S1 .

```

A node consumes a message that is addressed to it and has the `remote-out` message content in it by putting the evaluated tuple that comes with the message in its tuple space.

```

rl [out-remote-consume] :
  (S::{RHO} PP) || msg(S, remote-out(ET))
=>
  (S::{RHO} PP | out(ET)) .

```

³ In the OO-KLAIM specification, it is no longer necessary to differentiate between remote and local rules, since messages are created and addressed to the appropriate node.

KLAIM's *in* and *read* actions are blocking actions, i.e., a process which is prefixed by such an action can only proceed if a tuple that matches the tuple of the *in* or *read* action is found. In the following we discuss OO-KLAIM's semantics rules only for a remote *in* action, and omit the other rules here. A remote *in* action is consumed by a KLAIM node by putting a request message which addresses the remote node into the configuration. The message contains the tuple of the *in* action and the node's site that processed the *in* action. To simulate the blocking behavior, the node's process is prefixed by a `blockIn` action that contains the nodes site the message is sent to and the tuple of the *in* action.

```

cr1 [in-remote-request] :
  (S1::{RHO} (in(T) @ L) . SP | PP)
=>
  (S1::{RHO} blockIn(ET, S2) . SP | PP)
  || msg(S2, inRequest(S1, ET))
if ET := T[ | T | ]RHO /\ S2 := RHO(L) /\ S2 /= S1 .

```

A node consumes a message that is addressed to it and contains an `inRequest` if a tuple that matches the tuple of the message is present in its tuple space by putting a response message that contains the found tuple into the configuration. The message also contains the node's site that processed the message and is addressed to the node where the request came from.

```

cr1 [in-remote-response] :
  (S2::{RHO} SP | PP | out(ET1)) || msg(S2, inRequest(S1, ET2))
=>
  (S2::{RHO} SP | PP) || msg(S1, inResponse(S2, ET1))
if match(ET1, ET2) .

```

A node consumes a message that is addressed to it and contains an `inResponse` by matching the information of the sender and the tuple that come with the response with the information of a `blockIn` action in its process. If the tuples match, the received tuple is substituted for the tuple of the `blockIn` action in the process that the `blockIn` action prefixes. It is necessary for the receiving node to match the tuples, because a node can send two requests with non-matching tuples to the same remote node.

```

cr1 [in-remote-consume] :
  (S1::{RHO} blockIn(ET1, S2) . SP | PP)
  || msg(S1, inResponse(S2, ET2))
=>
  (S1::{RHO} SP [ ET2 / ET1 ] | PP)
if match(ET1, ET2) .

```

3.3 D-KLAIM

D-KLAIM is an extension of OO-KLAIM that allows specifications to be executed in a distributed environment. This means that D-KLAIM provides a correct-by-construction distributed implementation of OO-KLAIM. In essence,

the D-KLAIM extension allows for multiple instances of Maude to execute specifications based on OO-KLAIM. The OO-KLAIM specification instances communicate with each other through sockets which are handled by objects introduced in D-KLAIM. We use Maude’s support for rewriting with external objects and the predefined implementation of sockets.

For a configuration to communicate with external objects in Maude, the configuration must contain a so-called portal configuration. The default portal is part of the predefined module *CONFIGURATION*. An example for external objects are sockets. Currently, Maude supports IPv4 TCP sockets. The predefined module *SOCKET* of the Maude distribution includes the definition of messages to create, close, and interact with sockets. The messages are consumed by the portal configuration which internally then handles the socket communication.

To support communication through sockets, D-KLAIM adds methods to convert messages to strings which can be sent through sockets and to convert strings back to messages. Additionally, a communicator object is introduced and is part of a KLAIM net. Each Maude instance that executes a D-KLAIM specification has one communicator in the D-KLAIM net term. This communicator then handles the buffered communication between the Maude instances.

Example 3 (A Cloud-based architecture specification based on D-KLAIM). In the D-KLAIM-based specification of the Fibonacci service from Example 1, each participating entity, the developer, the consumer, and the server in the Cloud are modeled as D-KLAIM nets and are executed on distributed machines. Initially, each net contains a KLAIM node, a communicator object, and a portal. Figure 2 provides an overview of the Fibonacci Cloud service architecture. The behavior of the consumer, server, and developer nodes are specified as KLAIM processes.

As an example⁴, the equation

```
ceq initState =
  (IPS :: {[IPS / self]} Fib<nilProcessSeq, nilLocalitySeq,
    nilExpressionSeq> )
if IPS := '192.168.123.50 : 8000
```

specifies the initial configuration of the Fibonacci-Server where `Fib<nilProcessSeq, nilLocalitySeq, nilExpressionSeq>` is a process invocation and `nilProcessSeq`, `nilLocalitySeq`, and `nilExpressionSeq` are the constructors for empty process, locality, and expression sequences. The process definition of `Fib` is specified in the default context, which is specified by the equation

```
eq [cloud-context] : Context =
'Fib (nilProcessVarNameSeq, nilLocalityVarNameSeq, nilVarNameSeq)
  =def in(!u 'Client)@ self . out([0])@ u 'Client{0}
  . out([0], [1])@ self
  . 'FibRec <nilProcessSeq, nilLocalitySeq, nilExpressionSeq> &
'FibRec (nilProcessVarNameSeq, nilLocalityVarNameSeq,
  nilVarNameSeq)
  =def in(! u 'Client)@ self
```

⁴ For the complete specification we refer to [27,9].

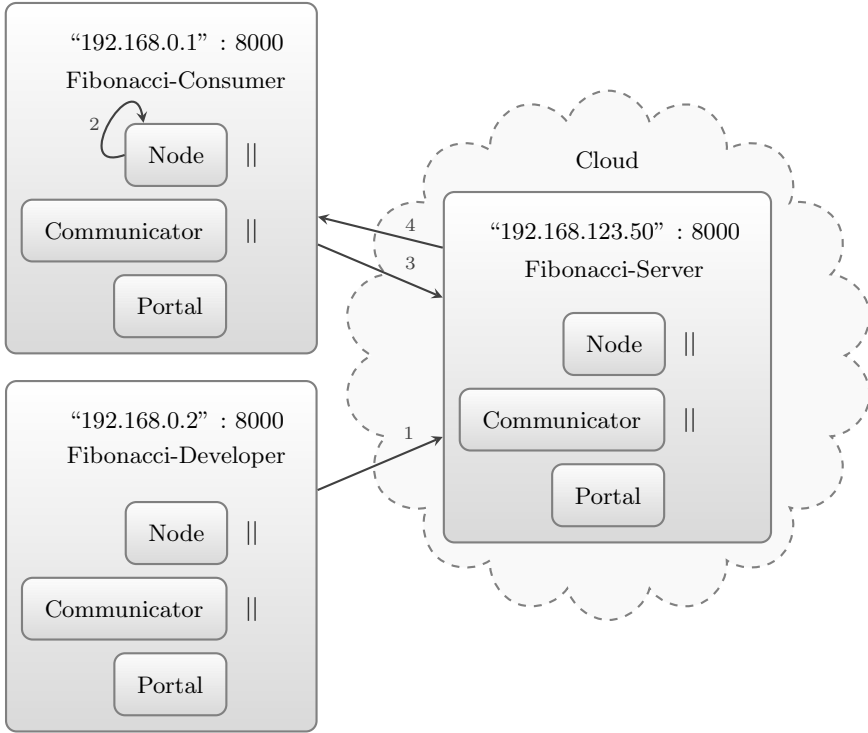


Fig. 2. Overview of the Fibonacci Cloud service architecture

```

. in(! x 'f1, ! x 'f2)@ self
. out(x 'f1{0} +e x 'f2{0})@ u 'Client{0}
. out(x 'f1{0} +e x 'f2{0}, x 'f1{0})@ self
. 'FibRec < nilProcessSeq, nilLocalitySeq, nilExpressionSeq > .

```

The specification can be executed on distributed machines using three Maude instances and the `erew` command. The example of the Cloud-based Fibonacci service shows that a Cloud-based architecture can easily be specified at a high level based on D-KLAIM. Furthermore, the specification can be executed in a distributed environment. In addition to providing the possibility for formal analysis which we will show in the following, this example shows that D-KLAIM and the Maude system can also be used as a rapid prototyping environment for Cloud-based architectures.

4 Formal Design and Analysis of Cloud-Based Systems Using *-KLAIM

As Maude's built-in socket capabilities do not allow for distributed specifications to be model checked, we developed a socket abstraction that captures the behavior of Maude's socket capabilities inside a Maude specification. Using the socket

abstraction, distributed specifications of systems that rely on D-KLAIM's socket communication can be specified in a unified specification. The unified specification can then also be LTL model checked.

4.1 D-KLAIM Socket Abstraction

At a high level of abstraction, a D-KLAIM net is composed of a set of D-KLAIM nodes, which are distributed across possibly different physical machines and which communicate via Maude's built-in socket capabilities. The socket abstraction adds one level of abstraction: the nodes a D-KLAIM net is composed of are put in one single soup (called a network configuration) being executed in one Maude instance. Additionally, the socket abstraction models the socket communication by rewrite rules that capture the behavior of the built-in socket mechanism of Maude. In this way, the built-in sockets are abstracted by rewrite rules so that we are able to perform model checking in D-KLAIM nets by using the socket abstraction.

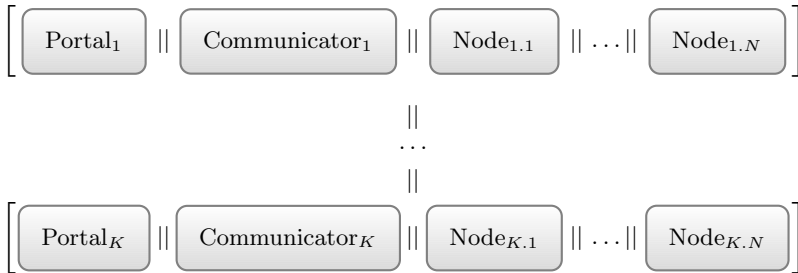


Fig. 3. Overview of a network configuration in the D-KLAIM socket abstraction

Example 4 (Unifying a D-KLAIM specification using the socket abstraction). Let a D-KLAIM specification consist of K configurations. Besides KLAIM nodes, each of these K configurations contains a portal and a communicator. To unify the configurations, we put all these configurations into a single network configuration. A network configuration for a single D-KLAIM net is constructed by the `[_] : Configuration -> NetworkConfiguration` operator. Additionally, the network configurations for the single D-KLAIM nets are concatenated using a `_||_` operator to form the unified network configuration. Figure 3 gives an overview of the unified network configuration for the K D-KLAIM nets using the socket abstraction.

The major difference using the socket abstraction is that instead of a normal portal, an abstract portal is used. The syntax of the abstract portal resembles the syntax of the portal in the predefined `CONFIGURATION` module. It adds a set of attributes which include the IP address and port of the physical location. The object identifier of the abstract portal is the `socketManager` object identifier, which is defined in Maude's `SOCKET` module. The external object with

that identifier that usually handles the creation of sockets is thereby replaced by an internal object in the socket abstraction. In the socket abstraction, sockets are modeled as internal objects that are part of the network configuration. Socket objects are identified by socket identifiers which are constructed by the identifiers of the socket's endpoints. The socket abstraction assumes that only one socket between two endpoints exists. Hence the socket identifier is unique. Socket objects store the content that is sent through the socket, i.e., the content that is just being transferred, keep track of the server and client endpoints and the object that created the socket. States are used by the socket objects and the abstract portal. The abstract portal can be in the states: initialized, listening, or accepting. Sockets can be in the states: initialized, receiving, or idle.

External rewrites in Maude happen only if no internal rewrites are possible. To reflect this in the specification of the socket abstraction, the meta-level is used to define the operator `noInternalTransitions`, which checks if rewrites internal to one of the configurations in a network configuration are possible. For each configuration in the network configuration, the operator calls the operator `metaSearch` that takes the meta-representation of the D-KLAIM socket communication semantics module and the meta-representation of the configuration term as arguments. The additional parameters define the search pattern. In our example, `metaSearch` searches if the configuration can be rewritten to another configuration in at least one rewriting step.

As an example, we show one rule of the socket abstraction to illustrate the simulation of the Maude's built-in sockets by the socket abstraction. The following rule is only applicable if no internal rewrites are possible.

The following rule states, that when a configuration sends data to a socket and the socket is in the receiving state, the socket adds this data to its content.

```

crl [send] :
  [C || send(socket(ID), 0, DATA)]
  || [socket(ID) :: state : receiving, content : CONTENT, ATTS]
=>
  [C || sent(0, socket(ID))]
  || [socket(ID) :: state : receiving, content : (CONTENT + DATA),
      ATTS]
if noInternalTransitions([C]) .

```

4.2 Maude-Based Formal Analysis of *-KLAIM

In the following we explain how specifications based on *-KLAIM (M-KLAIM, OO-KLAIM, and D-KLAIM) can be formally analyzed using the Maude LTL model checker and the Maude search command.

Maude LTL Model Checking. Maude supports on-the-fly explicit state linear temporal logic (LTL) model checking of concurrent systems [117]. Both, the system specification and the property specification are given in Maude. The Maude LTL model checker can be used to prove properties such as safety properties (something bad never happens) and liveness properties (something good will

eventually happen) when the set of states that are reachable from the initial state of a system module is finite. The operators needed for the specification of model checking in Maude are specified in the predefined module *MODEL-CHECKER*.

A *-KLAIM-Based Token-Based Mutual Exclusion Algorithm. This example demonstrates how a token-based mutual exclusion algorithm using a tuple space can be specified and analyzed in *-KLAIM. The goal is to give an executable specification of the algorithm and to model-check if the algorithm fulfills the mutual exclusion property and provides strong liveness guarantees.

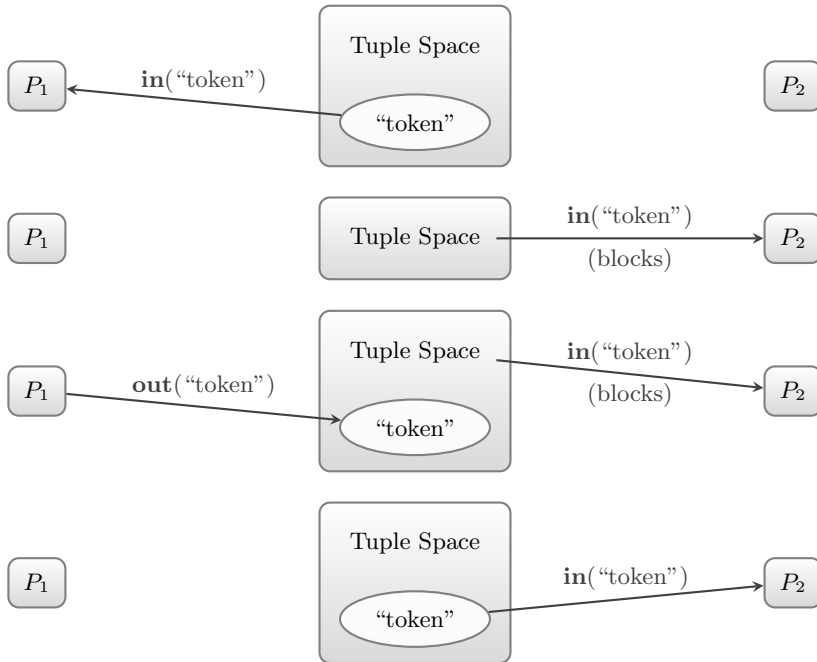


Fig. 4. Example of process synchronization using the Linda model

Figure 4 shows the set-up for this example. The KLAIM net consists of three KLAIM nodes: one token server and two consumers. A token exists in the net and is represented by the value [0]. It is, if available for consumption, present as a tuple in the tuple space of the token server. The consumers can bid for the token by requesting it from the token server. In case the token is available in the tuple space of a consumer, i.e., the token was transferred from the token server to the consumer, the consumer enters its critical section by changing the value of the token. A consumer is defined to be in a critical section if it holds the tuple [1] in its tuple space. A consumer leaves the critical section when it consumes the tuple [1] and puts back the token tuple [0] into the tuple space of the token server.

Defining mutual exclusion and strong liveness. Mutual exclusion and strong liveness are two desirable properties of a mutual exclusion algorithm. In our example, mutual exclusion means that the two consumers are not in their critical sections simultaneously. Strong liveness means that if a consumer requests the token at certain point in time, the consumer eventually gets the token in order to enter its critical section. We first define two auxiliary properties, `critical` and `requesting`, which, for a given site, state if the node at the specified site is in its critical section or is requesting the token from the token server. Mutual exclusion is then defined by the LTL formula

```
[ ] ~ (critical(consumer1) /\ critical(consumer2))
```

and strong liveness of the consumers is defined by the LTL formulas

```
([ ] <> requesting(consumer1)) -> ([ ] <> critical(consumer1))
```

and

```
([ ] <> requesting(consumer2)) -> ([ ] <> critical(consumer2)) .
```

**-KLAIM-based model checking.* In order to perform model checking, we first specify the properties `critical` and `requesting` for each of the KLAIM specifications. As an example, the following listing shows the specification of the two properties based on M-KLAIM.

```
ops critical requesting : Site -> Prop .

var S : Site .
var N : Net .
var AE : AllocationEnvironment .
var P : Process .
var PR : Prop .

eq (S {0}::{AE} out([1]) | P) || N
  |= critical(S) = true .
eq (S {0}::{AE}
  'Request < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >
  || N
  |= requesting(S) = true .
eq N |= PR = false [owise] .
```

Next, we specify the initial state for the model checking of the mutual exclusion algorithm based on each of the KLAIM specifications. Again, for reasons of brevity, only the initial state of the M-KLAIM specification is shown.

```
eq tokenServer = site 'TokenServer .
eq consumer1 = site 'Consumer1 .
eq consumer2 = site 'Consumer2 .

op initial : -> Net .
eq initial =
```

```

(tokenServer {0}:::[tokenServer / self]} out([0])) ||
(consumer1 {0}:::[consumer1 / self] * [tokenServer / '
  TokenServer]})
'Request < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >
||
(consumer2 {0}:::[consumer1 / self] * [tokenServer / '
  TokenServer]})
'Request < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >
.

```

Finally, model checking can be performed. For example, model checking of the mutual exclusion property of the algorithm based on M-KLAIM is achieved by the command

```

Maude> red
  modelCheck(initial, []~ (critical(consumer1) /\ critical(
    consumer2))) .
reduce in KLAIM-MUTEX-CHECK :
  modelCheck(initial, []~ (critical(consumer1) /\ critical(
    consumer2))) .
rewrites: 1016 in 0ms cpu (3ms real) (1395604 rewrites/second)
result Bool: true

```

which shows that the property holds. Model checking of the strong liveness properties is achieved by giving the commands

```

Maude> red
  modelCheck(initial, []<> requesting(consumer1) -> []<> critical(
    consumer1)) .
reduce in KLAIM-MUTEX-CHECK :
  modelCheck(initial, []<> requesting(consumer1) -> []<> critical(
    consumer1)) .
rewrites: 723 in 2ms cpu (2ms real) (263100 rewrites/second)
result [ModelCheckResult]: counterexample(...)

```

and

```

Maude> red
  modelCheck(initial, []<> requesting(consumer2) -> []<> critical(
    consumer2)) .
reduce in KLAIM-MUTEX-CHECK :
  modelCheck(initial, []<> requesting(consumer2) -> []<> critical(
    consumer2)) .
rewrites: 896 in 3ms cpu (4ms real) (288566 rewrites/second)
result [ModelCheckResult]: counterexample(...)

```

which show that the liveness properties do not hold. The counterexamples, which are omitted for reasons of brevity, show that each one of the consumers can starve, i.e., for each consumer a looping path of transitions exists where in each intermediate state the property `critical` does not hold for the consumer.

Model Checking Using the Maude Search Command. Maude's `search` command explores the reachable state space from an initial state for a pattern

that has to be reached, possibly subjected to a user-specified semantic condition. The search can be further restricted by the form of the rewriting proof from the initial to the final term. Possible forms are,

- \Rightarrow^* , which means that a proof of zero, one, or more steps is searched for.
- $\Rightarrow!$, which indicates that only canonical final states, i.e., states where no further rewrites are possible, are searched for.

To model check invariants, i.e., predicates that define a set of states that contain all the states reachable from an initial state, with the `search` command, the optional semantic condition, corresponding to the violation of the invariant, is specified. Under some assumptions, which are omitted here for reasons of simplicity, for any invariant $I(x : k)$ and an initial state *init*, I holds if and only if the command

```
search init =>* x:k such that not I(x : k) .
```

returns no solutions (k is the kind of the term *init*). For an in-depth description of the `search` command we refer to [7].

A D-KLAIM-Based Load Balancer. In this example we show how a simple load balancer based on D-KLAIM can be specified. We then analyze the load balancer using the Maude search command.

Four nodes, a producer, a load balancer, and two consumers, form a KLAIM net. Initially, the producer has four tuples in its tuple space. These tuples can be seen as abstractions of work tasks. The producer then puts each tuple into the tuple space of the load balancer. The load balancer consumes tuples in its tuple space and distributes the tuples across the tuple spaces of the consumers in an alternating order. The expected outcome of the scenario is that each consumer is assigned two work tasks, i.e., each consumer ends up with two tuples in its tuple space. Figure 5 provides a schematic overview of the load balancer example.

We first define the initial configuration of the simple load balancer example based on D-KLAIM and the socket abstraction.

```
ops producer loadBalancer consumer1 consumer2 : -> Site .
eq producer = "192.168.0.100" : 6000 # '0 .
eq loadBalancer = "192.168.0.1" : 8080 # '0 .
eq consumer1 = "192.168.123.1" : 9000 # '0 .
eq consumer2 = "192.168.123.2" : 9000 # '0 .

op loadBalancerExample : -> NetworkConfiguration .
eq loadBalancerExample =
  [startCommunicator("192.168.0.100", 6000, none) ||
   (producer {0}:: {[producer / self] * [loadBalancer / '
     WorkBalancer]})
   out([0]) | out([1]) | out([2]) | out([3]) |
   in(! x 'X)@ self . out(x 'X {0})@ 'WorkBalancer .
   in(! x 'X)@ self . out(x 'X {0})@ 'WorkBalancer .
   in(! x 'X)@ self . out(x 'X {0})@ 'WorkBalancer .
```

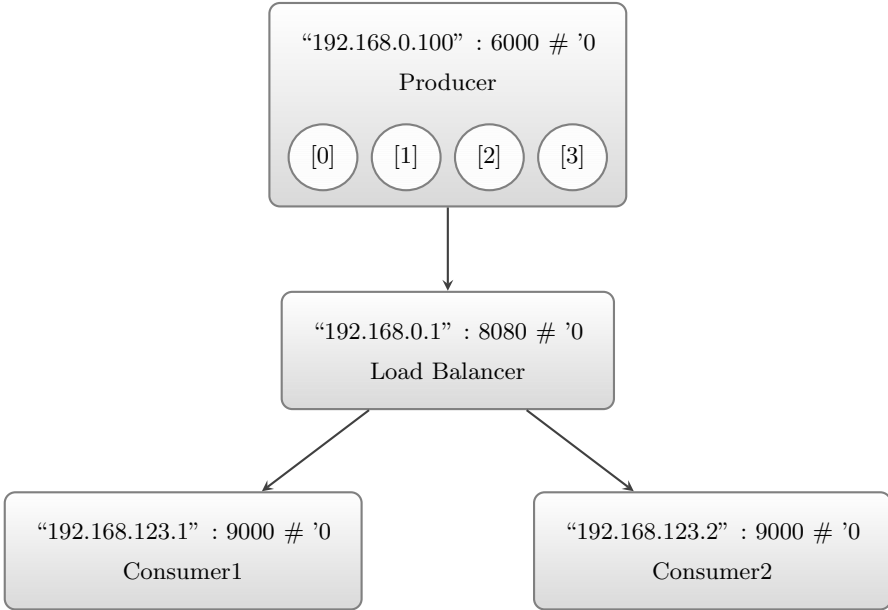


Fig. 5. Schematic overview of the load balancer example

```

in(! x 'X)@ self . out(x 'X {0})@ 'WorkBalancer . nil]] ||
[startCommunicator("192.168.0.1", 8080, none) ||
(loadBalancer {0}::{[loadBalancer / self] *
  [consumer1 / 'Consumer1] * [consumer2 / 'Consumer2]}
in(! x 'X)@ self . out(x 'X {0})@ 'Consumer1 .
in(! x 'X)@ self . out(x 'X {0})@ 'Consumer2 .
in(! x 'X)@ self . out(x 'X {0})@ 'Consumer1 .
in(! x 'X)@ self . out(x 'X {0})@ 'Consumer2 . nil]] ||
[startCommunicator("192.168.123.1", 9000, none) ||
  (consumer1 {0}::{[consumer1 / self]} nil)] ||
[startCommunicator("192.168.123.2", 9000, none) ||
  (consumer2 {0}::{[consumer2 / self]} nil)] .

```

Searching for possible final states. To determine all possible final states, i.e., all states in which no more rewrites are possible, the following Maude search command is used:

```

search in D-KLAIM-ABSTRACTION :
  loadBalancerExample =>! NC:NetworkConfiguration .

```

The search yields six possible final states, which correspond to the possible distribution of tuples across the consumers' tuple spaces. The following table shows the possible final configurations of the tuple spaces. Note that a tuple space is a commutative collection of tuples. E.g., `out([0]) | out([1])` and `out([1]) | out`

([0]) describe the same tuple space. In all cases each consumer ends up with two tuples in its tuple space, as conjectured.

Consumer 1's tuple space		Consumer 2's tuple space	
out([0])	out([1])	out([2])	out([3])
out([0])	out([2])	out([1])	out([3])
out([0])	out([3])	out([1])	out([2])
out([1])	out([2])	out([0])	out([3])
out([1])	out([3])	out([0])	out([2])
out([2])	out([3])	out([0])	out([1])

Model checking of an invariant. An invariant that our simple load balancer example should fulfill is that at no point in time a consumer has more than two tuples in its tuple space. In the following, the variables

```

var NC : NetworkConfiguration .
var C : Configuration .
var A : Site .
var AE : AllocationEnvironment .
var P : Process .
var AP : AuxiliaryProcess .
var SP : SyntacticProcess .

```

are used.

We first define the auxiliary property

```

op lessEqThanTwo : NetworkConfiguration -> Bool .

```

which takes a network configuration as an argument and determines if the two consumers in the network configuration each have less or equal than two tuples in their tuple spaces.

```

op count : Process -> Nat .
eq lessEqThanTwo([C || (A {0}::{AE} P)])
  = if A == consumer1 or A == consumer2 then
    count(P) <= 2
  else false fi .
eq lessEqThanTwo([C] || NC) =
  lessEqThanTwo([C]) or lessEqThanTwo(NC) .
eq count(SP) = 0 .
eq count(AP | P) = s(count(P)) .

```

We then use the command

```

Maude> search loadBalancerExample =>* NC:NetworkConfiguration
such that not lessEqThanTwo(NC) .

```

to model check the invariant. The result is, that the invariant holds as no solutions are found.

4.3 Formal Design and Analysis of Cloud-Based Systems Using *-KLAIM

As we have seen, Cloud-based architectures based on KLAIM can be formally specified and analyzed based on Maude and the *-KLAIM specifications. However, we face the state space explosion problem relatively fast. This is a common issue in model-checking. For example, in the example we performed model-checking, which is a relatively small example, the model checking of the individual properties based on M-KLAIM and OO-KLAIM only took seconds, while the model checking based on D-KLAIM took up to 13 hours on the same machine.

To overcome this scalability restriction, we propose an alternative approach based on a composite actor model together with statistical model-checking. This approach is presented in the next section.

5 Case Studies: Design and Analysis of Cloud-Based Architectures Using the Composite Actor Model

The specification of Cloud-based architectures as composite actor-based models and the formal analysis using statistical model checking is an alternative to the KLAIM-based approach described earlier in this work. This alternative approach promises a more scalable way to analyze quantitative as well as qualitative properties of large Cloud-based architectures.

Modular, distributed, and concurrent systems such as Cloud-based architectures can naturally be modeled as actors. The *actor model of computation* [18,17,1] is a mathematical model of concurrent computation in distributed systems. The main building blocks of a distributed system in the actor model are, as its name suggests, *actors*. Similar to the *object-oriented programming paradigm*, in which the philosophy that *everything is an object* is adopted, the *actor model* follows the paradigm that *everything is an actor*. An *actor* is a concurrent object that encapsulates a state and can be addressed using a unique name. Actors can communicate with each other using asynchronous messages. Upon receiving a message, an actor can change its state and can send messages to other actors.

Temporal logic properties of such actor-based models are model checked either by exact model checking algorithms or, in an approximate but more scalable way, by statistical model checking. The idea of statistical model checking is to verify the satisfaction of a temporal logic property by statistical methods up to a user-specified level of statistical confidence. For this, a large enough number of Monte-Carlo simulations of the system are performed, and the formula is evaluated on each of the simulations. PVESTA [3] is an extension and parallelized version of the VESTA model checking tool. It supports the statistical model checking analysis of probabilistic rewrite theories in Maude. Properties are thereby expressed as QUATEX [2] formulas.

We use a PMaude-based [2] specification of the composite actor model which reflects the so-called “Russian Dolls” model [23] to support an arbitrary hierarchical composition of entities. To guarantee the absence of un-quantified non-determinism, an important prerequisite for statistical model checking, we

developed a top-level scheduling approach for composite actor models. To provide a modular way of specifying these models and to encourage reusability we introduced the notion of formal patterns [10]. Formal patterns enhance pattern descriptions with formal executable specifications that can support the mathematical analysis of qualitative and quantitative properties. Just as “normal patterns”, a formal pattern is structured in context, problem, solution, advantages and shortcomings (cf. e.g. [28][12]). Instead of using UML or Java we describe these patterns formally as a parametrized Maude module $M[S]$ with a parameter theory S in Maude. The context of the pattern typically includes a description of the assumptions of the parameter theory S . Composing these patterns allows the specification of a composite actor systems in a modular and reusable way.

Our methodology to verify Cloud-based architectures proceeds as follows:

1. Specification of an executable formal model of a Cloud-based architecture as a composite actor system in PMAude using a formal pattern-based approach.
2. Definition of appropriate standard probabilistic temporal logic properties and quantitative temporal logic properties for describing the required quality of service properties.
3. Specification of an initial state which contains our top-level scheduler that prevents un-quantified non-determinism.
4. Formal analysis of the properties defined in 2. over the initial state defined in 3. using the statistical model checker PVESTA.

In the following we present three case studies that follow this methodology: the formal specifications and analyses of (i) a key distribution mechanism based on Zookeeper, (ii) a Cloud- and broker-tree-based publish/subscribe system for stock exchange events, and (iii) a Cloud-based DoS prevention mechanism.

5.1 A Key Distribution Mechanism Based on Zookeeper

The Berkeley view on Cloud Computing lists ten main obstacles for the growth and adoption of Cloud Computing; among them is “Bugs in Large Distributed Systems”. This obstacle can be negotiated by early formal analyses. In this case study we modelled a key distribution service and performed formal analysis of basic properties. Based on our results, we detected a serious issue in the design of the service.

Based on the composite actor model, we model a group key management service (GKMS) that was under development at the time of modeling [16]. The GKMS makes use of the notification mechanism of ZooKeeperTM, a centralized open-source server that provides highly reliable distributed coordination; the service is internally distributed and provides high reliability [31][32].

We performed formal analyses, e.g., we formally analyzed the “success ratio”, i.e., how many of the total key updates actually do arrive at the members of the GKMS in time? A key property of a group-key management system is the freshness of the keys, i.e., key updates need to arrive in time at the members of the group. The results show that the freshness property of the group-key

management system is only poorly satisfied. In a system with 420 clients just about 21% of the key updates arrive at the clients. This is a serious design issue of the GKMS. We analyzed the system and highlighted two main shortcomings of the approach. Furthermore, we suggested as future work an adaption of the GKMS to overcome the shortcomings.

5.2 A Publish/Subscribe System for Stock Exchange Events

Broker-based Publish/Subscribe is an asynchronous message exchange pattern, in which producers and consumers of messages are loosely coupled and communicate via brokers. Brokers are intermediaries that can select and forward the published information relevant to each consumer. Quality of service (QoS) properties such as an on-time delivery of messages are crucial. It is the task of the service management to determine the performance of the system so that quality of service guarantees can be given. However, several parameters and the unreliability of best-effort networks, especially when deployed in a worldwide setting, make it difficult to analyze such systems.

We defined a Maude-based concrete model of a stock exchange information system that provides current trading information similar to Google finance [15]. The model is based on the composite actor model with publishers, subscribers, and brokers being actors communicating via message passing. Subscribers in this model can subscribe to trading events that show specific characteristics, such as being related to a certain listing or being a high-volume trade. In our setting, several brokers are located at sites across the world where initially there is only a single broker at a site. Each trading event has a specific lifetime, and the event information is only useful to subscribers if they receive it within its lifetime. In our model, we assume an average frequency (one event per producer per second) and low lifetime duration (as low as one second) event dissemination. This represents a challenging setting, as communication latencies in a best-effort worldwide network already consume a big fraction, if not all, of the lifetime. Additionally, the system is flooded with events which may lead to high workloads at the brokers. We asked three questions regarding this kind of stock exchange information system:

1. Can a QoS guarantee be made regarding the timely delivery of events?
2. Does the system scale, i.e., how many subscribers can the system handle without violating the aforementioned QoS guarantee?
3. How many resources should the service provider provision, and what are the expected operating costs?

In a first step, we formally analyzed the model of the stock exchange information system regarding these questions using statistical model checking. The statistical model checking analysis of the model suggested that with event lifetime durations of 1s, 30s, and 60s, it is hard to guarantee a timely delivery of events in a system that is distributed across the globe; and that the reliability of such a system can be improved by reducing the processing delays of intermediaries in the message

forwarding mechanism. In a second step, we added a simple Cloud-based broker replication mechanism to the model which allows each broker in the network to replicate itself at its current site. The analysis of the Cloud-based model has shown that the reliability of the system could be greatly increased using the replication mechanism. However, the local replication strategy does not, without overprovisioning, provide strong industry-grade quality of service guarantees. Again, even with broker replication, event lifetime durations of under 60s are a challenging assumption for a global-scale system.

5.3 A Cloud-Based DoS Prevention Mechanism

Availability is an important security property for Internet services and a key ingredient of most service level agreements. It can be compromised by distributed Denial of Service (DoS) attacks. DoS defense mechanisms help maintaining availability; nevertheless even when equipped with defense mechanisms, systems will typically show performance degradation. Thus, one of the goals of security measures is to achieve stable availability, which means that with very high probability service quality remains very close to a constant quantity, which does not change over time, regardless of how bad the DoS attack can get [10].

We used our composite actor-based approach and formal patterns to study defense mechanisms against DoS attacks in a model-based setting. Two formal patterns which can serve as defenses against DoS attacks have been studied: (i) the Adaptive Selective Verification (ASV) [20] pattern defending against DoS attacks and (ii) the Server Replicator (SR) pattern in a cloud setting.

The ASV protocol is a well-known cost-based defense mechanism against DoS attacks in the typical situation that clients and attackers use a shared channel where neither the attacker nor the client have full control over the communication channel [20]. The ASV protocol adapts to increasingly severe DoS attacks and provides improved availability. However, it cannot provide stable availability. The SR pattern utilizes the Cloud's ability to dynamically allocate new resources on demand to adapt to high demand situations and achieve stable availability. However the cost of provisioned servers drastically increases in a DoS attack situation. As thus, we proposed the pattern composition of ASV and SR, ASV+SR, as a third defense mechanism.

We formally specified the ASV, SR, and ASV+SR DoS defense mechanisms as formal patterns in Maude and applied them on a Cloud-based client-server request response service. Figure 6 shows the application of the ASV+SR meta-object pattern composition on such a service. By formally analyzing the quantitative properties of the ASV+SR pattern in this setting using the statistical model checker PVESTA [3], we were able to show that, unlike the two original patterns, ASV+SR achieves stable availability in presence of a large number of attackers at reasonable cost, which can be predictably controlled by the choice of an overloading parameter. This overloading factor describes how much load in form of requests is put on each server before a new server is allocated and reflects how much the ASV mechanism is used compared to the SR mechanism. In particular, we were checking the following availability properties: the client

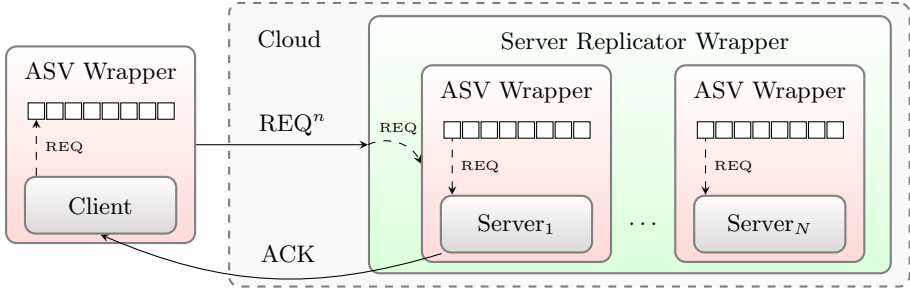


Fig. 6. Application of the ASV^+SR meta-object composition on a Cloud-based client-server request-response service

success ratio, which defines the ratio of clients that receive an ACK from the server (i.e., the ratio of real clients that were able to successfully receive a response); and the average time to service (TTS), which defines the average time it takes for a successful client to receive an ACK from the server. In addition to these availability properties, we also checked the number of provisioned servers. Each of the properties was defined as a QUATEX formula [2]. The results show that in our scenario for 200 attackers and an overloading factor of 4, pure ASV provides a success ratio of 76% and an average TTS of 2.4s using 1 server; pure SR provides a success ratio of 100%, an average TTS of 0.25s, and provisions 135 servers; finally, ASV^+SR provides a 97% success ratio, an average TTS of 0.70s and provisions only 43 servers. A full description of the specification and the formal analysis is given in [10,27,9].

6 Conclusion

In this paper we have studied the coordination language KLAIM and a modularized actor approach with the aim of formally modelling Cloud-based architectures. We have presented an executable rewriting semantics of KLAIM and its distributed implementation in Maude, where multiple instances of Maude communicate via sockets. By simulating and analyzing three simple Cloud architectures with Maude and the Maude LTL modelchecker we have shown that it is possible to formally specify and analyze Cloud-based architectures using KLAIM and Maude, but that state space explosion makes it hard to deal with complex scenarios. To tackle this scalability issue we have proposed a composite actor model as an alternative approach. We report on three Cloud case studies which we have formally specified and analyzed using Maude and the Maude-based statistical model checker PVeStA, and argue that this model is also well-suited for modeling Cloud architectures and, in addition, scales even better for the purpose of qualitative and quantitative analysis.

These are encouraging results. Formal models, and, in particular, the rewriting logic approach with its Maude tools are well-suited for the design of Cloud-based system architectures. Checking for formal guarantees can help to tackle and to

overcome some of the main obstacles for the growth and adoption of Cloud Computing. Executable formal methods can help to make John McCarthy's dream a reality.

Of course, the approach and the case studies we have presented are just a proof of concept. We need to carry out more case studies; scalability is and will remain an issue for automated formal analysis. To make progress we have to exploit compositionality properties and to develop even better abstraction, refinement, and model transformation techniques. The tools for making progress are to exploit compositionality properties and to develop even better abstraction, refinement, and model transformation techniques. For example, in [6] it is shown that adding cookies to a client-server system preserves all safety properties, and in [22] it is proven that the Physically Asynchronous Logically Synchronous (PALS) architectural pattern reduces the design and verification of a distributed real-time system with asynchronous communication to that of its much simpler synchronous version. In the realm of process algebra, fluid-flow approximation using computationally inexpensive ordinary differential equations provides an excellent complementary tool for performance evaluation of Cloud systems [33].

References

1. Agha, G.: *Actors: a model of concurrent computation in distributed systems*. MIT Press (1986)
2. Agha, G., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. *ENTCS* 153(2), 213–239 (2006)
3. AlTurki, M., Meseguer, J.: PVESTA: A Parallel Statistical Model Checking and Quantitative Analysis Tool. In: Corradini, A., Klin, B., Cirstea, C. (eds.) *CALCO 2011*. LNCS, vol. 6859, pp. 386–392. Springer, Heidelberg (2011)
4. Braga, C., Meseguer, J.: *Modular Rewriting Semantics in Practice*. *ENTCS* 117, 393–416 (2005)
5. Carriero, N., Gelernter, D.: Linda in context. *Communications of the ACM* 32, 444–458 (1989)
6. Chadha, R., Gunter, C.A., Meseguer, J., Shankesi, R., Viswanathan, M.: Modular Preservation of Safety Properties by Cookie-Based DoS-Protection Wrappers. In: Barthe, G., de Boer, F.S. (eds.) *FMOODS 2008*. LNCS, vol. 5051, pp. 39–58. Springer, Heidelberg (2008)
7. 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)
8. De Nicola, R., Ferrari, G.L., Pugliese, R.: KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE TSE* 24, 315–330 (1998)
9. Eckhardt, J.: *A Formal Analysis of Security Properties in Cloud Computing*. Master's thesis, LMU Munich, TU Munich (2011)
10. Eckhardt, J., Mühlbauer, T., AlTurki, M., Meseguer, J., Wirsing, M.: Stable Availability under Denial of Service Attacks through Formal Patterns. In: de Lara, J., Zisman, A. (eds.) *FASE 2012*. LNCS, vol. 7212, pp. 78–93. Springer, Heidelberg (2012)
11. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: *WRLA*. *ENTCS*, vol. 71, pp. 162–187 (2004)

12. Erl, T.: SOA Design Patterns. Prentice Hall (2008)
13. Garfinkel, S.L.: Architects of the Information Society. MIT Press (1999)
14. Gelernter, D.: Generative communication in Linda. TOPLAS 7, 80–112 (1985)
15. Google. Google finance, <http://finance.google.com/> (visited: September 2011)
16. Gupta, J.: Available group key management for NASPInet. Master's thesis, University of Illinois in Urbana-Champaign (2011)
17. Hewitt, C., Baker, H.G.: Laws for communicating parallel processes. In: IFIP, pp. 987–992 (1977)
18. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: IJCAI, pp. 235–245 (1973)
19. Guseva, I.: Gartner: Top Technology Predictions for 2010 and Beyond (2010), <http://cmswire.com/cms/enterprise-20/gartner-top-technology-predictions-for-2010-and-beyond-006390.php#singlereqrespmp> (visited April 2012)
20. Khanna, S., Venkatesh, S., Fatemeh, O., Khan, F., Gunter, C.: Adaptive Selective Verification. In: IEEE INFOCOM, pp. 529–537 (2008)
21. Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. TCS 96(1), 73–155 (1992)
22. Meseguer, J., Ölveczky, P.C.: Formalization and Correctness of the PALS Architectural Pattern for Distributed Real-Time Systems. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 303–320. Springer, Heidelberg (2010)
23. Meseguer, J., Talcott, C.: Semantic Models for Distributed Object Reflection. In: Deng, T. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 1–36. Springer, Heidelberg (2002)
24. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
25. Milner, R.: Communicating and Mobile Systems: the π -Calculus. Cambridge University Press (1999)
26. Minkiewicz, A.: Cloud Nine, Are we there yet? JST 14(4) (2011)
27. Mühlbauer, T.: Formal Specification and Analysis of Cloud Computing Management. Master's thesis, LMU Munich, TU Munich (2011)
28. Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns. Wiley (2005)
29. Serbanuta, T.-F., Rosu, G., Meseguer, J.: A rewriting logic approach to operational semantics. Information and Computation 207, 305–340 (2009)
30. Samson, T.: Feds take cloud-first approach to IT. In: InfoWorld TechWatch, December 8 (2010), <http://www.infoworld.com/t/cloud-computing/feds-take-cloud-first-approach-it-829> (visited: April 2012)
31. The Apache Software Foundation. Apache Zookeeper, <http://zookeeper.apache.org/> (visited: September 2011)
32. The Apache Software Foundation. Apache Zookeeper Wiki, <https://cwiki.apache.org/confluence/display/ZOOKEEPER/ProjectDescription> (visited: September 2011)
33. Tribastone, M., Gilmore, S.: Scaling Performance Analysis Using Fluid-Flow Approximation. In: Wirsing, M., Hölzl, M. (eds.) SENSORIA. LNCS, vol. 6582, pp. 486–505. Springer, Heidelberg (2011)
34. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. JLAP 67(1-2), 226–293 (2006)

Making Maude Definitions More Interactive

Andrei Arusoaie¹, Traian Florin Șerbănuță^{1,2},
Chucky Ellison², and Grigore Roșu²

¹ University Alexandru Ioan Cuza of Iași
{andrei.arusoaie,traian.serbanuta}@info.uaic.ro
² University of Illinois at Urbana-Champaign
{tserban2,celliso2,grosu}@illinois.edu

Abstract. This paper presents an interface for achieving interactive executions of Maude terms by allowing console and file input/output (I/O) operations. This interface consists of a Maude API for I/O operations, a Java-based server offering I/O capabilities, and a communication protocol between the two implemented using the external objects concept and Maude’s TCP sockets. This interface was evaluated as part of the \mathbb{K} framework, providing interactive interpreter capabilities for executing and testing programs for multiple language definitions.

Keywords: Maude, interactive, input/output, API.

1 Introduction

Formal specifications are often used when designing complex systems. The executability aspect of rewriting logic [1], and Maude’s ability to efficiently execute, explore, and analyze rewrite theories [5] offers an additional level of support when designing a new system, as it allows the designer to experiment, test, and revise a specification before deciding to implement it. In certain cases, it even allows for the specification to become the implementation, eliminating the need of building another executable model. However, many systems include a human component, who is allowed/required to provide input to the system for directing its evolution.

To handle interaction, Maude provides the read-eval-print loop in the LOOP-MODE standard module, but this allows only for very limited user interaction. Furthermore, according to the Maude manual [4], it “may not be maintained in future versions, because the support for communication with external objects makes it possible to develop more general and flexible solutions for dealing with input/output in future releases.” While Maude’s external objects do allow interaction in principle, they are low-level and cannot be used for generic input/output (I/O) without significant infrastructure external to Maude. Our contribution is to provide this infrastructure and to develop an easy to use interface for it within Maude.

Figure 1 presents a high-level view of the I/O interface. It includes a Maude API for I/O, a Java-based server for handling requests, and a protocol for delivering queries and transmitting the responses. Using this interface, potentially

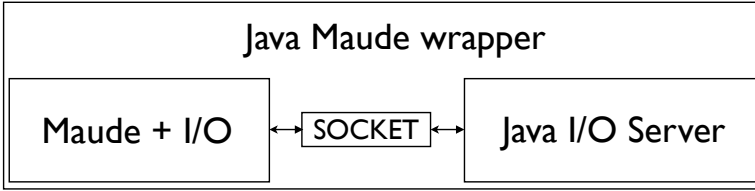


Fig. 1. The architecture of the Maude I/O interface

any Maude definition can be enhanced with I/O capabilities. A Java wrapper which runs on top of both Maude and the Java server allows for the user to experience interpreter-like behavior: the console in which the program is started is the one interactively displaying the output of the program and requesting input. In addition to console I/O, this new framework also provides support for file I/O, including both sequential and random-access. Moreover, all of these actions are allowed to take place potentially anywhere in the term.

The Maude I/O interface and the examples presented in this paper, as well as the Java I/O server are available online [3,2]. Additionally, this interface has been used as part of the \mathbb{K} framework tool [1] to obtain interpreter-like behavior for a number of language definitions, including a definition of the C language [8].

The remainder of this paper is structured as follows. Section 2 describes the I/O interface from a user point-of-view and illustrates some usage patterns through examples. Section 3 details the implementation, both for the Maude I/O client and the Java I/O server components. Section 4 reviews related work and Section 5 concludes.

2 The I/O Interface

The basic standard I/O interface exposes several I/O commands for the standard input and standard output streams defined in the STANDARD-IO module:

```

op #printString : String → IOResult .      op #printChar : Char → IOResult .
op #readInt() : → IOResult .                op #readChar() : → IOResult .
op #eof() : → IOResult .                    op #readToken() : → IOResult .
  
```

The resulting sort for these directives is “IOResult” which is defined in the “IO-INTERFACE” together with several constructors for it:

```

op #success : → IOResult .
op #string : [String] → IOResult .
op #int : Int → IOResult .
op #char : [Char] → IOResult .
op #ioError : String → IOResult .
op #flag : [Bool] → IOResult .
op #eof : → IOResult .
  
```

`#printString` sends an entire string, character by character, to the standard output stream and returns `#success`. `#readInt` reads a token and returns an `#int` containing the number read. `#eof()` tests the standard input stream for the end of file and returns a `#flag` result with the argument set appropriately. `#printChar` sends

one character to the standard output stream and returns `#success`. `#readChar` reads a character from the standard input stream and returns a `#char` or `#eof`. Finally, `#readToken` skips the whitespace from the standard input and then returns a `#string` containing all characters read until the next whitespace or the end of file is encountered, or `#eof` if the end of file is reached while skipping over whitespace. In case of an I/O error or a communication error, an `#ioError` term detailing the error is produced. We will present their formal definition in the next section, but in the meanwhile, let us start with some examples.

2.1 Example 1: A Straightforward Usage of the I/O Interface

Let us begin with an example showing how our I/O interface can be used in a Maude definition. For this we have chosen a very simple expression language, called EXP. The following module defines its syntax:

```

mod EXP-SYNTAX is
  including INT .
  including STRING .
  sort Exp .
  subsort Int < Exp .
  op _+_ : Exp Exp → Exp [ditto] .
  op *_ : Exp Exp → Exp [ditto] .
  op _ifnz_ : Exp Exp → Exp [strat(2 0)] .
  op nzloop : Exp → Exp [strat (0)] .
  op input : String → Exp .
  op print : String Exp → Exp .
endm

```

EXP has integers as basic values and extends two of the integer operations: addition and multiplication. Moreover, it provides a guarded expression, `ifnz`, which evaluates its first argument only if the evaluation of the second one produces a non-zero value, and a fix-point operator, `nzloop`, which evaluates its argument as long as its value is not zero. Note that in the absence of any side effects, the `nzloop` construct is rather non-sensical, as its argument would always evaluate to the same value. However, adding I/O constructs to the language allows for some interesting (albeit simple) programs to be written in this language, like, for example:

```

nzloop(print("3*x=", 3 * input("x= (0 to stop)? ")))

```

The intended meaning of `input` construct is somehow similar to the `INPUT` instruction of the BASIC language [6], in that it prompts the string in its argument to the user and expects an integer to be entered, returning that integer. The meaning of `print` is that it prints the string in the first argument, then prints the string representation of the second argument (which is expected to be evaluated to an integer), and then advances the line feed. With this intuition in mind, the semantics of the program above is that reads a number “x” from the console and computes and displays “3*x” until the number entered is 0 (included).

The module in Figure 2 formally defines the semantics described above. Assuming this module is included in a file named `io-test.maude` (which also loads the `io-interface.maude` file) and that the Maude command for rewriting (with external objects) the above program is written in a file named `io-test-cmd.maude`, the following command “executes” the program interactively:

```

mod EXP-SEMANTICS is
  including EXP-SYNTAX . including STANDARD-IO .
  op read :  $\rightarrow$  Exp .

  eq input(S)                               eq print(S,I)
    = #printString(S);                       = #printString(S + string(I,10) + "\n");
    #readInt();                               I .
    read .

  op _;_ : IOResult Exp  $\rightarrow$  Exp
                                         [strat (1 0)] .

  eq nzloop(E) = nzloop(E) ifnz E .
  eq E ifnz 0 = 0 .
  eq E ifnz NzI = E .

  var I : Int . var S : String . var NzI : NzInt . var E : Exp .
endm

```

Fig. 2. A Straight-forward semantics for the EXP language

```

java -jar MaudeIO.jar \
  --maudeFile io-test.maude \
  --moduleName EXP-SEMANTICS \
  --commandFile io-test-cmd.maude

```

Here is a possible interaction sequence between the user and the tool:

```

x= (0 to stop)? 5
3*x=15
x= (0 to stop)? 10
3*x=30
x= (0 to stop)? 7
3*x=21
x= (0 to stop)? 0
3*x=0
Maude> =====
rewrite in KRUNNER : nzloop(print("3*x=", 3 * input("x= (0 to stop)? "))) .
rewrites: 8452 in 59ms cpu (5345ms real) (141321 rewrites/second)
result Zero: 0
Maude> Bye.

```

Allowing I/O operations anywhere in the term/program provides a high degree of flexibility, but at a price. As running multiple I/O commands at the same time creates race conditions, the user has to provide mechanisms to avoid these races. One such example is the “`_;_`” command defined in the semantics above whose only purpose is to ensure that the printing command completes before the next command is executed (enforced by the strategy annotation), resembling the `_>>_` sequentialization operator of the Haskell I/O monad [13].

However, parallel I/O commands are still possible in our language, producing potentially undesirable effects. For example, when executing the program

```
print("Hello ",1) + print("World!",2)
```

a possible result would be the following:

```
Hwoerllldo! 2
1
Maude> =====
rewrite in KRUNNER : print("Hello ", 1) + print("World!", 2) .
rewrites: 1525 in 9ms cpu (23ms real) (160374 rewrites/second)
result NzNat: 3
```

While this may be acceptable behavior, it is also possible to avoid it if not. This can be done by sequentializing the two printing expressions programatically, e.g., by relying on the strategy of `ifnz`, like `print("World!",2) ifnz print("Hello ",1)`.

2.2 Interaction with Maude’s Analysis Tools

The definition presented above is very simple, but it can also easily become quite chaotic, due to its very direct use of I/O (through external objects). It is also impossible to test it in the absence of the I/O server. However, it is quite easy to add I/O as a natural extension to specifications which are already amenable for testing, exploration, and analysis.

Figure 3 presents an SOS-style rewriting logic semantics [17,12] for the EXP language. To simulate input and output it uses a configuration which, in addition to the program to be evaluated, contains a list of integers for input and a string buffer for output. The small-step semantics is given through the rules for the `•_` operation which defines the application of a single computation step in an SOS style. Also note that the semantics of `input` has changed, by creating separate small steps for the prompt and read operations.

Given an input list, this semantics can be tested, explored, and even model-checked using Maude. Moreover, using the proposed I/O interface, it is quite easy to turn it into an I/O-enabled interpreter as shown in Figure 4.

One relatively simple way to achieve input capabilities is by adding a special input constant `requestInt` and an equation for requesting an int when the input list becomes empty (lines 8-9). This request will be propagated to the top level of EXP SOS configuration.

To extend the semantics with standard I/O we encapsulate the existing state of EXP within a new state. At each iteration this configuration will execute one step using the old configuration and then it will flush the input/output buffers (lines 11-17). The output is flushed using the equation shown in lines 19-20 by sending the output string to the standard output stream, and then returning the new configuration once the printing succeeded (line 25). The request for input is translated into a read from the standard input stream which is then added to the input list on success (line 26). Note that these operations are sequenced using the same idea as in the previous definition.

Finally, when the execution has completed the result of the original computation is retrieved from the encapsulated state (lines 29-30).

```

mod EXP-SEMANTICS is including EXP-SYNTAX .
  including LIST{Int} . including CONVERSION .

  sort State .
  op {_,_,_} : Exp List{Int} String → State .
  ops •_*_* : State ⇔ State .
  op read : → Exp .

  var I : Int . var S : String . var NzI : NzInt . var E E1 E2 E' E1' E2' : Exp .
  var In In' : List{Int} . var Out Out' : String . var State : State .

  eq * State = * . State .
  cr1 •{E1 * E2,In,Out} ⇒ {E1' * E2,In',Out'}
    if •{E1,In,Out} ⇒ {E1',In',Out'} .
  cr1 •{E1 + E2,In,Out} ⇒ {E1' + E2,In',Out'}
    if •{E1,In,Out} ⇒ {E1',In',Out'} .
  cr1 •{print(S,E),In,Out} ⇒ {print(S,E'),In',Out'}
    if •{E,In,Out} ⇒ {E',In',Out'} .
  rl •{input(S),In,Out} ⇒ {read,In,Out + S} .
  rl •{read,I In, Out} ⇒ {I, In, Out} .
  rl •{print(S,I),In,Out} ⇒ {I,In,Out + S + string(I,10) + "\n"} .
  cr1 •{E2 ifnz E1,In,Out} ⇒ {E2 ifnz E1',In',Out'}
    if •{E1,In,Out} ⇒ {E1',In',Out'} .
  rl •{E ifnz 0,In,Out} ⇒ {0,In,Out} .
  rl •{E ifnz NzI,In,Out} ⇒ {E,In,Out} .
  rl •{nzloop(E),In,Out} ⇒ {nzloop(E) ifnz E,In,Out} .
endm

```

Fig. 3. An SOS-like semantics for EXP in Maude

Since in this particular example the communication with the I/O server is enforced to occur at the top of the term, this semantics guarantees the sequencing of the print statements above, producing a reasonable output:

```

Hello 1
World!2
Maude> =====
erewrite in KRUNNER : * {{print("Hello ", 1) + print("World!", 2),nil,""}} .
rewrites: 1055 in 9ms cpu (58ms real) (114624 rewrites/second)
result IOState: result(3)
Maude> Bye.

```

2.3 The File I/O Interface

As detailed in the next section, the simple I/O interface exhibited above is implemented in terms of a file I/O interface, defined by the IO-INTERFACE module, which provides file-handle-parameterized versions of its basic I/O commands:


```

1 mod EXP-STANDARD-IO is
2   including EXP-SEMANTICS .
3   including STANDARD-IO .
4
5   var N : Nat .   var I : Int .   var E : Exp .
6   var In : List {Int} .   var Out : String .   var State : State .
7
8   op requestInt :  $\rightarrow$  List {Int} .
9   rl •{read, nil , Out}  $\Rightarrow$  {read, requestInt , Out} .
10
11  sort IOState .
12  op { } : State  $\rightarrow$  IOState .
13  op { } flush : State  $\rightarrow$  IOState .
14  op * _ : IOState  $\rightsquigarrow$  IOState .
15  eq *{State} = *{. State} flush .
16  ceq {{E, In, ""} flush = {{E, In, ""}
17    if In  $\neq$  requestInt .
18
19  ceq {{E, In, Out} flush = {#printString(Out); {E, In, ""} flush
20    if Out  $\neq$  "" .
21
22  eq {{E, requestInt, ""} flush = {#readInt(); {E, nil, ""} flush .
23
24  op _ ; _ : IOResult State  $\rightarrow$  State .
25  eq #success ; State = State .
26  eq #int(I) ; {E, In, Out} = {E, I In, Out} .
27
28  — Catching the result of computation
29  op result : Int  $\rightarrow$  IOState .
30  eq *{. {I, nil, ""} flush = result(I) .
31
32  endm

```

Fig. 4. Extending EXP-SEMANTICS with standard I/O capabilities

```

op #fPrintString : Nat String  $\rightarrow$  IOResult .   op #fPrintChar : Nat Char  $\rightarrow$  IOResult .
op #fReadInt : Nat  $\rightarrow$  IOResult .               op #fReadChar : Nat  $\rightarrow$  IOResult .
op #fEof : Nat  $\rightarrow$  IOResult .                  op #fReadToken : Nat  $\rightarrow$  IOResult .

```

In addition to those, it also provides commands for opening and closing files,

```

op #open : String  $\rightarrow$  IOResult .   — opens a file , mapping it to a #handle
op #reopen : Nat String  $\rightarrow$  IOResult . — maps a different file to the #handle
op #close : Nat  $\rightarrow$  IOResult .     — closes the file mapped to the #handle

```

as well as several lower-level commands for accessing the contents of a file:

```

op #fReadByte : Nat  $\rightarrow$  IOResult .   op #flush : Nat  $\rightarrow$  IOResult .
op #fPutByte : Nat Nat  $\rightarrow$  IOResult . op #tell : Nat  $\rightarrow$  IOResult .
op #fPeekByte : Nat  $\rightarrow$  IOResult .    op #seek : Nat Nat  $\rightarrow$  IOResult .

```

As a simple example of how this more advanced interface could be used, can be seen in Figure 5 which presents a generalization of the I/O semantics from Figure 4, by replacing standard I/O with file I/O. To achieve that, the encapsulated configuration is extended with two parameters: the first for the input stream handle, and the second for the output stream handle. Then the functions for achieving standard I/O are replaced with the corresponding ones for the file I/O, using these stream handles as parameters.

```

1  mod EXP-IO is including EXP-SEMANTICS . including IO-INTERFACE .
2
3  op requestInt : → List {Int} .
4  rl •{read, nil, Out} ⇒ {read, requestInt, Out} .
5
6  sort IOState .
7  op {_,_,_} : State IOResult IOResult → IOState .
8  op {_,_,_}flush : State IOResult IOResult → IOState .
9  op *_ : IOState ~> IOState .
10 eq *{State, HIn, HOut} = *{. State, HIn, HOut}flush .
11 ceq {{E, In, ""}, HIn, HOut}flush = {{E, In, ""}, HIn, HOut}
12   if In =/= requestInt .
13
14 var N : Nat . var I : Int . var E : Exp . var HIn HOut : IOResult .
15 var In : List {Int} . var Out : String . var State : State .
16
17 ceq {{E, In, Out}, HIn, #handle(N)}flush
18   = {#fPrintString(N, Out); {E, In, ""}, HIn, #handle(N)}flush
19   if Out =/= "" .
20
21 eq {{E, requestInt, ""}, #handle(N), HOut}flush
22   = {#fReadInt(N); {E, nil, ""}, #handle(N), HOut}flush .
23
24 op _;_ : IOResult State → State .
25 eq #success ; State = State .
26 eq #int(I) ; {E, In, Out} = {E, I In, Out} .
27
28 — Catching the result of computation
29 op result : Int → IOState .
30 eq *{. {I, nil, ""}, HIn, HOut}flush = result(I) .
31 endm

```

Fig. 5. Extending EXP-SEMANTICS with file I/O capabilities

For example, assuming the contents of the `test.in` file are “3 -5 0”, rewriting with the following command:

```

erew * {{
nzloop(print("3*x=", 3 * input("x= (0 to stop)? "))
, nil, ""),

```

```
#open("file:test.in#r"),
#open("file:test.out#w")} .
```

will produce the output:

```
Maude> =====
erewrite in KRUNNER : * {{nzloop(print("3*x=",
  3 * input("x= (0 to stop)? "))), nil, ""},
  #open("file:test.in#r"),#open("file:test.out#w")} .
rewrites: 4539 in 33ms cpu (70ms real) (133767 rewrites/second)
result IOState: result(0)
Maude> Bye.
```

Upon the completion of the command the “test.out” file contains:

```
x= (0 to stop)? 3*x=9
x= (0 to stop)? 3*x=-15
x= (0 to stop)? 3*x=0
```

The argument of `#open` contains the usual URI description of a file location before the “#” symbol, while the “r” after the symbol specifies that the file should be opened for read-only access and “w” specifies that it should be opened for write-only.

2.4 Separating the I/O Server from Maude

The common usage pattern for the I/O interface presented above is that both Maude and the Java I/O Server are executed as subprocesses of the same Java wrapper application. This way, users interact directly with the console they used to start the execution of the program. Moreover, as this close integration uses a fresh TCP port for communication, this allows multiple instances of Maude using I/O to be running at the same time. Unfortunately this hides the Maude console and inhibits the user from interacting with Maude directly. As sometimes it might be useful to have both the I/O console and the Maude console available, let us describe how this can be achieved.

First, one needs to fix the port on which communication takes place. To do so, it needs to add an equation of the form “`eq #TCPPORT = 8001 .`” either in the definition using the I/O interface, or right after the definition of the `#TCPPORT` constant in the `tcp-interface.maude` file (if this behavior is desired for all definitions). Then, the server must be started first in a console taking as parameter the same port number:

```
java -jar ioserver.jar 8001
```

Once the server is running, the definition can be loaded into Maude in a separate console. Now we have two consoles, communicating between them. The rewriting commands can be given using the normal Maude console and the Maude result will be displayed here, while the I/O messages will be displayed in the console running the I/O server and I/O input will be requested from there.

3 Implementation

This section comes to give additional details for the implementation of the I/O interface described in the prior sections.

As hinted in the introduction and depicted in Figure 1, both the I/O Server and Maude are wrapped by a Java process which is intended to offer to the user the console he expects when a program is executed. The wrapper hides some operations that a casual user would not care about. This wrapper is executed as a normal Java jar, taking as arguments the file containing the Maude definition, the name of the main module, and the file containing the rewrite command to be executed, as exhibited in Section 2.1. The main purpose of this wrapper is to automatically setup Maude and the I/O Server. An unused TCP port p is identified and an equation:

$$\text{eq } \#TCP\text{PORT} = p .$$

is added automatically in a module KRUNNER, which includes the main module of the definition. The wrapper also sets up the port number for the I/O Server which is then started before Maude is launched and the files containing the modified definition and the rewriting command are loaded.

3.1 The Maude I/O Client

Figure 6 presents the architecture of the Maude component of the I/O interface and briefly describes the interaction between its various sub-components. The user interacts with it using either the basic, console-only interface provided by

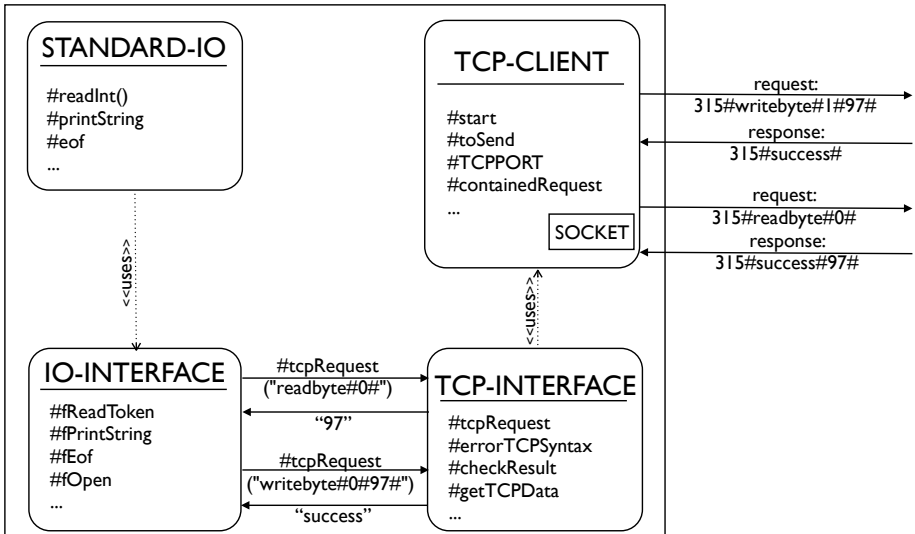


Fig. 6. The architecture of the Maude component of the I/O interface

the `STANDARD-IO` module (see Section 2.1 and 2.2), or the more comprehensive, file-based one provided by the `IO-INTERFACE` module (see Section 2.3). The constructs in the `STANDARD-IO` module desugar into their correspondents from the `IO-INTERFACE` module. The `IO-INTERFACE` module reduces all operations to simple, byte-based requests, and uses the `TCP-INTERFACE` function `#tcpRequest` as an interface to the I/O Server. This result of communication is further interpreted by the functions in the `IO-INTERFACE` module and is transformed into a term of the `IOResult` sort which contains a series of constructors for each specific type of answers, such as `#success`, `#int`, `#string`, and `#eof`.

As most of the I/O interface was exhibited in the previous section, we will focus here on the implementation details: how the high-level constructs are expressed in terms of the lower level ones and how the communication takes place.

The `STANDARD-IO` module defines handles for the standard input, output, and error streams:

```
ops #stdin #stdout #stderr : → Nat .
eq #stdin = 0 .      eq #stdout = 1 .      eq #stderr = 2 .
```

These handles are then used to express the `STANDARD-IO` constructs in terms of those from the `IO-INTERFACE`:

```
eq #eof() = (#fEof(#stdin)) .
eq #readChar() = (#fReadChar(#stdin)) .
eq #printChar(C) = #fPrintChar(#stdout,C) .

eq #readToken() = (#fReadToken(#stdin)) .
eq #readInt() = #fReadInt(#stdin) .
eq #printString(S) = #fPrintString(#stdout,S) .
```

The `IO-INTERFACE` module provides functionality for lower level I/O constructs falling the following three categories. The first category consists of I/O primitives whose semantics is simply a request to the server:

```
eq #open(S) = #handle(rat(#tcpRequest("open#" + S + "#"), 10)) .
eq #close(N) = #checkSuccess(#tcpRequest("close#" + string(N,10) + "#")) .
eq #fReadByte(N)
  = #byte(rat(#tcpRequest("readbyte#" + string(N,10) + "#"),10)) .
ceq #fPutByte(N,B)
  = #checkSuccess(#tcpRequest(
    "writebyte#" + string(N,10) + "#" + string(B,10) + "#"))
if B < 256 .
```

These functions rely on functions like `#checkSuccess` to translate the string answer provided by `#tcpRequest` into a term of the appropriate `IOResult` type. Note that these TCP requests have a very regular form, of `#`-separated pieces of information, among which the first is the command, and the following are additional arguments to the command, like, e.g., “`open#file:in.txt#r#`”.

The second category includes functions that easily desugar into primitives, e.g.:

```

eq #fPrintChar(N,C) = #fPutByte(N,ascii(C)) .
eq #fReadChar(N) = #char(#fReadByte(N)) .
eq #fReadInt(N) = #int(#fReadToken(N)) .

```

```

op #char : IOResult → IOResult .           op #int : IOResult → IOResult .
eq #char(#byte(B)) = #char(char(B)) .       eq #int(#string(S)) = #int(rat(S,10)) .
eq #char(#eof) = #eof .                     eq #int(#eof) = #eof .

```

Finally, there are the more involved functions like `#fPrintString` which iterates over the characters of a string one by one, waiting for the printing of a character to succeed before printing the next and flushing the output buffer at the end,

```

eq #fPrintString(N,"") = #flush(N) .
eq #fPrintString(N,S)
  = #fPrintString(N,S, #fPrintChar(N,substr(S,0,1))) [owise] .
op #fPrintString : Nat String IOResult → IOResult .
eq #fPrintString(N,S,#success) = #fPrintString(N,substr(S,1,length(S))) .

```

or like `#fReadToken` which reads character by character from the file specified by the handle, skipping over the initial whitespace and then accumulating the characters read until whitespace is again encountered.

The *TCP-INTERFACE* module provides functionality for initializing the communication process and extracting the relevant data once the communication process concludes with a result. The semantics of the `#tcpRequest` construct is:

```

eq #tcpRequest(S:String) = #tcpRequest(S:String, counter) .

```

```

op #tcpRequest : String [Nat] → String .
eq #tcpRequest(S:String, N:Nat)
  = #checkResult(#containedRequest(#start(N:Nat) #toSend(S:String))) .

```

`#tcpRequest` creates a object configuration, wrapped by the `#containedRequest` construct; this configuration includes a primitive to start the communication with the server and the request to be sent. `#checkResult` expects the result to be either of the form `success#data###` or of the form `fail#reason###` and returns “data” in case of success or an `#errorTCPSyntax` term otherwise.

The *TCP-CLIENT* module provides rules for initiating the TCP communication through a socket, sending a message, waiting for a response, and closing the connection. First, a fresh id is generated using Maude’s builtin COUNTER module, and is used to establish a connection with the I/O server using the TCP sockets interface provided by Maude:

```

op #start : → Configuration .           eq #start = #start(counter) .
op #start : Nat → Configuration .       op cnum : Nat → Oid .

```

```

rl #start(N)
⇒ <> < cnum(N) : Client | state: connecting >
  createClientTcpSocket(socketManager, cnum(N), "localhost", #TCPPOINT) .

```

The `#TCPPORT` constant is the one mentioned in Section 2.4, being either set manually if running the I/O server separately from Maude, or automatically by the Java wrapper.

Once the initial exchange of messages takes place between Maude and the server, the `#toSend` message is transformed into a message addressed to the server and the state of the system becomes `sending`:

```

rl < cnum(N) : Client | state: connected,  connectedTo: Server,  A > #toSend(S)
⇒ < cnum(N) : Client | state: sending,  connectedTo: Server,  A >
   send(Server, cnum(N), (string(N:Nat,10) + "#" + S + "\r\n")) .

```

Note that the body of the message is prefixed with the number of the client (for logging and debugging reasons) and is appended with the end-of-line markers as a message separator. Thus, the complete message being sent to the server is of the form “23#open#file:in.txt#r#\r\n”.

There are additional rules for continuing the dialogue with the server following the TCP protocol, but once the communication has finished, the answer is extracted by the rule below and the header of the message (containing the number identifying the communication) is removed by the `#checkAnswer` function:

```

rl #containedRequest(<< < Me : Client | state: finished,  answer: S,  A >)
⇒ #checkAnswer(Me, S) .

```

3.2 The Java I/O Server

In this section we discuss the I/O Server architecture and implementation details. While describing the main components, we will motivate our choices regarding their design. The purpose of the I/O Server is to implement a socket-based service for simulating file operations over regular files, standard input, and standard output. So far it has only been used with Maude as a client, but the implementation is rather client-independent.

The architecture and data flow of the I/O Server is depicted in Figure 7. The server has two main components: the communication component, which is responsible for receiving/sending messages, and the resource management component, which manages resources (files, stdin, stdout, stderr) and operations on them. In this section, we will refer to them as RequestManager and ResourceManager.

Before describing these components, we will briefly explain the behavior and capabilities of the I/O Server. First, the complete list of operations currently accepted by the server is the following:

- *open* - open a new file
- *close* - close a file or stream
- *readbyte* - read a byte
- *writebyte* - write a byte
- *flush* - flush the buffer
- *reopen* - reopen an existent file
- *seek* - seek a particular location
- *position* - go to a specific location
- *peek* - peek a byte
- *eof* - check end of file

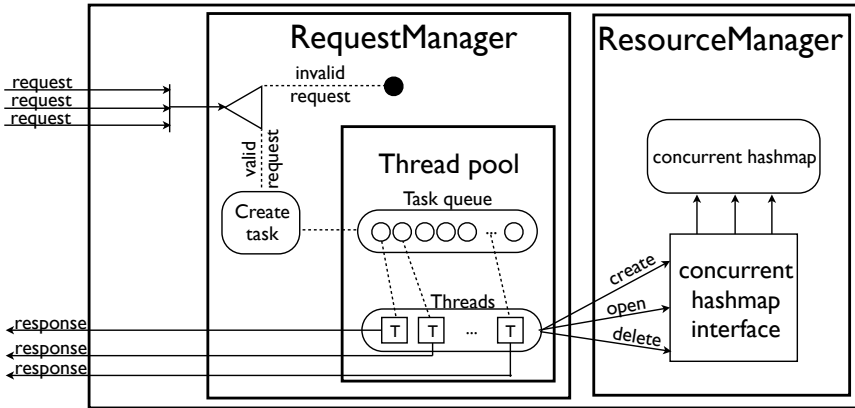


Fig. 7. The data-flow scheme of the I/O server

The server receives requests, each request carrying one of the commands above, and it answers the requests upon executing the corresponding operation. The requests and answers are formatted as a string which contains data separated by “#”:

```
request: 315#writebyte#1#97#
response: 315#success#
```

The request contains the client id, the operation name followed by the resource id and operation parameters, if any. For instance in the above request the client having the id 315 asks the server to write at the standard out stream the byte corresponding to character “a” (the stdin, stdout and stderr streams have fixed ids 0, 1, and 2, respectively). The I/O Server generates a fresh id (identifying the file handle) as response to the client which requests to “open” a new file. The response usually contains the client id which stands for a weak kind of authentication, the status of the operation and the result of its execution if exists.

The RequestManager component uses TCP sockets to provide a reliable point-to-point communication with the client. To be able to handle multiple request concurrently, we use the *Thread Pool* pattern. For each request, the associated command is analyzed and, if found valid, a task is created and queued for execution. Commands are executed in parallel, each thread being responsible for executing a command and sending the response to the client. The thread pool executor (defined in the `ThreadPoolExecutor` class) registers commands as members of the abstract class `Command` which implements the Java `Runnable` interface, and assigns them to threads as these become available.

The second component of the I/O Server handles resources and operations on them. Currently, the ResourceManager can store three types of resources: random access files, standard input, and standard output. It provides operations to add, retrieve, or delete a resource. Some operations, for instance *peek*, *read-byte*, *seek*, and *position* cannot be applied on the standard output; in such cases,

when operations are not applicable on a specific type of resource, the Resource-Manager throws exceptions to the RequestManager, which in turn will send to the client a meaningful failure message.

Regarding the implementation, for each resource we have a corresponding class (`ResourceInFile`, `ResourceOutFile`, ...) which must extend the abstract class `Resource`. This class contains abstract methods which correspond to commands received by the I/O Server (`readbyte()`, `writebyte()`, ...). To store the resources we use the `ConcurrentHashMap` class which has two highly concurrent properties: writing to it locks only a portion of the map and reads can generally occur without locking.

4 Related Work

This paper uses a technology similar to that used for developing Mobile Maude [7], but with a different aim: there sockets communication was used to communicate between different instances of Maude; here we use a similar mechanism to communicate with an external I/O server.

The motivation for this work came from our research in programming language design, namely in our efforts to make the implementation of the \mathbb{K} semantic framework [116] easier to use and experiment with. \mathbb{K} [16] is a rewrite-based executable semantic framework specialized for defining programming languages, type systems and formal analysis tools. The \mathbb{K} tool [118] transforms \mathbb{K} definitions into rewriting logic theories which can be executed, explored and analyzed using Maude. So far the \mathbb{K} tool has been used to give complete definitions to real languages like C [8] and Scheme [10], along with many educational languages and a novel rewriting-based program verification logic named matching logic [15][14]. The I/O interface described in this paper is an integral part of the \mathbb{K} tool and provides I/O capabilities for all \mathbb{K} semantics defined using the tool. Most notably, it has been used to extensively test the \mathbb{K} definition of C [8] against the gcc torture tests [9].

5 Conclusions and Future Work

We have described a methodology and a system for achieving interactive (file) input/output from within the Maude system. This technology was designed for modeling, in an executable way, runtime interaction needed in a system. It can additionally be used for runtime logging or tracing, and provides an easy way of getting output from a Maude execution. The I/O interface is generic and can easily be used in potentially any Maude definition. The interface itself is reasonably stable, as it has been implemented and extensively used as part of the \mathbb{K} framework.

This work could serve as a means for experimenting with I/O before the technology is integrated in Maude directly as a special purpose external object. As our interface uses URIs, it should be relatively easy to incorporate support for accessing additional resources such as URLs. Moreover, adding primitives for locking resources would offer a thread-safe mechanism of accessing the resources.

Acknowledgments. The research presented in this paper was supported in part by the DAK project Contract 161/15.06.2010, SMIS-CSNR 602-12516.

References

1. The K semantic framework website (2010), <https://k-framework.googlecode.com>
2. The Java I/O server: svn repository with full sources (2011), <https://k-framework.googlecode.com/svn/trunk/core/java>
3. The Maude I/O interface: svn repository with specification and examples (2011), <https://k-framework.googlecode.com/svn/branches/inProgress/core/io>
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual (Version 2.6) (January 2011), <http://maude.cs.uiuc.edu/maude2-manual/>
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)
6. Coan, J.S.: Basic BASIC: an introduction to computer programming in BASIC language. Hayden Book Co., Rochelle Park (1978)
7. Durán, F., Riesco, A., Verdejo, A.: A distributed implementation of Mobile Maude. In: WRLA 2006. ENTCS, vol. 176(4), pp. 113–131 (2007)
8. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. In: POPL 2012. ACM (to appear, 2012)
9. FSF: C language test suites: C-torture version 4.4.2 (2010), <http://gcc.gnu.org/onlinedocs/gccint/C-Tests.html>
10. Meredith, P., Hills, M., Rosu, G.: An executable rewriting logic semantics of K-Scheme. In: SCHEME 2007, pp. 91–103 (2007)
11. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
12. Meseguer, J., Braga, C.: Modular Rewriting Semantics of Programming Languages. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 364–378. Springer, Heidelberg (2004)
13. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: POPL 1993, pp. 71–84 (1993)
14. Rosu, G., Ștefănescu, A.: Matching logic: A new program verification approach. In: ICSE 2011 (NIER Track), pp. 868–871 (2011)
15. Rosu, G., Ellison, C., Schulte, W.: Matching Logic: An Alternative to Hoare/Floyd Logic. In: Johnson, M., Pavlovic, D. (eds.) AMAST 2010. LNCS, vol. 6486, pp. 142–162. Springer, Heidelberg (2011)
16. Rosu, G., Serbănută, T.F.: An overview of the K semantic framework. J. of Logic and Algebraic Programming 79(6), 397–434 (2010)
17. Serbănută, T.F., Rosu, G., Meseguer, J.: A rewriting logic approach to operational semantics. Information and Computation 207, 305–340 (2009)
18. Serbănută, T.F., Rosu, G.: K-Maude: A Rewriting Based Tool for Semantics of Programming Languages. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 104–122. Springer, Heidelberg (2010)

Model Checking LTLR Formulas under Localized Fairness

Kyungmin Bae and José Meseguer

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

Abstract. Many temporal logic properties of interest involve both state and action predicates and only hold under suitable fairness assumptions. Temporal logics supporting both state and action predicates such as the Temporal Logic of Rewriting (TLR) can be used to express both the desired properties and the fairness assumptions. However, model checking such properties directly can easily become impossible for two reasons: (i) the exponential blowup in generating the Büchi automaton for the implication formula including the fairness assumptions in its condition easily makes such generation unfeasible; and (ii) often the needed fairness assumptions cannot even be expressed as *propositional* temporal logic formulas because they are *parametric*, that is, they correspond to *universally quantified* temporal logic formulas. Such universal quantification is succinctly captured by the notion of *localized fairness*; for example, fairness localized to the parameter o in object fairness conditions. We summarize the foundations and present the language design and implementation of the new Maude LTLR Model Checker under localized fairness. This is the first tool we are aware of which can model check temporal logic properties under parametric fairness assumptions.

1 Introduction

Many temporal logic properties of interest involve both state and action predicates and only hold under suitable fairness assumptions. For example, the effective transmission of data by a fault-tolerant network protocol can only be proved under the assumption that the receiving node will receive messages infinitely often if it is infinitely often enabled to receive them. Although in principle temporal logics supporting both state and action predicates such as the Temporal Logic of Rewriting (TLR) can be used to express both the desired properties and the fairness assumptions, in practice model checking *directly* such properties can easily become impossible for two reasons. First of all, the exponential blowup in generating the Büchi automaton for the formula $\psi \rightarrow \varphi$, where φ is the desired property and ψ specifies the fairness assumptions, easily makes such generation unfeasible. To address this problem, various techniques to build in the fairness assumptions ψ into the model checking algorithm, so that only φ has to be model checked, have been proposed as the basis of various model

checkers [12,14,18]. However, a second serious difficulty not addressed by those techniques and tools exists: often the needed fairness assumptions ψ cannot even be expressed as *propositional* temporal logic formulas because they are *parametric*, that is, they correspond to *universally quantified* temporal logic formulas which are outside the scope of current fairness-supporting model checkers.

A good example of parametric fairness is provided by *object fairness* assumptions such as “each object o infinitely often enabled to receive a message of the form $m(o, q)$ will receive it infinitely often,” which is universally quantified over all the (possibly dynamically changing) objects o in the system. In rewriting logic such message reception can be expressed by a rewrite rule of the form:

$$[rec] : [o \mid s] m(o, k) \rightarrow [o \mid f(s, k)]$$

where $f(s, k)$ denotes a new state after receiving a message. This object fairness assumption can be described as the universally quantified LTLR formula:

$$(\forall o) \text{ enabled.rec}(o) \rightarrow \text{rec}(o)$$

where $\text{enabled.rec}(o)$ is the obvious state predicate holding iff the *rec* rule is enabled for object o . Such universal quantification can be succinctly captured by the notion of *localized fairness* [15]. The idea is that a rewrite rule like the one above is itself universally quantified over a finite set of variables; for example, for the *rec* rule the set $\{o, s, k\}$. Then the above strong object fairness condition corresponds to localizing the strong fairness requirement for rule *rec* to the singleton subset $\{o\}$. In general, fairness assumptions for a given rule can be localized to any chosen *subset* of its variables.

Is it possible at all to model check temporal logic properties under such parametric fairness assumptions? The question is nontrivial, because, even under the finite-state assumption which can make the number of actual instances of the universal quantification finite, it may be impossible to have a priori knowledge about the actual number of such instances. For example, we may be dealing with an object-based system where objects are dynamically created, so that the entire state space may first have to be searched to determine which objects exist. We have recently reported on the automata-theoretic and algorithmic foundations of a novel model checking algorithm which solves the problem of model checking LTL properties under parameterized fairness and has good performance in practice [2]. However, the work in [2] dealt with this problem at an automata-theoretic level and did not present a suitable *property specification language* in which such parameterized fairness assumptions could be naturally expressed.

In this paper we show that the intrinsically parametric nature of rewrite rules and the great flexibility of the Linear Temporal Logic of Rewriting (LTLR) to express parametric action patterns based on such rules makes LTLR an ideal property specification language for a model checker under parameterized fairness assumptions. In this way, the algorithm presented in [2] becomes available for rewriting logic specifications. Furthermore, we present and illustrate with examples a new LTLR model checker under localized fairness assumptions for Maude, which implements the algorithm in [2] at the C++ level as an extension

of the Maude system. A nontrivial part of this model checker is its user interface. First of all, a simple way for users to specify localized fairness assumptions as rule annotations is provided. Secondly, since LTLR formulas involve spatial action patterns which in turn involve rule labels, LTLR formulas go beyond the syntax available to Maude from parsing the system module to be model checked, or even such module syntax extended by the state predicates and the temporal logic connectives. We explain how rule annotations can also be used for this purpose. Of course, *reflection* techniques and the Full Maude infrastructure are used in an essential way to obtain this expressive and user-extensible user interface. The new Maude LTLR model checker is the first tool we are aware of which can model check temporal logic properties under parametric fairness assumptions. The tool and a collection of examples can be accessed at <http://maude.cs.uiuc.edu/tools/tlr>.

2 Preliminaries on Localized Fairness

This section recalls preliminary notions on rewrite theories, the linear temporal logic of rewriting (LTLR), and localized fairness specifications. We also summarize our previous works on the LTLR model checking algorithm [1] and the LTL model checking algorithm under parameterized fairness assumptions [2], which provide the basis for the new LTLR model checker described in this paper.

2.1 Rewrite Theories

A rewrite theory [4] is a triple $\mathcal{R} = (\Sigma, E, R)$ such that:

- (Σ, E) is a theory in *membership equational logic* with Σ a signature, E a set of *conditional* equations and memberships, and
- R is a set of (possibly conditional) *rewrite rules* written $l : q \rightarrow r$, where l is a *label*, and q and r are Σ -terms.

The state space with a chosen type k is specified as the k -component of the initial algebra $T_{\Sigma/E,k}$, i.e., each state is an E -equivalence class $[t]_E$ of ground terms with type k . Each rule $l : q \rightarrow r$ specifies the system's concurrent transitions. A *one-step rewrite* from a state $[t[\theta q]]_E$ containing a substitution instance θq to the state $[t[\theta r]]_E$ in which θq has been replaced by θr is denoted by:

$$[t[l(\theta)]]_E : [t[\theta q]]_E \rightarrow_{\mathcal{R}} [t[\theta r]]_E$$

where $[t[l(\theta)]]_E$ is called a *one-step proof term*. A *computation* (π, γ) of \mathcal{R} is a path $\pi(0) \xrightarrow{\gamma(0)} \pi(1) \xrightarrow{\gamma(1)} \pi(2) \xrightarrow{\gamma(2)} \dots$ where $\pi(i) = [t_i]_E$ with the state type k , $\pi(i) \xrightarrow{\gamma(i)} \pi(i+1)$ is a one-step rewrite with a one-step proof term $\gamma(i)$ for each $i \in \mathbb{N}$. $(\pi, \alpha)^i$ denotes the suffix of (π, α) beginning at position $i \in \mathbb{N}$, i.e., $(\pi, \alpha)^i = (\pi \circ s^i, \alpha \circ s^i)$ with s the successor function. Any finite computation of \mathcal{R} can be extended into an infinite computation if \mathcal{R} has no deadlock states. Any rewrite theory whose rules have no rewrites in their conditions can be transformed into a semantically an equivalent deadlock-free theory [17]. We will assume from now on that a rewrite theory is deadlock free.

2.2 The Linear Temporal Logic of Rewriting

The *linear temporal logic of rewriting* (LTLR) is a state/event extension of linear temporal logic with *spatial action patterns* that describe properties of one-step rewrites [1]. The only syntactic difference between LTLR and LTL is that an LTLR formula may include spatial action patterns $\delta_1, \dots, \delta_n$ as well as state propositions p_1, \dots, p_m , and therefore can describe properties involving both states and events, e.g., fairness conditions. Given a set of state propositions Π and a set of spatial action patterns W , the syntax of LTLR formulas is as follows:

$$\varphi ::= p \mid \delta \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \bigcirc\varphi \mid \varphi\mathcal{U}\varphi'$$

where $p \in \Pi$ and $\delta \in W$. Other operators can be defined by equivalences, e.g., $\diamond\varphi \equiv \text{true}\mathcal{U}\varphi$, and $\square\varphi \equiv \neg\diamond\neg\varphi$.

The semantics of LTLR over a set of state propositions Π and a set of spatial action patterns W is defined on a rewrite theory \mathcal{R} that contains a subtheory for Π , W , boolean values, and one-step proof terms. A state proposition (resp., a spatial action pattern) is defined by a parametric function symbol of the form $p : s_1 \dots s_n \rightarrow \mathbf{Prop}$ (resp., $\delta : s_1 \dots s_m \rightarrow \mathbf{Action}$). The satisfaction relations for state propositions and spatial action patterns are defined by means of equations using the following auxiliary operators:

$$_ \models _ : k \mathbf{Prop} \rightarrow \mathbf{Bool} \quad _ \models _ : \mathbf{ProofTerm} \mathbf{Action} \rightarrow \mathbf{Bool}$$

in which a state proposition p is satisfied on a state $[t]_E$ if and only if $E \vdash (t \models p) = \text{true}$, and a spatial action pattern δ is satisfied on a one-step proof term $[\lambda]_E$ if and only if $E \vdash (\lambda \models \delta) = \text{true}$. An LTLR formula φ is satisfied on \mathcal{R} from an initial state $[t]_E$, denoted by $\mathcal{R}, [t]_E \models \varphi$, if and only if for each infinite computation (π, γ) starting from $[t]_E$, the path satisfaction relation $\mathcal{R}, (\pi, \gamma) \models \varphi$ holds, which is defined inductively as follows:

- $\mathcal{R}, (\pi, \gamma) \models p$ iff $E \vdash (\pi(0) \models p) = \text{true}$
- $\mathcal{R}, (\pi, \gamma) \models \delta$ iff $E \vdash (\gamma(0) \models \delta) = \text{true}$
- $\mathcal{R}, (\pi, \gamma) \models \neg\varphi$ iff $\mathcal{R}, (\pi, \gamma) \not\models \varphi$
- $\mathcal{R}, (\pi, \gamma) \models \varphi \wedge \varphi'$ iff $\mathcal{R}, (\pi, \gamma) \models \varphi$ and $\mathcal{R}, (\pi, \gamma) \models \varphi'$
- $\mathcal{R}, (\pi, \gamma) \models \bigcirc\varphi$ iff $\mathcal{R}, (\pi, \gamma)^1 \models \varphi$
- $\mathcal{R}, (\pi, \gamma) \models \varphi\mathcal{U}\varphi'$ iff $\exists j \geq 0. \mathcal{R}, (\pi, \gamma)^j \models \varphi', \forall 0 \leq i < j. \mathcal{R}, (\pi, \gamma)^i \models \varphi$.

2.3 Localized Fairness

Fairness of a rewrite theory \mathcal{R} is often expressed by patterns of rewrite events, i.e., by spatial action patterns. A one-step rewrite event is usually too specific to describe a general fairness requirement.

Definition 1. A basic action pattern of a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is a simple parametric spatial action pattern of the form $l(\bar{y})$ such that l is the label of some rule $l : q \rightarrow r \in R$ and $\bar{y} \subseteq \text{vars}(q)$. A ground instance $\theta(l(\bar{y}))$ of a basic action pattern $l(\bar{y})$ is satisfied on each one-step proof term of the form $[t[l(\theta)]]_E$.

A basic action pattern $l(\emptyset)$ with no variables is denoted by just a rule label l . Also, a parametric state proposition $enabled(l(\bar{y}))$ can be defined in \mathcal{R} such that a ground instance $\theta(enabled(l(\bar{y})))$ is satisfied on each state $[t]_E$ from which there exists a one-step rewrite $[t[l(\theta)]]_E$. The satisfaction relations for both basic action patterns and enabled propositions can be defined by equations and automatically generated from \mathcal{R} (see Section 4.3).

A *localized fairness* specification [15] is a pair of finite sets $(\mathcal{J}, \mathcal{F})$ whose elements are basic action patterns of the general form $l(\bar{y})$. The set \mathcal{J} stands for weak fairness conditions¹ and \mathcal{F} stands for strong fairness conditions². This localized fairness specification is quite general so that many different notions of fairness, such as object/process fairness, can be expressed in a unified way [15]. Intuitively, localized fairness given by $l(\bar{y}) \in \mathcal{J} \cup \mathcal{F}$ means that for each ground instance $\vartheta(l(\bar{y}))$ of $l(\bar{y})$, the corresponding one-step rewrite satisfies the desired weak or strong fairness requirements. Each localized fairness pattern can be expressed by an equivalent *universally quantified* LTLR formula [2] of the form $\forall \bar{y} \varphi$, where φ is quantifier-free, $vars(\varphi) \subseteq \bar{y}$, and:

$$\mathcal{R}, (\pi, \gamma) \models \forall \bar{y} \varphi \quad \Leftrightarrow \quad \mathcal{R}, (\pi, \gamma) \models \theta \varphi \text{ for each ground substitution } \theta$$

A weak (resp. strong) fairness condition with respect to a basic action pattern $l(\bar{y})$ is then expressed by the quantified LTLR formula $\forall \bar{y} \diamond \square enabled(l(\bar{y})) \rightarrow \square \diamond l(\bar{y})$ (resp., $\forall \bar{y} \square \diamond enabled(l(\bar{y})) \rightarrow \square \diamond l(\bar{y})$).

A localized fairness specification $(\mathcal{J}, \mathcal{F})$ defines a set of fair computation of a rewrite theory \mathcal{R} . An infinite computation (π, γ) of \mathcal{R} is \mathcal{J}, \mathcal{F} -fair if and only if every localized fairness condition in $\mathcal{J} \cup \mathcal{F}$ is satisfied on (π, γ) in \mathcal{R} . That is,

- $\mathcal{R}, (\pi, \gamma) \models \forall \bar{y}_j \diamond \square enabled(l_j(\bar{y}_j)) \rightarrow \square \diamond l_j(\bar{y}_j)$, for each $l_j(\bar{y}_j) \in \mathcal{J}$, and
- $\mathcal{R}, (\pi, \gamma) \models \forall \bar{y}_f \square \diamond enabled(l_f(\bar{y}_f)) \rightarrow \square \diamond l_f(\bar{y}_f)$, for each $l_f(\bar{y}_f) \in \mathcal{F}$.

An LTLR formula φ is then *fairly* satisfied on \mathcal{R} from an initial state $[t]_E$ under $(\mathcal{J}, \mathcal{F})$, denoted by $\mathcal{R}, [t]_E \models_{\mathcal{J} \cup \mathcal{F}} \varphi$, if and only if $\mathcal{R}, (\pi, \gamma) \models \varphi$ holds for each \mathcal{J}, \mathcal{F} -fair computation (π, γ) starting from the initial state $[t]_E$.

2.4 Model Checking Algorithms

The model checking problem for an LTLR formula φ on a rewrite theory \mathcal{R} can be characterized by automata-theoretic techniques on the associated *labeled Kripke structure* (LKS) using the Büchi automaton $\mathcal{B}_{-\varphi}$ [13].

Definition 2. An LKS \mathcal{K} is a 6-tuple $(S, S_0, \Pi, \mathcal{L}, W, T)$, where S is a set of states, $S_0 \subseteq S$ is a set of initial states, Π is a set of state propositions, $\mathcal{L} : S \rightarrow \mathcal{P}(\Pi)$ is a state-labeling function, W is a set of events (i.e., spatial action patterns), and $T \subseteq S \times \mathcal{P}(W) \times S$ is a labeled transition relation.

¹ If an event is continuously enabled beyond a certain point, it is taken infinitely often.

² If an event is enabled infinitely often, then it is taken infinitely often.

A *path* (π, α) of \mathcal{K} is an infinite sequence $\langle \pi(0), \alpha(0), \pi(1), \alpha(1), \dots \rangle$ such that $\pi(i) \in S$, $\alpha(i) \subseteq W$, and $\pi(i) \xrightarrow{\alpha(i)} \pi(i+1)$ for each $i \geq 0$. If \mathcal{R} is a *computable* rewrite theory that satisfies additional decidability conditions [16], given an initial state $[t]_E$, a set of state propositions Π , and a set of spatial action patterns W , we can construct the corresponding LKS $\mathcal{K}_{\Pi, W}(\mathcal{R})_t$ such that $\mathcal{R}, [t]_E \models \varphi$ if and only if $\mathcal{K}_{\Pi, W}(\mathcal{R})_t \models \varphi$ for any LTLR formula φ over Π and W [11]. Therefore, a formula φ has no counterexample on \mathcal{R} from an initial state $[t]_E$ if and only if the product automaton $\mathcal{K}_{\Pi, W}(\mathcal{R})_t \times \mathcal{B}_{\neg\varphi}$ has no accepting path, which can be easily checked using the nested depth first search algorithm [10].

On the other hand, the model checking algorithm for a universally quantified LTLR formula $\forall \bar{x} \varphi$ on a rewrite theory \mathcal{R} is nontrivial, since in general, such a variable quantification ranges over an infinite set, e.g., a set of tuples of ground terms having specified sorts in \mathcal{R} . We cannot directly use the corresponding LKS $\mathcal{K}_{\Pi, W}(\mathcal{R})_t$ for model checking such universally quantified LTLR formulas. However, the satisfaction relation for $\forall \bar{x} \varphi$ can be efficiently determined on a finite LKS satisfying *finite instantiation property* (FIP) [2].

Definition 3. An LKS $\mathcal{K} = (S, S_0, \Pi, \mathcal{L}, W, T)$ satisfies a finite instantiation property (FIP) if and only if: (i) for each state $s \in S$, $\mathcal{L}(s)$ is finite, and (ii) for each transition $s \xrightarrow{A} s' \in T$, the set A is finite³

The *path-realized* set $\mathcal{R}_{(\pi, \alpha), \varphi}$ is a set of substitutions that is guaranteed to be finite if the underlying LKS \mathcal{K} is *finite* and satisfies FIP (see [2] for a detailed definition). Only such a finite path-realized set is necessary to decide the satisfaction of a universally quantified formula $\forall \bar{x} \varphi$ on \mathcal{K} as shown in the following lemma [2], in which $\mathcal{K}_\perp = (S, S_0, \Pi \cup \Pi_\perp, \mathcal{L}, W \cup W_\perp, T)$ is an extension of \mathcal{K} such that \mathcal{K}_\perp may have additional *void* state propositions Π_\perp and spatial action patterns W_\perp which are never satisfied on \mathcal{K}_\perp .

Lemma 1. Given a finite LKS \mathcal{K} satisfying FIP, a universally quantified LTLR formula $\forall \bar{x} \varphi$, and a path (π, α) , for each ground substitution θ , there exists $\vartheta \in \mathcal{R}_{(\pi, \alpha), \varphi}$ such that $\mathcal{K}, (\pi, \alpha) \models \theta\varphi$ iff $\mathcal{K}_\perp, (\pi, \alpha) \models \vartheta\varphi$.

For a finite LKS \mathcal{K} satisfying FIP, we can have an efficient algorithm to model check an LTLR formula φ under fairness assumptions of the form $\forall \bar{y} \diamond \Box \Phi \rightarrow \Box \diamond \Psi$ or $\forall \bar{y} \Box \diamond \Phi \rightarrow \Box \diamond \Psi$, where Φ and Ψ are boolean formulas with no temporal operators [2]. The model checking algorithm consists basically in finding a reachable strongly connected component (SCC) \mathfrak{S} from initial states in the product automaton $\mathcal{K} \times \mathcal{B}_{\neg\varphi}$ such that: (i) \mathfrak{S} satisfies an acceptance condition of $\mathcal{B}_{\neg\varphi}$, and (ii) all realized substitution instances of fairness formulas hold in \mathfrak{S} . The SCC \mathfrak{S} satisfies the universally quantified fairness formulas, thanks to Lemma 1. Such an SCC \mathfrak{S} exists in $\mathcal{K} \times \mathcal{B}_{\neg\varphi}$ if and only if there is a fair counterexample of φ in \mathcal{K} . The complexity of the algorithm is $O(k \cdot f \cdot |\mathcal{K}| \cdot 2^{|\varphi|})$, where k is the number of fairness formulas and f is the number of realized substitutions [2].

³ More precisely, there are only finitely many ground instances for each parametric proposition. But if the signature is finite as usual, the definitions are equivalent.

3 The Maude LTLR Model Checker under Fairness

We have developed a new Maude LTLR model checker to support model checking under localized fairness specifications using the algorithm in [2]. This tool extends the previous LTLR model checker [1] and the existing LTL model checker [8] in Maude. The new LTLR model checker allows the user to specify localized fairness conditions for each rule in a system module, and provides a simple user interface to perform model checking under localized fairness assumptions. The earlier version of the LTLR model checker [1] did not support localized fairness assumptions, and before this work the model checking algorithm in [2] had not been integrated with a suitable *property specification language* in which such parameterized fairness assumptions could be naturally expressed. This section presents rewrite theories and LTLR as an ideal property specification language for a parameterized fairness assumptions with user-friendly tool support.

Throughout this section, we will use a simple fault-tolerant client-server communication model borrowed from [16] to illustrate the new LTLR model checker under localized fairness specifications. In this model, each client C sends a query N to a server S to receive an answer, and the server returns the answer $f(S, C, N)$ of the query using a function f . The configuration of the system is a multiset of clients, servers, and messages, with the empty multiset `null`. A client is represented as a term $[C, S, N, W]$ with C the client's name, S a server's name, N a number representing a query, and W either a number representing an answer or `nil` if the answer has not yet been received. A server is represented as a term $[S]$ with the name S , and a message is represented as a term $I \leftarrow \{J, N\}$ with I the receiver's name, J the sender's name, and N a number. The following rewriting rules define the behavior of the system:

```

r1 [req]   : [C,S,N,nil]                => [C,S,N,nil] S <- {C,N} .
r1 [reply]: S <- {C,N} [S]              => [S] C <- {S,f(S,C,N)} .
r1 [rec]   : C <- {S,M} [C,S,N,W]      => [C,S,N,M] .
r1 [dupl]  : I <- {C,N}                 => I <- {C,N} I <- {C,N} .
r1 [loss]  : I <- {C,N}                 => null .
    
```

This system has an infinite number of states, but we can apply the equational abstraction [17] to collapse the set of states into a finite set by adding the following abstraction equation and coherent completion rule as described in [16]:

```

eq I <- {C,N} I <- {C,N} = I <- {C,N} .
r1 [reply]: S <- {C,N} [S] => S <- {C,N} [S] C <- {S,f(S,C,N)} .
    
```

A liveness property we may wish to verify is the LTLR formula $\Diamond \mathbf{rec}$ with a basic action pattern \mathbf{rec} , which means that some client will eventually receive an answer; however, $\Diamond \mathbf{rec}$ does *not* hold without fairness. The fairness assumptions needed to prove the formula $\Diamond \mathbf{rec}$ are: (i) weak fairness of the rule *req* for each client C , (ii) strong fairness of the rule *reply* for each server S and client C , and (iii) strong fairness of the rule *rec* for each client C . A general fairness specification of this system is nontrivial, because the number of fairness conditions depends on the number of servers and clients in initial configurations.

Furthermore, if the number of clients and servers can be changed during execution, the number of fairness conditions depends on the maximum number of clients and servers during execution. However, such fairness conditions can be naturally expressed as the localized fairness specification:

$$\mathcal{J} = \{\mathbf{req}(\mathbf{C})\} \quad \mathcal{F} = \{\mathbf{reply}(\mathbf{S}, \mathbf{C}), \mathbf{rec}(\mathbf{C})\}$$

no matter how many clients and servers make up the system. It is then easy to show that for any initial state *init* consisting of one or more servers, each with one or more clients connected to it and having *nil* in their fourth component, we have the desired satisfaction $\mathcal{R}, \mathit{init} \models_{\mathcal{J} \cup \mathcal{F}} \diamond \mathbf{rec}$.

3.1 Specification of Localized Fairness

A localized fairness specification $(\mathcal{J}, \mathcal{F})$ of a system module is given by a *meta-data* attribute for each rule, which is a list of fairness items separated by the “;” symbol. Each fairness item for a rule $l : q \rightarrow r$ has one of the following forms:

$$\mathit{just}(x_1, \dots, x_n) \quad \mathit{fair}(x_1, \dots, x_n) \quad l(x_1, \dots, x_n)$$

where $x_1, \dots, x_n \in \mathit{vars}(q)$. A fairness item with no variables is expressed by *just*, *fair*, or *l*. A variable in the left side of a matching condition can also be used in a fairness item. Whenever a fairness item $\mathit{just}(x_1, \dots, x_n)$ (resp., $\mathit{fair}(x_1, \dots, x_n)$) is included in a metadata attribute of a rule with label *l*, the corresponding basic action pattern $l(x_1, \dots, x_n)$ is included in the weak fairness specification \mathcal{J} (resp., the strong fairness specification \mathcal{F}). A fairness item $l(x_1, \dots, x_n)$ in a metadata rule attribute only declares the signature of the basic action pattern $l(x_1, \dots, x_n)$, which may be used in model checking formulas⁴ but not in a localized fairness specification $(\mathcal{J}, \mathcal{F})$. Such a signature is required to parse LTLR formulas containing the basic action pattern in model checking commands.

Each fairness item in metadata attributes determines the signature of the corresponding basic action patterns, which is usually not a part of the original rewrite theory. The user needs to define the necessary basic action pattern signature before executing any model checking command under localized fairness assumptions. Although it is possible to automatically generate all the possible basic action patterns from a rewrite theory, it easily causes confusion and ambiguity on the meanings of different basic action patterns. For example, a rewrite rule $l : f(x_1, x_2) \rightarrow g(x_1)$ in which the variables x_1 and x_2 have the same sort *S* has two ambiguous basic action patterns $l(x_1)$ and $l(x_2)$ that cannot be distinguished by their syntax when applied to a concrete instance, e.g., $l(u)$ with some ground term *u* of sort *S*. In order to avoid such ambiguities, our tool takes the metadata rule attributes into account so that the user can specify the exact basic action patterns they intended to use for model checking purposes.

⁴ $l(x_1, \dots, x_n)$ gives us a very expressive syntax for basic action patterns, since any subset $\{x_1, \dots, x_n\} \subseteq X$ contained in the set of variables *X* of the rule labeled *l* can then be used as a basic action pattern if $l(x_1, \dots, x_n)$ has been declared this way.

In the following client-server communication example introduced above, the metadata attributes of the rules define the localized fairness specification ($\mathcal{J} = \{\text{req}(C)\}$, $\mathcal{F} = \{\text{reply}(S,C), \text{rec}(C)\}$). The fairness item `rec` of the rule `rec` is written in order to declare the basic action pattern, so that the formula $\Diamond \text{rec}$ can be used in the model checking command later. The metadata attributes define the signature of the basic action patterns `req(C)`, `reply(S,C)`, `rec`, and `rec(C)`, and their corresponding enabled propositions such as `enabled(req(C))`.

```

r1 [req]  : [C,S,N,nil]          => [C,S,N,nil] S <- {C,N}
           [metadata "just(C)" ] .
r1 [reply]: S <- {C,N} [S]       => [S] C <- {S,f(S,C,N)}
           [metadata "fair(S,C)"] .
r1 [rec]  : C <- {S,M} [C,S,N,W] => [C,S,N,M]
           [metadata "rec; fair(C)"] .
r1 [dupl] : I <- {C,N}          => I <- {C,N} I <- {C,N} .
r1 [loss] : I <- {C,N}          => null .
    
```

Some ambiguous basic action patterns can still be mistakenly introduced, and should be manually resolved by the user. For example, the ambiguity between `req(C)` and `req(S)` could be removed by making `C` and `S` have different kinds.

3.2 The Fair LTLR Model Checker Interface

The interface of the new Maude LTLR model checker under localized fairness specifications is developed as an extension of Full Maude. Contrary to our previous LTLR model checker [1], in which each spatial action pattern should be manually defined in a similar way to the case of state propositions, the new interface automatically generates the necessary declarations for the basic action patterns from rule attributes (see Section 4.3). If the formulas to be verified only contain basic action patterns written in metadata attributes, then no additional declarations are required. As in [1], the tool also supports more general user-defined spatial action patterns (see Section 3.3), for example, a pattern $l(u_1, \dots, u_n)$ with some of the u_i non-variable terms.

First of all, there is a command `pfmc t |= φ` for model checking an LTLR formula φ with an initial state t under a given localized fairness specification $(\mathcal{J}, \mathcal{F})$. For example, the following is the fair model checking result of the formula $\Diamond \text{rec}$ for the client-server communication example:

```

Maude> (pfmc [a] [b,a,1,nil] [c,a,0,nil] |= <> rec .)
ltlr model check under localized fairness in CLIENT-SERVER-CHECK :
  [a][b,a,1,nil][c,a,0,nil] |= <> rec
result Bool :
  true
    
```

The other command `mc t |= φ` is a usual LTLR model checking command without localized fairness. However, unlike the previous Maude LTLR model checker, basic action patterns are automatically declared for input formulas if the patterns were declared in the rule attribute. For example, the following `mc` command

returns a counterexample, where the server a keeps replying to the client b , but the client b receives no message because $\text{rec}(b)$ does not satisfy the fairness assumptions (parts of the counterexample are replaced by ...):

```
Maude> (mc [a] [b,a,1,nil] [c,a,0,nil] |= <> rec .)
ltlr model check in CLIENT-SERVER-CHECK :
  [a] [b,a,1,nil] [c,a,0,nil] |= <> rec
result ModelCheckResult :
  counterexample(
    {[a] [b,a,1,nil] [c,a,0,nil], {'req : 'C \ b ; 'S \ a}}
    {[a] (a <-b,1) [b,a,1,nil] [c,a,0,nil], {'reply : 'C \ b ; 'S \ a}}
    ...
    {[a] (a <-{b,1}) (a <-{c,0}) (b <-{a,f(a,b,1)}) [b,a,1,nil] [c,a,0,nil],
      {'reply : 'C \ b ; 'S \ a}})
```

A counterexample of an LTLR formula consists of a finite prefix and an infinite cycle in which each item is a pair of a state and a simplified one-step proof term.

Furthermore, each model checking command allows the user to specify additional *ground fairness conditions*, which can be used when some fairness conditions cannot be expressed by a localized fairness specification.⁵ A ground fairness specification is a finite set of ground weak fairness ($\text{just} : \Phi \Rightarrow \Psi$) and ground strong fairness ($\text{fair} : \Phi \Rightarrow \Psi$), where $\text{just} : \Phi \Rightarrow \Psi$ (resp., $\text{fair} : \Phi \Rightarrow \Psi$) is a shorthand for a fairness formula $\diamond \Box \Phi \rightarrow \Box \diamond \Psi$ (resp., $\Box \diamond \Phi \rightarrow \Box \diamond \Psi$). In this case, the formulas Φ and Ψ can be any boolean formulas involving state propositions and spatial action patterns. The following model checking result is for the same example with enough ground fairness conditions to prove $\diamond \text{rec}$:

```
Maude> (mc [a] [b,a,1,nil] [c,a,0,nil] |= <> rec under
  (just : enabled(req(b)) => req(b)) ;
  (fair : enabled(rec(b)) => rec(b)) ;
  (fair : enabled(reply(a,b)) => reply(a,b)) .)
ltlr model check in CLIENT-SERVER-CHECK :
  [a] [b,a,1,nil] [c,a,0,nil] |= <> rec
under fairness :
  (just : enabled(req(b))=> req(b));
  (fair : enabled(rec(b))=> rec(b));
  fair : enabled(reply(a,b))=> reply(a,b)
result Bool :
  true
```

In contrast, the model checking command with only ground weak fairness conditions gives the following counterexample in which no client receives any message since all of them are taken away by the *loss* rule:

⁵ For example, we may have objects a, b, c, d , and e , but we may only want to specify fairness requirements for a, c , and e , but not for b and d . Or we may have fairness requirements $\diamond \Box \Phi \rightarrow \Box \diamond \Psi$ or $\Box \diamond \Phi \rightarrow \Box \diamond \Psi$ where the formulas Φ and Ψ do not correspond to the fairness requirements for a rule application.

```

Maude> (mc [a] [b,a,1,nil] [c,a,0,nil] |= <> rec under
      (just : enabled(req(b)) => req(b));
      (just : enabled(req(c)) => req(c));
      (just : enabled(reply(a,b)) => reply(a,b));
      (just : enabled(reply(a,c)) => reply(a,c));
      (just : enabled(rec(b)) => rec(b));
      just : enabled(rec(c)) => rec(c) .)
ltlr model check in CLIENT-SERVER-CHECK :
  [a] [b,a,1,nil] [c,a,0,nil] |= <> rec
under fairness :
  (just : enabled(req(b))=> req(b)); (just : enabled(req(c))=> req(c));
  (just : enabled(reply(a,b))=> reply(a,b));
  (just : enabled(reply(a,c))=> reply(a,c));
  (just : enabled(rec(b))=> rec(b)); just : enabled(rec(c))=> rec(c)
result ModelCheckResult :
  counterexample(nil,
    {[a] [b,a,1,nil] [c,a,0,nil], {'req : 'C \ b ; 'S \ a}}
    {[a] (a <-{b,1}) [b,a,1,nil] [c,a,0,nil], {'reply : 'C \ b ; 'S \ a}}
    {[a] (b <-{a,f(a,b,1)}) [b,a,1,nil] [c,a,0,nil], {'req : 'C \ c ; 'S \ a}}
    {[a] (a <-{c,0}) (b <-{a,f(a,b,1)}) [b,a,1,nil] [c,a,0,nil], {'loss : 'I \ a}}
    {[a] (b <-{a,f(a,b,1)}) [b,a,1,nil] [c,a,0,nil], {'loss : 'I \ b}})
    
```

Note that, since all the objects in the initial state have been given weak fairness requirements corresponding to rule applications, we can simplify the above complex model checking command using the `pfmc` command and metadata attributes in the style shown above.

3.3 More General Spatial Action Patterns

The Maude LTLR model checker under localized fairness provides capabilities for the user to define spatial action patterns, in a way similar to the equational definition of state propositions, as well as basic action patterns. Recall that the syntax of a spatial action pattern is defined by a parametric function symbol of sort `Action`, and the satisfaction relation of a spatial action pattern is given by equations using the auxiliary operator $_ \models _ : \text{ProofTerm } \text{Action} \rightarrow \text{Bool}$ involving one-step proof terms and spatial action patterns. As a matter of fact, the syntax and the semantics of the basic action patterns given in metadata attributes are also defined in the exact same way, but such definitions are automatically generated by the tool.

The basic signature for model checking is specified in the system module `LTLR-MODEL-CHECKER`, which is inherited from the earlier version of the LTLR model checker [1] but has been extended to support localized fairness specifications. First, sort `BasicActionPattern` for basic action patterns is introduced as a subsort of a spatial action pattern sort `Action`. A one-step proof term $[t[l(\theta)]]_E$ is represented as a triple of a context term $t[\square]$ that has a hole \square inside, a rule label l , and a substitution θ as an assignment set of the form $\mathbf{x}_1 \setminus \mathbf{u}_1 ; \dots ; \mathbf{x}_n \setminus \mathbf{u}_n$, enclosed by the triple operator:

```
op {_|_:_} : StateContext RuleName Substitution -> ProofTerm [ctor ...] .
```

Each spatial action pattern is then declared using the above constructs. For instance, a basic action pattern $l(x_1, \dots, x_n)$ can be defined by:

```
op l : S1 ... Sn -> BasicActionPattern [ctor] .
eq {CONTEXT | 'l : 'x1 \ x1 ; ... ; 'xn \ xn ; SUBST} |= l(x1, ..., xn) = true .
```

where $'l, 'x_1, \dots, 'x_n$ are quoted identifier constants of sort `Qid`, which are used for explicitly expressing variable names.

This mechanism enables the user to define much more general form of spatial action patterns. The following declarations show some predefined spatial action patterns in the module `LTLR-MODEL-CHECKER` whose satisfaction depends on the rewriting positions [16] in addition to the rule labels and the substitutions:

```
op top : BasicActionPattern -> ActionPattern .
op {_|_} : StateContext BasicActionPattern -> ActionPattern .

var C : StateContext . var S : Substitution . var BSP : BasicActionPattern .

eq {[] | R:RuleName : S} |= top(BSP) = {[] | R:RuleName : S} |= BSP .
eq {C | R:RuleName : S} |= {C | BSP} = {C | R:RuleName : S} |= BSP .
```

As defined by the above satisfaction equations, a ground spatial action pattern $top(l(u_1, \dots, u_n))$ holds on a one-step rewrite that happens at the top position and satisfies the basic action pattern $l(u_1, \dots, u_n)$. Similarly, a ground spatial action pattern $\{t[\square] | l(u_1, \dots, u_n)\}$ holds on a one-step rewrite that happens at the position represented by the context term $t[\square]$ and satisfies the basic action pattern $l(u_1, \dots, u_n)$. Such spatial action patterns with context terms are meaningful only if the signature of context terms is given [1]. In the new Full Maude interface, the signature of context terms can be generated by the module expression `CONTEXT [M]` from a system module `M`.

4 The Maude LTLR Model Checker Implementation

The Maude LTLR model checker has been implemented at both the Core Maude and Full Maude levels for the sake of gaining efficiency while keeping expressiveness and user-friendliness. The new LTLR model checker under localized fairness consists of three components: (i) the graph traversal engine that constructs the corresponding LKS from a rewrite theory, (ii) the model checking algorithms under parameterized fairness assumptions using the LKS, and (iii) the user interface of the model checker. For efficiency reasons, the first and second components are implemented at the C++ level within the Maude system. In particular, the model checking algorithm under parameterized fairness requires that the underlying LKS satisfies FIP. But for basic action patterns, the corresponding LKS of a rewrite theory satisfies FIP for free as we show below. Finally, the user interface of the model checker has been implemented by extending Full Maude, since it involves several theory transformations that automate the user interface.

4.1 Localized Fair Model Checking of Rewrite Theories

Basically, model checking an LTLR formula φ under a localized fairness specification $(\mathcal{J}, \mathcal{F})$ is to find an \mathcal{J}, \mathcal{F} -fair counterexample that satisfies $\neg\varphi$. By definition, given a rewrite theory \mathcal{R} and an LTLR formula φ , a \mathcal{J}, \mathcal{F} -fair counterexample (π, γ) invalidating φ should satisfy:

- $\mathcal{R}, (\pi, \gamma) \models \neg\varphi$,
- for each $l_j(\bar{y}_j) \in \mathcal{J}$, $\mathcal{R}, (\pi, \gamma) \models \forall \bar{y}_j \diamond \square \text{enabled}(l_j(\bar{y}_j)) \rightarrow \square \diamond l_j(\bar{y}_j)$, and
- for each $l_f(\bar{y}_f) \in \mathcal{F}$, $\mathcal{R}, (\pi, \gamma) \models \forall \bar{y}_f \square \diamond \text{enabled}(l_f(\bar{y}_f)) \rightarrow \square \diamond l_f(\bar{y}_f)$.

In order to apply the model checking algorithm for parameterized fairness [2] to find such a counterexample, the corresponding LKS \mathcal{K} of \mathcal{R} should be finite and satisfy FIP. Given a computable rewrite theory \mathcal{R} with a finite number of reachable states from an initial state $[t]_E$, we can construct the *finite* LKS $\mathcal{K}_{\Pi, W}(\mathcal{R})_t$ with respect to state propositions Π and spatial action patterns W .

Definition 4. A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is finite-state if and only if E and R are finite, and for each initial state $[t_0]_E \in T_{\Sigma/E, k}$, the set of reachable states $\text{Reach}_{\mathcal{R}}([t_0]_E) = \{[t^*]_E \in T_{\Sigma/E, k} \mid [t_0]_E \rightarrow_{\mathcal{R}}^* [t^*]_E\}$ is always finite.

The only remaining requirement is that the LKS \mathcal{K} satisfies FIP with respect to the state propositions and the spatial action patterns appearing in $(\mathcal{J}, \mathcal{F})$, which have the form of either $\text{enabled}(l(\bar{y}))$ or $l(\bar{y})$. By definition, each ground instance $\theta(\text{enabled}(l(\bar{y})))$ is satisfied on a state $[t]_E$ if and only if there exists a one-step rewrite $[t[l(\theta)]]_E$. Since a finite-state rewrite theory has only finitely many one-step rewrites for each state, each state $[t]_E$ of \mathcal{R} satisfies only finitely many ground instances of $\text{enabled}(l(\bar{y}))$. Similarly, since each ground instance $[\theta(l(\bar{y}))]_E$ is satisfied on a one-step proof term $[t[l(\theta)]]_E$, each one-step rewrite of \mathcal{R} satisfies only one ground instance of $l(\bar{y})$. Therefore, the associated LKS of a finite-state rewrite theory \mathcal{R} always satisfies FIP with respect to a localized fairness specification $(\mathcal{J}, \mathcal{F})$.

4.2 The Model Checking Algorithm Implementation

The new LTLR model checker under localized fairness implements the parametric generalized fairness algorithm [2] in C++ on top of the previous LTLR model checking algorithm, which constructs state/event-based product automaton between a system LKS and a formula Büchi automaton [1]. For generating a Büchi automaton, it reuses the existing LTL model checker implementation [8]. Besides dealing with fairness, the new model checker also generates shorter counterexamples than the previous model checkers in Maude. When a counterexample is found, we perform a breadth-first search from loop states to the initial states using only *already visited* states to find the shortest prefix in the explored state space. The performance of the new Maude LTLR model checker is comparable to other explicit-state model checkers such as SPIN [11] and PAT [18] as shown in [2], and it is currently the only tool we know supporting *parameterized* fairness.

Different model checking algorithms are used by the tool for handling different cases of input fairness, because the general algorithms are more computationally expensive. For usual LTLR model checking with no fairness requirements, we use the nested depth first search algorithm of [10] as the previous LTLR model checker. If only weak fairness conditions are specified, we use the SCC-based algorithm [5] for generalized Büchi automata, in which weak fairness conditions are directly incorporated as an acceptance condition. In the case of strong fairness conditions, the Streett automata emptiness checking algorithm is employed as explained in [2]. If some of the fairness conditions are given by a localized fairness specification, such a fairness model checking algorithm is combined with the parametric fairness algorithm that computes realized substitutions.

4.3 Theory Extension for Localized Fairness

In order to simplify the user interface, the model checker uses theory transformations to automatically generate each basic action pattern $l(\bar{y})$ and its corresponding state proposition $enabled(l(\bar{y}))$ in the metadata rule attribute. Such theory transformations are incorporated into the Maude LTLR model checker as part of the model checking interface extending Full-Maude.

Given a system module M , the module expression $\text{ACTION}[M]$ builds a module that contains a signature for the basic action patterns $\mathfrak{B} = \{l_1(\bar{y}_1), \dots, l_n(\bar{y}_n)\}$ given by the metadata attributes of the rules in M . For each basic action pattern $l(x_1, \dots, x_n)$ in \mathfrak{B} , where each variable x_i has sort S_i in the corresponding rule in M , the module $\text{ACTION}[M]$ includes the following operators and equations:

- a constructor for the basic action pattern $l(x_1, \dots, x_n)$

```
op l : S1 ... Sn -> BasicActionPattern [ctor] .
```
- assignment operators for each sort S_i of the variable x_i in $l(x_1, \dots, x_n)$

```
op _\_ : Qid Si -> Assignment [ctor ...] .
```
- an equation to define the satisfaction relation with respect to proof terms

```
eq {C | 'l : 'x1 \ x1 ; ... ; 'xn \ xn ; SUBST} |= l(x1, ..., xn) = true .
```

Together with the theory transformation $\text{CONTEXT}[_]$ to generate a context signature, the theory transformation $\text{ACTION}[_]$ replaces a previous theory transformation [1] that was defined in the old version of the LTLR model checker to generate a signature for context terms and assignment operators.

Next, the module expression $\text{FAIR}[M]$ creates a declaration of the state proposition $enabled(l(\bar{y}))$ for each $l(\bar{y}) \in \mathfrak{B}$. Basically, such *enabled* propositions are defined by operators $_enables_ : K \text{ Action} \rightarrow \text{Bool}$ for each relevant kind K , where $E \vdash t \text{ enables } \delta = \text{true}$ means that the one-step rewrite associated to the one-step proof term δ can happen *inside* the term t .

`ceq S:State |= enabled(BSP) = true if S:State enables BSP .`

For each $l(x_1, \dots, x_n) \in \mathfrak{B}$ and its associated rule $l : t \rightarrow t'$ if *cond*, where the kind of t is K_l , the basic declarations of *enables* operators are given as follows:

```
op _enables_ : K_l Action -> Bool .
ceq t enables l(x_1, ..., x_n) = true if cond .
```

For each free constructor operator $f : K_1 \dots K_n \rightarrow K$, where *enables* operators are defined for a nonempty set of kinds $\{K_{i_1}, \dots, K_{i_k}\} \subseteq \{K_1, \dots, K_n\}$, the following declarations of *enables* operators are given in `FAIR[M]`:

```
op _enables_ : K Action -> Bool .
ceq f(X_1:K_1, ..., X_n:K_n) enables BSP
if X_{i_1} enables BSP or-else ... or-else X_{i_k} enables BSP .
```

Finally, for each associative constructor operator $g : K K \rightarrow K$, and for each equation `ceq t enables δ = true if cond` for a kind K , the extended declarations of *enables* operators are given in `FAIR[M]` as follows, where $X_1 : K$ and $X_2 : K$ are fresh variables not appearing in the original equation:

```
op _enables_ : K Action -> Bool .
ceq g(X_1:K, t, X_2:K) enables  $\delta$  = true if cond .
ceq g(X_1:K, t) enables  $\delta$  = true if cond .
ceq g(t, X_2:K) enables  $\delta$  = true if cond .
```

If the associative constructor satisfies another axiom such as commutative or identity, only some of the above equations will be required. Note that the above extended declarations are essentially the generalization of equations for extension matching modulo equational axioms [4].

The *enables* declarations for free and associative constructors can be computed iteratively until reaching a fixed point. Since the right side of each *enables* equation is `true`, we can easily prove the following proposition by induction on the height of the conditional proof tree.

Proposition 1. *Given a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, \mathcal{R})$ with signature of constructors Ω that are free modulo the axiom of B₆ for a basic action pattern $l(\bar{y})$ of \mathcal{R} , a term $[t]_E$, and a substitution ϑ , if $E \vdash t$ enables $\vartheta(l(\bar{y})) = \text{true}$, then there exists a one-step rewrite from $[t]_E$ with a one-step proof term λ such that $E \vdash (\lambda \models l(\vartheta\bar{y})) = \text{true}$.*

5 An Example

This section illustrates a rewriting logic specification of the Evolving Dining Philosophers problem [13] with a localized fairness specification. This problem is similar to the famous Dining Philosophers problem, but a philosopher can join or leave the table, so that the number of philosophers can change dynamically. In this example, it is very hard to specify exact fairness conditions even if an

⁶ That is, $T_{\Sigma/E \cup B} \upharpoonright \Omega \simeq T_{\Omega/B}$.

initial state of the system is already given, because we cannot know how many philosophers can be in the model without exploring the entire state space, but the fairness conditions of this system depend on *each* philosopher in the system.

Each philosopher is represented by a term $\text{ph}(I, S, C)$, where I is the philosopher's id, S is the philosopher's status, and C is the number of chopsticks held. Likewise, a chopstick with id I is represented by a term $\text{stk}(I)$. The configuration of the system is described by a *set* of philosophers and chopsticks, built by an associative-commutative set union operator $_;_$. The signature is defined in the Maude language as follows:

```

sorts Philo Status Chopstick Conf .   op ph : Nat Status Nat -> Philo .
subsort Philo Chopstick < Conf .     ops think hungry : -> Status .
op none : -> Conf .                   op stk : Nat -> Chopstick .
op  $_;_$  : Conf Conf -> Conf [comm assoc id: none] .

```

The state is a triple $\langle P, N, CF \rangle$ with sort **Top**, where P is a global counter, N is the number of philosophers, and CF is a *set* of philosophers and chopsticks. The behavior of philosophers is then described by the following rewrite rules with a necessary localized fairness specification:

```

rl [wake] : ph(I, think, 0) => ph(I, hungry, 0)
  [metadata "just(I)"] .
cr1 [grab] : <P, N, ph(I, hungry, C) ; stk(J) ; CF>
  => <P, N, ph(I, hungry, C+1) ; CF>
  if J == left(I) or J == right(I, N)
  [metadata "fair(I)"] .
rl [stop] : <P, N, ph(I, hungry, 2) ; CF>
  => <P, N, ph(I, think, 0) ; stk(left(I)) ; stk(right(I, N)) ; CF> .

```

The functions $\text{left}(I) = I$ and $\text{right}(I, N) = (I + 1) \text{ rem } N$ return the chopstick's id on the left (resp., right) of philosopher I , where $\text{rem} : \text{Nat Nat} \rightarrow \text{Nat}$ is the remainder operator.

We now specify the dynamic behavior of philosophers in the Evolving Dining Philosopher problem. Although there is no limit to the number of philosophers in the original problem, we can give an unpredictable bound using the Collatz conjecture [6]. There is a global counter P that symbolizes a philosophical problem, and philosophers keep thinking the problem by changing the number n to: (i) $3n + 1$ for n odd, or (ii) $n/2$ for n even.

```

cr1 [solve]: <P, N, ph(I, think, 0) ; CF> => <Q, N, ph(I, think, 0) ; CF>
  if P > 1 /\ Q := collatz(P) .

```

New philosophers can join the group only if the global number is a multiple of the current number of philosophers. No more philosophers can join after the number eventually goes to 1. We assume that only the last philosopher can leave the group for simplicity. To keep consistency, whenever a philosopher joins or leaves the table, the related chopsticks should not be held by another philosopher.

```

crl [join] : <P, N, ph(N, think, 0) ; CF>
            => <P, N+1, ph(N, think, 0) ; ph(N+1, think, 0) ; stk(N+1) ; CF>
    if P rem N == 0 .
crl [leave]: <P, N, CF ; ph(N, think, 0) ; stk(N)> => <P, N-1, CF>
    if N > 2 .
    
```

In order to perform model checking, we define the following module after loading the LTLR model checker interface in Full Maude.

```

(mod PHIL0-CHECK is
  including PHIL0 .
  including LTLR-MODEL-CHECKER .

  subsort Top < State .
  op eating : Nat -> Prop [ctor] .
  op init : -> State .

  vars P N : Nat . var I : NzNat . var CF : Conf .
  eq < P, N, ph(I, hungry, 2) ; CF > |= eating(I) = true .
  eq init = < 12, 2, ph(1,think,0) ; stk(1) ; ph(2,think,0) ; stk(2) > .
endm)
    
```

The state proposition `eating(I)` is satisfied if the philosopher `I` is eating. The initial state is the case of 2 philosophers with the global counter 12, expressed by `< 12, 2, ph(1, think, 0) ; stk(1) ; ph(2, think, 0) ; stk(2) >`.

We are interested in verifying the liveness property `[] ~ deadlock -> <> eating(1)`, where `deadlock` is a spatial action pattern satisfied on deadlock states [1]. Without fairness assumptions, the model checker generates the following counterexample for this formula, in which only the philosopher 2 performs actions while the order philosophers keep idle and no new philosopher joins:

```

Maude> (mc init |= [] ~ deadlock -> <> eating(1) .)
ltlr model check in PHIL0-CHECK :
  init |= [] ~ deadlock -> <> eating(1)
result ModelCheckResult :
  counterexample(
    {< 12,2,stk(1) ; stk(2) ; ph(1,think,0) ; ph(2,think,0)>, {'solve : 'I \ 1}}
    {< 6,2,stk(1) ; stk(2) ; ph(1,think,0) ; ph(2,think,0)>, {'solve : 'I \ 1}}
    ...,
    {< 1,2,stk(1) ; stk(2) ; ph(1,hungry,0) ; ph(2,think,0)>, {'wake : 'I \ 2}}
    {< 1,2,stk(1) ; stk(2) ; ph(1,hungry,0) ; ph(2,hungry,0)>,
      {'grab : 'I \ 2 ; 'J \ 1}}
    {< 1,2,stk(2) ; ph(1,hungry,0) ; ph(2,hungry,1)>, {'grab : 'I \ 2 ; 'J \ 2}}
    {< 1,2,ph(1,hungry,0) ; ph(2,hungry,2)>, {'stop : 'I \ 2}} )
    
```

When we assume localized fairness conditions, the model checker can verify the formula `[] ~ deadlock -> <> eating(1)` as follows:

```

Maude> (pfmc init |= [] ~ deadlock -> <> eating(1) .)
    
```

```

ltlr model check under localized fairness in PHILO-CHECK :
  init |= []~ deadlock -> <> eating(1)
result Bool :
  true

```

Note that the reachable state space from the initial state has 12 ground fairness conditions instantiated by realized substitutions. The previous LTL and LTLR model checkers cannot verify the formula with those 12 fairness conditions in a reasonable time. Furthermore, using the previous model checker, we could not know how many ground fairness conditions would be required to prove the formula before exploring the entire state space .

6 Related Work and Conclusions

The usual model checking method to verify a property φ under parameterized fairness assumptions, is to construct the conjunction of corresponding instances of fairness, and to apply either: (i) a standard LTL model checking algorithm for the reformulated property $fair \rightarrow \varphi$, or (ii) a specialized model checking algorithm which handles fairness, based on either explicit graph search [7,9,14], or a symbolic algorithm [12]. Approach (i) is inadequate for fairness, since the time complexity is exponential in the number of strong fairness conditions, while the other is linear. Furthermore, compiling such a formula, expressing a conjunction of fairness conditions, into Büchi automata is usually not feasible in reasonable time [19]. There are several tools to support the specialized algorithms such as PAT [18] and Maria [14]. Our tool is related to the second approach to deal with fairness, but it does not require pre-translation of parameterized fairness, and can handle *dynamic* fairness instances.

In conclusion, we have addressed the real need of verifying temporal logic properties under parametric fairness assumptions. Such parametric assumptions occur very often in practice, but up to now have not been supported by existing model checking techniques and tools. To address this need three things are required: (i) expressive system specification languages; (ii) expressive temporal logics; and (iii) novel model checking techniques and tools. This paper has argued and demonstrated with examples that rewriting logic answers very well need (i) and that TLR, and in particular LTLR, are very expressive to deal with need (ii). It has also presented a novel Maude LTLR model checker under localized fairness which directly addresses need (iii) in an efficient way.

Acknowledgments. This work has been supported in part by NSF Grants CNS 09-04749 and CCF 09-05584, AFOSR Grant FA8750-11-2-0084, Boeing Grant C8088-557395, and the “Programa de Apoyo a la Investigación y Desarrollo” (PAID-02-11) of the Universitat Politècnica de València.

References

1. Bae, K., Meseguer, J.: The Linear Temporal Logic of Rewriting Maude Model Checker. In: Ólveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 208–225. Springer, Heidelberg (2010)

2. Bae, K., Meseguer, J.: State/Event-Based LTL Model Checking under Parametric Generalized Fairness. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 132–148. Springer, Heidelberg (2011)
3. Chaki, S., Clarke, E., Ouaknine, J., Sharygina, N., Sinha, N.: State/Event-Based Software Model Checking. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 128–147. Springer, Heidelberg (2004)
4. 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)
5. Couvreur, J.-M., Duret-Lutz, A., Poitrenaud, D.: On-the-Fly Emptiness Checks for Generalized Büchi Automata. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 169–184. Springer, Heidelberg (2005)
6. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems* 19, 253–291 (1997)
7. Duret-Lutz, A., Poitrenaud, D., Couvreur, J.-M.: On-the-fly Emptiness Check of Transition-Based Streett Automata. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 213–227. Springer, Heidelberg (2009)
8. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL Model Checker and Its Implementation. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 230–234. Springer, Heidelberg (2003)
9. Henzinger, M.R., Telle, J.A.: Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning. In: Karlsson, R., Lingas, A. (eds.) SWAT 1996. LNCS, vol. 1097, pp. 16–27. Springer, Heidelberg (1996)
10. Holzmann, G., Peled, D., Yannakakis, M.: On nested depth first search (extended abstract). In: *The Spin Verification System*, pp. 23–32. American Mathematical Society (1996)
11. Holzmann, G.: *The SPIN model checker: Primer and reference manual*. Addison Wesley Publishing Company (2004)
12. Kesten, Y., Pnueli, A., Raviv, L., Shahar, E.: Model checking with strong fairness. *Formal Methods in System Design* 28(1), 57–84 (2006)
13. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering* 16(11), 1293–1306 (2002)
14. Latvala, T.: Model Checking LTL Properties of High-Level Petri Nets with Fairness Constraints. In: Colom, J.-M., Koutny, M. (eds.) ICATPN 2001. LNCS, vol. 2075, pp. 242–262. Springer, Heidelberg (2001)
15. Meseguer, J.: Localized Fairness: A Rewriting Semantics. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 250–263. Springer, Heidelberg (2005)
16. 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)
17. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. *Theoretical Computer Science* 403(2-3), 239–264 (2008)
18. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
19. Vardi, M.Y.: Automata-Theoretic Model Checking Revisited. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 137–150. Springer, Heidelberg (2007)

Modelling and Analyzing Adaptive Self-assembly Strategies with Maude^{*}

Roberto Bruni¹, Andrea Corradini¹, Fabio Gadducci¹,
Alberto Lluch Lafuente², and Andrea Vandin²

¹ Dipartimento di Informatica, Università di Pisa, Italy
{bruni, andrea, gadducci}@di.unipi.it

² IMT Institute for Advanced Studies Lucca, Italy
{alberto.lluch, andrea.vandin}@imtlucca.it

Abstract. Building adaptive systems with predictable emergent behavior is a challenging task and it is becoming a critical need. The research community has accepted the challenge by introducing approaches of various nature: from software architectures, to programming paradigms, to analysis techniques. We recently proposed a conceptual framework for adaptation centered around the role of control data. In this paper we show that it can be naturally realized in a reflective logical language like Maude by using the Reflective Russian Dolls model. Moreover, we exploit this model to specify and analyse a prominent example of adaptive system: robot swarms equipped with obstacle-avoidance self-assembly strategies. The analysis exploits the statistical model checker PVesta.

Keywords: Adaptation, self-assembly, swarms, ensembles, Maude.

1 Introduction

How to engineer autonomic system components so to guarantee that certain goals will be achieved is one of today's grand challenges in Computer Science. First, autonomic components run in unpredictable environments, hence they must be engineered by relying on the smallest possible amount of assumptions, i.e. as *adaptive* components. Second, no general formal framework for adaptive systems exists that is widely accepted. Instead, several adaptation models and guidelines are presented in the literature that offer ad hoc solutions, often tailored to a specific application domain or programming language. Roughly, there is not even a general agreement about what "adaptation" is. Third, it is not possible to mark a b/w distinction between failure and success, because the randomized behaviour of the system prevents an absolute winning strategy to exist. Fourth, efforts spent in the accurate analysis of handcrafted adaptive components are unlikely to pay back, because the results are scarcely reusable when the components software is frequently updated or extended with new features.

We address here some of the above concerns, presenting the methodology we have devised for prototyping well-engineered self-adaptive components. Our

^{*} Research supported by the European Integrated Project 257414 ASCENS.

case study consists of modeling and analyzing self-assembly strategies of robots whose goal is crossing a hole while navigating towards a light source. We specified such robots with Maude, exploiting on one hand the Reflective Russian Dolls (RRD) model [21] and on the other hand the conceptual framework we proposed in [6], which provides simple but precise guidelines for a clean structuring of self-adaptive systems. We report also on the results of the analysis of our model using PVesta [2].

When is a software system adaptive? Self-adaptation is a fundamental feature of autonomic systems, that can specialize to several other so-called self-* properties (like self-configuration, self-optimization, self-protection and self-healing, as discussed e.g. in [10]). Self-adaptive systems have become a hot topic in the last decade: an interesting taxonomy of the concepts related to self-adaptation is presented in [18]. Several contributions have proposed reference models for the specification and structuring of self-adaptive software systems, ranging from architectural approaches (including the well-known MAPE-K [9,10,12], FORMS [23], the adaptation patterns of [7], and the already mentioned RRD [21]), to approaches based on model-based development [24] or model transformation [11], to theoretical frameworks based on category theory [17] or stream-based systems [5].

Even if most of those models have been fruitfully adopted for the design and specification of interesting case studies of self-adaptive systems, in our view they missed the problem of characterizing *what is adaptivity* in a way that is independent of a specific approach. We have addressed this problem in [6], where we have proposed a very simple criterion: a software system is *adaptive* if its behaviour depends on a precisely identified collection of *control data*, and such control data can be modified at run time. We discuss further this topic in §3.

Is Maude a convenient setting to study self-adaptation? A “convenient” framework must provide a reusable methodology for modelling self-adaptive systems independently of their application domain together with a flexible analysis toolset to investigate formal properties of the semantics of such systems. There are several reasons why we think that Maude [8] is a good candidate. First, the versatility of rewrite theories can offer us the right level of abstraction for addressing the specification, modelling and analysis of self-adaptive systems and their environments within one single coherent framework. Second, since Maude is a rule-based approach, the control-data can be expressed naturally as a sub-set of the available rules and the reflection capability of Maude can be exploited to express control-data manipulation via ordinary rewrite rules, along the so-called *tower of reflection* and its modular realization as the RRD approach [14]. Third, the conceptual framework for adaptation described in [6], to be further elaborated in §4, facilitates early and rapid prototyping of self-adaptive systems, to be simulated. Fourth, the formal analysis toolset of Maude can support simulations and analysis over the prototypes. In particular, given the probabilistic nature of adaptive systems, where absolute guarantees cannot be proved, we think that the statistical model checker PVesta [2] can be useful, because it allows to conduct analysis that are parametric w.r.t. the desired level of statistical confidence.

Pragmatically, the possibility to rapidly develop and simulate self-adaptive systems and to compare the behaviour emerging from different adaptation strategies at the early stages of software development is very important for case studies like the robotic scenario described in the next paragraphs. Indeed, such physical devices require specialized programming skills and their experimentation in real world testing environments involves long time consumption (6 hours or more for each run) and only a limited number of pieces is available (around 25 units) because their maintenance is expensive. Also, their hardware (both mechanic and electronic parts) and software are frequently updated, making it harder to build, to maintain and to rely on sophisticated simulators that can take as input exactly the same code to be run on the robots. Even when this has been attempted, the tests conducted on the real systems can differ substantially from the simulated runs. Thus, early simulation on prototypes can at least speed-up debugging and dispense the programmers from coding lowest-performance strategies.

Synopsis. In §2 we present the case study analysed in this paper. In §3 we summarize the conceptual framework for adaptation proposed in [6], along which we design adaptive systems in Maude. The general guidelines and principles to be exploited in Maude for modelling self-adaptive systems are described in §4, together with the software architecture used to realize our conceptual framework. In §5 we illustrate the concrete implementation of the case study, while the experimentations are described in §6; for the sake of presentation, we focus on just one of the self-assembly strategies. Some concluding remarks and ongoing research avenues are discussed in §7.

We assume the reader to have some familiarity with the Maude framework.

2 Case Study: Self-assembly Robot Swarms

Self-assembly robotic systems are formed by independent robots capable to connect physically when the environment prevents them from reaching their goals individually. Self-assembly is a contingency mechanism for environments where versatility is a critical issue and the size and morphology of the assembly cannot be known in advance. Thus, self-assembly units must be designed in a modular way and their logic must be more sophisticated than, say, that of cheaper pre-assembled units. Such features make the self-assembly robot swarm a challenging scenario to engineer.

In [16], different self-assembly strategies are proposed to carry out tasks that range from hill-crossing and hole-crossing to robot rescue: case by case, depending e.g. on the steepness of the hill, the width of the hole, the location of the robot to be rescued, the robots must self-assemble because incapable to complete the task individually. We focus on the *hole-crossing scenario* as a running case study, where “the robots in the swarm are required to cross a hole as they navigate to a light source” and depending on the width of the hole “a single unit by itself will fall off into the crevice, but if it is a connected body, falling can be prevented”.

The experiments in [16] were conducted on the SWARM-BOT robotic platform [15], whose constituents are called s-bots (see Fig. 6, bottom right).

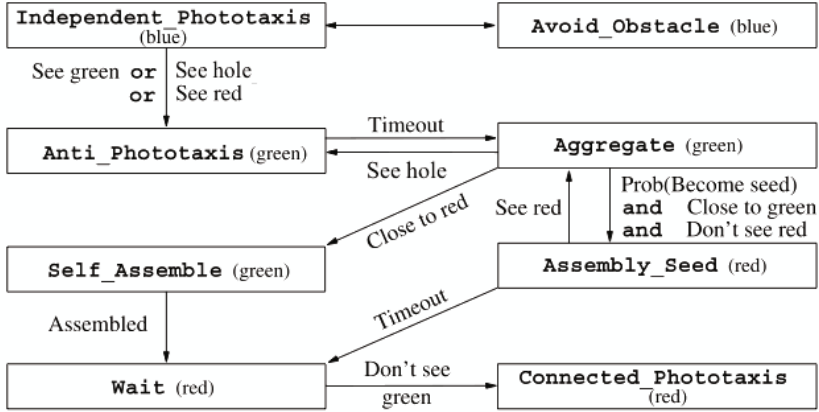


Fig. 1. Excerpt of the basic self-assembly response strategy (borrowed from [16])

Each s-bot has a traction system that combines tracks, wheels and a motorised rotation system, has several sensors (including infra-red proximity sensors to detect obstacles, ground facing proximity sensors to detect holes, and a 360 degrees view thanks to a camera turret), and is surrounded by a transparent ring that contains eight RGB colored LEDs (Light Emitting Diodes) distributed uniformly around the ring. The LEDs can provide some indications about the internal state of the s-bot to (the omni-directional cameras of) nearby s-bots. For example, the green color can be used to signal the willingness to connect to an existing ensemble, and the red color can be used for the willingness to create a new assembly. The ring can also be grasped by other s-bots thanks to a gripper-based mechanism.

Roughly, the strategies described in [16] are: (i) the *independent execution* strategy, where s-bots move independently one from the other and never self-assemble; (ii) the *basic self-assembly response* strategy (see below), where each s-bot moves independently (blue light) until an obstacle is found, in which case it tries to aggregate (green light) to some nearby assembly, if any is available, or it becomes the *seed* of a new assembly (red light); (iii) the *preemptive self-assembly* strategy, where the s-bots self-assemble irrespectively of the environment and not by emergency as in the basic self-assembly response; (iv) the *connected coordination* strategy, where the sensing and actuation of assembled robots is coordinated according to a leader-follower architecture.

The experiments were conducted with different strategies in a few alternative scenarios (with holes of various dimensions and random initial positions of the s-bots) and repeated for each strategy within each scenario (from a minimum of 20 times and 2 s-bots to a maximum of 60 times and 6 s-bots). Videos of the experiments described in [16] are linked from the web page describing our Maude implementation: <http://sysma.lab.imtlucca.it/tools/ensembles>.

Basic self-assembly response strategy. We describe here the *basic self-assembly strategy* of [16], which is the strategy on which we will focus in the rest of the

paper. The finite state machine of the strategy is depicted in Fig. 11. Each state contains its name and the color of the LEDs turned on in that state, while transitions are labelled with their firing condition.

This controller is executed independently in each individual s-bot (a concrete one in [16], or a software abstraction in this work). In the starting state (**Independent_Phototaxis**) each s-bot turns on its blue LEDs, and navigates towards the target light source, avoiding possible obstacles (e.g. walls or other robots). If an s-bot detects a hole (through its infra-red ground sensors), or sees a green or red s-bot, then it switches to state **Anti_Phototaxis**, i.e. it turns on its green LEDs and retreats away from the hole.

After the expiration of a timeout, the s-bot passes to state **Aggregate**: it randomly moves searching for a red (preferably) or a green s-bot. In case it sees a red s-bot, it switches to state **Self_Assemble**, assembles (grabs) to the red s-bot, turns on its red LEDs and switches to state **Wait**. If instead it sees a green s-bot, with probability $\text{Prob}(\text{Become seed})$ it switches to state **Assembly_Seed**, turns on its red LEDs, and becomes the seed of a new ensemble. Once in state **Assembly_Seed**, the s-bot waits until a timeout expires and switches to state **Wait**, unless it sees another red s-bot, in which case it reverts to state **Aggregate**. Once no green s-bots are visible, assembled “waiting” s-bots switch to state **Connected_Phototaxis** and navigate to the light source.

3 A Framework for Adaptation

Before describing how we modeled and analysed the scenario we just presented, let us explain some guidelines that we followed when designing the system. The main goal was to develop a software system where the adaptive behaviour of the robots is explicitly represented in the system architecture. To this aim, we found it necessary to first understand “*when is a software system adaptive*”, by identifying the features distinguishing such systems from ordinary (“non-adaptive”) ones.

We addressed this problem in [6], proposing a simple structural criterion to characterize adaptivity. Oversimplifying a bit, according to a common *black-box* perspective, a software system is “self-adaptive” if *it can modify its behaviour as a reaction to a change in its context of execution*. Unfortunately this definition is hardly usable: accordingly to it, almost any software system can be considered self-adaptive. Indeed, any system can *modify its behaviour* (e.g. executing different instructions, depending on conditional statements) as a *reaction to a change in the context of execution* (like the input of a data from the user).

We argue that to distinguish situations where the modification of behaviour is part of the application logic from those where they realize the adaptation logic, we must follow a *white-box* approach, where the internal structure of a system is exposed. Our framework requires to make explicit that the behavior of a component depends on some well identified *control data*. We define *adaptation* as the *run-time modification of the control data*. From this definition we derive that a component is called *adaptable* if it has a clearly identified collection of

control data that can be modified at run-time. Further, a component is *adaptive* if it is adaptable and its control data are modified at run-time, at least in some of its executions; and it is *self-adaptive* if it can modify its own control data.

Under this perspective, and not surprisingly, any computational model or programming language can be used to implement an adaptive system, just by identifying the part of the data governing the behavior. Consequently, the nature of control data can greatly vary depending on the degree of adaptivity of the system and on the computational formalisms used to implement it. Examples of control data include configuration variables, rules (in rule-based programming), contexts (in context-oriented programming), interactions (in connector-centered approaches), policies (in policy-driven languages), aspects (in aspect-oriented languages), monads and effects (in functional languages), and even entire programs (in models of computation exhibiting higher-order or reflective features).

In [6] we discussed how our simple criterion for adaptivity can be applied to several of the reference models we mentioned in the introduction, identifying what would be a reasonable choice of control data in each case. Interestingly, in most situations the explicit identification of control data has the effect of revealing a precise interface between a managed component (mainly responsible for the application logic) and a control component (encharged of the adaptation logic). As a paradigmatical example, consider the MAPE-K architecture [9], according to which a self-adaptive system is made of a component implementing the application logic, equipped with a control loop that *monitors* the execution through sensors, *analyses* the collected data, *plans* an adaptation strategy, and finally *executes* the adaptation of the managed component through effectors; all the phases of the control loop access a shared *knowledge* repository. Applying our criterion to this model suggests a natural choice for the control data: these must include the data of the managed component that are modified by the execute phase of the control loop. Clearly, by our definitions the managed component is adaptive, and the system made of both component and control loop is self-adaptive.

The construction can be iterated, as the control loop itself could be adaptive. Think e.g. of an adaptive component which follows a plan to perform some tasks.

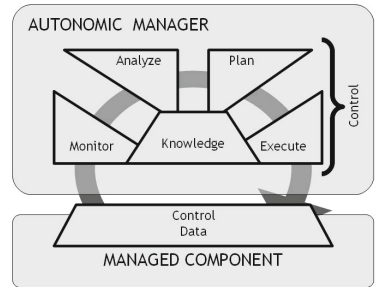


Fig. 2. Control data in MAPE-K

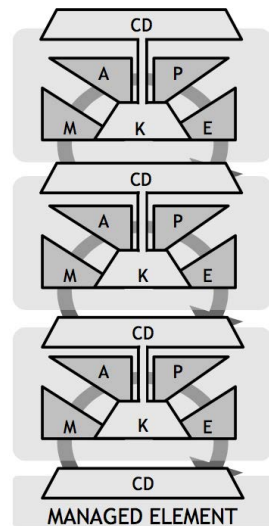


Fig. 3. Tower of adaptation

This component might have a manager which devises new plans according to changes in the context or in the component’s goals. But this planning component might itself be adaptive, where some component controls and adapts its planning strategy, for instance determining the new strategy on the basis of a tradeoff between optimality of the plans and computational cost. In this case the manager itself (the control loop) should expose its control data (conceptually part of its knowledge repository) in its interface. In this way, the approach becomes compositional in a layered way, which allows one to build towers of adaptive components (Fig. 3) as we do in §5 and §6 for robot prototypes.

4 Adaptivity in Maude

We argue here the suitability of Maude and rewriting logic as a language and a model for adaptivity (§4.1), we describe a generic architecture for developing adaptive components in Maude (§4.2) and we show that it conforms to well-assessed conceptual models for adaptivity, including our framework (§4.3).

4.1 Maude, Logical Reflection and Adaptivity

As argued in [14], Rewriting Logic (RL) is well-suited for the specification of adaptive systems, thanks to its reflective capabilities. The reflection mechanism yields what is called the *tower of reflection*. At the ground level, a rewrite theory \mathcal{R} (e.g. a software module) allows to infer a computation step $\mathcal{R} \vdash t \rightarrow t'$ from a term t (e.g. a program state) to a term t' . A universal theory \mathcal{U} lets infer the computation $\mathcal{U} \vdash (\overline{\mathcal{R}}, \bar{t}) \rightarrow (\overline{\mathcal{R}'}, \bar{t}')$ at the “meta-level” where theories and terms are meta-represented as terms. The process can be repeated as \mathcal{U} itself is a rewrite theory. This mechanism is efficiently supported by Maude and fostered many meta-programming applications like analysis and transformation tools. Since a theory can be represented by a term, it is also possible to specify *adaptation rules* that change the (meta-representation of the) theory, as in $r \vdash (\overline{\mathcal{R}}, \bar{t}) \rightarrow (\overline{\mathcal{R}'}, \bar{t}')$, so that the reduction continues with a different set of rules \mathcal{R}' .

The reflection mechanism of RL has been exploited in [14] to formalize a model for distributed object reflection, suitable for the specification of adaptive systems. Such model, called Reflective Russian Dolls (RRD), has a structure of layered configurations of objects, where each layer can control the execution of objects in the lower layer by accessing and executing the rules in their theories, possibly after modifying them, e.g. by injecting some specific adaptation logic in the wrapped components. It is worth stressing that logical reflection is only one possible way in which a layer can control the execution of objects of the lower level: objects within a layer interact via message passing, thus objects of the higher layer might intercept messages of the lower level, influencing their behaviour. But even if the resulting model is still very expressive, some form of reflection seems to be very convenient, if not necessary, to implement adaptivity. This is clearly stated in [14] and at a more general level in [3], where (*computational*) *reflection is promoted as a necessary criterion for any self-adaptive software system*.

The RRD model has been exploited for modeling policy-based coordination [21] and for the design of PAGODA, a modular architecture for specifying autonomous systems [22].

4.2 Generic Architecture

This section describes how we specialize the RRD architecture for modeling adaptive components. We focus on the structure of the layers and on the interactions among them, abstracting from the details of our case study, discussed in §5.

Intra-layer Architecture. Each layer is a component having the structure illustrated in Fig. 4. Its main constituents are: *knowledge* (K), *effects* (E), *rules* (R) and *managed component* (M). Some of them are intentionally on the boundary of the component, since they are part of its interface: knowledge and effects act respectively as input and output interfaces, while rules correspond to the component's control interface.

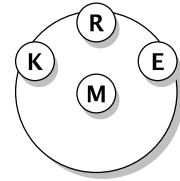


Fig. 4. Intra-layer

Therefore we will consider the rules R as the *control data* of a layer.

The managed component is a lower-level layer having the same structure: clearly, this part is absent in the innermost layer. The knowledge represents the information available in the layer. It can contain data that represent the internal state or assumptions about the component's surrounding environment. The effects are the actions that the component is willing to perform on its enclosing context. The rules determine which effects are generated on the basis of the knowledge and of the interaction with the managed component. Typical rules update the knowledge of the managed component, execute it and collect its effects. In this case the layer acts as a sort of interpreter. In other cases rules can act upon the rules of the managed component, modifying them: since such rules are control data, the rules modifying them are *adaptation rules* according to §3.

Inter-layer Architecture. Layers are organized hierarchically: each one contains its knowledge, effects, rules and, in addition, the managed underlying layer (see the leftmost diagram of Fig. 5). The outermost layer interacts with the environment: its knowledge represents the perception that the adaptive component has of the environment, while its effects represent the actions actually performed by the component. Each layer elaborates its knowledge and propagates it to the lower one, if any. In general, while descending the hierarchy, the knowledge becomes simpler, and the generated effects more basilar. Similarly to layered operating systems, each layer builds on simpler functionalities of the lower one to compute more complex operations.

The diagram in the middle of Fig. 5 shows the control and data flow of ordinary behavior (without adaptations). Knowledge is propagated down to the core (layer 0) and the effects are collected up to the skin (layer 2). This flow of information

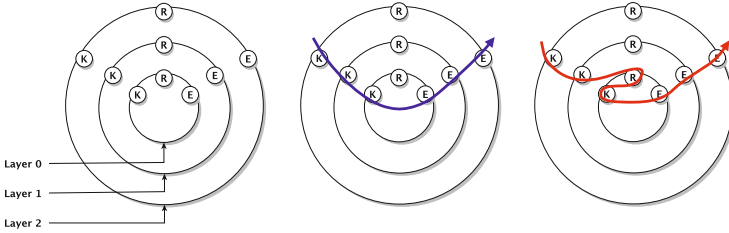


Fig. 5. Inter-layer architecture (left), ordinary flow (center), adaptation flow (right)

is governed by the rules. Knowledge and effects are subject to modifications before each propagation. For example, layer 2 may decide to propagate to layer 1 only part of the knowledge perceived from the environment, possibly after pre-processing it. Symmetrically, layer 1 may decide to filter part of the effects generated by layer 0 before the propagation to layer 2, for example discarding all those violating some given constraints.

The rightmost diagram of Fig. 5 corresponds to a phase of adaptation. Here the outermost layer triggers an adaptation at layer 1. This can be due to some conditions on the knowledge of layer 2 or to the status of the managed component (layer 1). The result is that the rules of layer 2 change (among other things) the rules of layer 1 (as shown by the arrow crossing the corresponding *R* attribute).

4.3 Generic Architecture and Adaptation Frameworks

Let us relate the generic architecture just presented with some general frameworks used for modeling adaptive systems. As suggested in §3, we identified explicitly the control data of each layer, i.e., its set of rules: this will allow us to distinguish the adaptation behaviour from the standard computations of the system.

Our architecture is a simplified version of the RRD of [14], because each layer is a single object rather than a proper configuration. The interaction between a layer and its managed component is realized both with logical reflection and with access to shared data (knowledge and effects). Further, there is a clear correspondence between the reflective tower of the RRD model and the adaptation tower discussed in §3, as depicted in Fig. 6, showing that the rules of each layer implement the MAPE control loop on the lower layer. Moreover, the generic architecture imposes the encapsulation of all components of the tower, apart from the robot itself. This offers several advantages: (i) management is hierarchical (e.g. self- or mutually-managing layers are excluded); and (ii) at each level in the hierarchy the adaptation logic of the underlying layer is designed separately from the execution of basic functionalities, delegated to lower layers.

5 Concrete Architecture and Case Study Implementation

This section instantiates the generic architecture shown in §4.2 to our case study (§5.1), and presents some relevant details of its implementation (§5.2). We will call s-bots simply robots in the following.

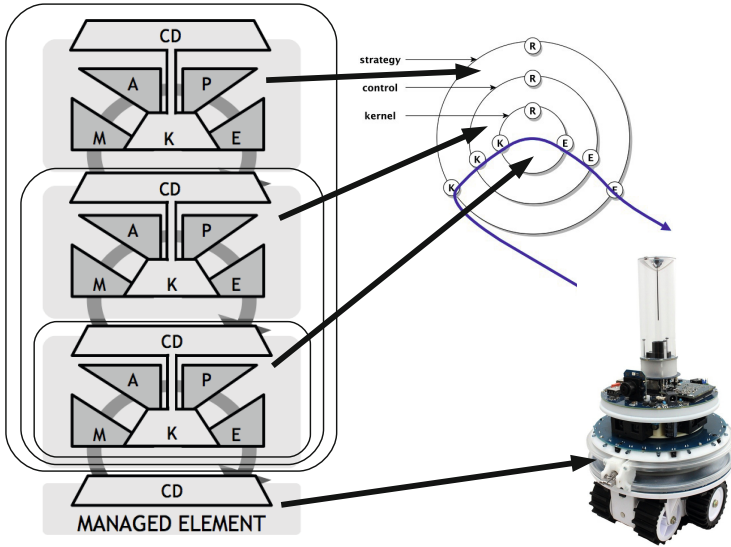


Fig. 6. Architecture as an instance of the framework

5.1 Architecture of the Case Study

The three layers of the concrete architecture of the case study (cf. Fig. 6, top-right) essentially capture the informal description of [16], in the following sense.

Layer 0 (kernel). This layer models the core functionalities of a robot (see [16, §3]). The rules implement basic movements and actioning of the gripper. Layer 0 corresponds to what some authors call *hardware abstraction layer* (see e.g. [22]).

Layer 1 (basic control). This layer represents the basic controller managing the core functionalities of the robot according to the context. The controller may allow to move only in some directions (e.g. towards a light source) or to search for a robot to grab. This layer corresponds to the individual states of state machines modeling the self-assembly strategies, like the one of Fig. 1 (see [16, §5 and §7]).

Layer 2 (adaptation). This is the layer of the adaptation manager, which reacts to changes in the environment activating the proper basic controller. In our case study, this layer corresponds to the entire state machine modelling the self-assembly strategy of Fig. 1 and, in particular, it takes care of the transitions between its states. This is done by constantly monitoring the environment and the managed component M , and by executing an adaptation phase when needed, which means changing the rules of M . A few other self-assembly strategies are discussed in [16]: they can be implemented by changing the rules of this layer.

The three layers differ in their sets of rules and, of course, in the managed component, but they share part of the signature for knowledge and effects. In particular, knowledge includes predicates about properties of the ground (wall,

hole, free), the presence of robots in the surrounding (their LED emissions), and the direction of the light source (the goal). Effects include moving or emitting a color towards a direction, and trying to grab a robot located in an adjacent cell.

Knowledge and effects are currently implemented as plain sets of predicates. More sophisticated forms of knowledge representation based on some inference mechanism (like PROLOG specifications, epistemic logics, ontologies or constraints) may be possible but are not necessary in the case study we present.

Simulator. The execution environment of the robots is realized by a simulator which consists of three parts: the *orchestrator*, the *scheduler* and the *arena*.

The orchestrator takes care of the actual execution of the actions required to manage the effects generated by (the outermost layer of) a robot. For instance, it decides if a robot can actually move in the direction it is willing to move.

The scheduler, implemented as an ordinary discrete-event scheduler, activates the scheduled events, allowing a robot or the orchestrator to perform its next action. Intuitively, the emission of an effect e by the outermost layer of a component c causes the scheduling of the event “execute effect e on c ” for the orchestrator. Symmetrically, the handling by the orchestrator of an effect previously generated by a component c induces the scheduling of an event “generate next effect” for c .

Finally, the arena defines the scenario where robots run. We abstracted arenas in discrete grids, very much like a chessboard. Each grid’s cell has different attributes regarding for example the presence of holes or light sources. A cell may also contain in its attributes (at most) one robot, meaning that the robot is in that position of the arena. Each robot can move or perform an action in eight possible directions (up, down, left, right and the four diagonals).

5.2 Implementation Details

On the Structure of Adaptive Components. Our implementation, similarly to the systems described in [14], relies on Maude’s object-like signature (see [8, Chapter 8]). Such signature allows to model concurrent systems as *configurations* (collections) of *objects*, where each object has an identifier, a class and a set of attributes. Intuitively, $\langle \text{oid} : \text{cid} \mid \text{attr1}, \text{attr2} \rangle$ is an object with identifier oid , class cid and two attributes attr1 , attr2 .

Each layer is implemented as an object with attributes for knowledge (\mathbf{K}), effects (\mathbf{E}), rules (\mathbf{R}) and managed component (\mathbf{M}): the first two are plain sets of predicates, the third one is a meta-representation of a Maude module, and the fourth one is an object. Three classes are introduced for the different layers, namely AC0 , AC1 and AC2 . For design choice, the objects implementing the layers of a robot have the same identifier: in terms of [14] we use *homunculus objects*.

Therefore a sample robot can have the following overall structure

```
< c(0) : AC2 | K: gripper(open) on(right,none) towards(right,light) ...,
    E: emitt(up,Green) go(right) ...,
    R: mod_is_sorts_.....endm,
    M: < c(0) : AC1 | K: ..., E: ..., R: ...,
        M: < c(0):AC0 | K:..., E:..., R:...> >
```


On the Structure of the Simulator. The arena is implemented as a multi-set of objects of class `Cell`. A cell may contain in the attributes an object of class `AC2` representing a robot, and the orchestrator implements the move of a robot by changing the cell in which it is stored. This way the robots have no information about the global environment or their current position, but only about the contiguous cells and the direction to take to reach the goal.

The cell encapsulating a robot actually acts as a fourth layer over the object of class `AC2`. In fact, it is responsible of updating its knowledge, of taking care of its effects (e.g. the cell must expose the status of robot's LEDs), and of handling the interactions between the robot and the scheduler.

Rules of Adaptive Components. The behaviour of each layer is specified by the rules contained in its attribute `R`: a term of sort `Module` consisting of a meta-representation of a Maude module. This solution facilitates the implementation of the behaviour of components as ordinary Maude specifications and their treatment for execution (by resorting to meta-level's rewriting features), monitoring and adaptation (by examining and modifying the meta-representation of modules). In fact, on the one hand a generic meta-rule can be used to *self-execute* an object: the object with rules R proceeds by executing R in its meta-representation. On the other hand, rules are exposed to the outer component, which can execute or manipulate the inner one, and analyse the obtained outcome.

In order to give an idea on how the flows of execution and information of Fig. 5 are actually implemented, we present one sample rule for each of the three layers. For the sake of presentation we abstract from irrelevant details.

Layer 0. This layer implements the core functionalities of robots. For example, the following rule computes the set of directions towards which a robot can move

```

rl [admissibleMovements] :
  < oid : ACO | K: oneStep k0, E: e0 , AO >
=> < oid : ACO | K: k0, E: e0 canMoveTo(freeDirs(k0)), AO > .

```

A rule, like `admissibleMovements`, can be applied to a Maude term t if its left-hand side (LHS) (here the object `< oid : ACO | ... >` preceding `=>`) matches a subterm of t with some matching substitution σ , and in this case the application consists of replacing the matched sub-term with the term obtained by applying σ to the right-hand side, i.e. the object following `=>`. We shall also use Maude equations: they have higher priority than rules, meaning that rules are applied only to terms in normal form w.r.t. the equations.

Rule `admissibleMovements` rewrites an `ACO` object to itself, enriching its effects with the term obtained by simplifying equationally `canMoveTo(freeDirs(k0))`. Notice that the constant `oneStep` is consumed by the application of the rule: intuitively, it is a token used to inhibit further applications of the rule, obtaining a one step rewriting. The equations will reduce `freeDirs(k0)` to the set of directions containing each `dir` appearing in a fact `on(dir, content)` of `k0` such that `content` does not contain obstacles. Operator `canMoveTo` instead is a constructor, hence it cannot be further reduced.

Layer 1. Objects of class **AC1** correspond to components of layer 1, implementing individual states of the state machine of Fig. 11. Rules of this layer can execute the component of the lower level providing additional knowledge, and can elaborate the resulting effects. The following rule implements (part of) the logic of state **Independent_Phototaxis**, computing the desired direction towards which to move

```

cr1 [IP-main]: < oid:AC1 | K: oneStep k1, E: e1          ,
                M: < oid:ACO | K: k0 , E: e0, R: m0, AO >, A1 >
=>
    < oid:AC1 | K:          k1, E: e1 go(dir),
                M: < oid:ACO | K: k0b, E: e0, R: m0, AO b >, A1 >
if < oid : ACO | K: k0b, E: e0 canMoveTo(freeDirs), AO b > :=
    execute(< oid : ACO | K: oneStep update1To0(k1,k0), E: e0, AO >, m0)
/\ preferredDirs := intersection(freeDirs, dirsToLight(k1))
/\ dir := uniformlyChooseDir(preferredDirs, freeDirs) .

```

This is a conditional rule, as evident from the keyword `cr1` and the `if` clause following the RHS. Thus, it can be applied to a matched sub-term only if its (firing) condition is satisfied under the matching. In this case the condition is the conjunction (\wedge) of three sub-conditions, each consisting of a sort of assignment. The sub-conditions are evaluated sequentially, and the LHS of symbol `:=` will be bound in the rest of the rule to the term obtained by reducing its RHS.

The first sub-condition exploits reflection, since `execute(obj,m)` makes use of Maude's meta-level functionalities to execute object `obj` via the rules meta-represented in `m`. More precisely, in rule `IP-main` the operator `execute` will apply a single rule of module `m0` to the managed component `< oid : ACO ... >`, after having updated its knowledge. In fact, the operation `update1To0(k1,k0)` implements a (controlled) propagation of the knowledge from layer 1 to layer 0, filtering `k1` before updating `k0` (e.g. information about the surrounding cells is propagated, but information about the goal is discarded).

The assignment of the first sub-condition also binds `freeDirs` to the directions towards which the managed component can move. This is used in the second sub-condition to compute the intersection between the directions in `freeDirs` and those towards the light, evaluated reducing `dirsToLight(k1)`. The resulting set of directions is bound to `preferredDirs`. Finally, in the third sub-condition `dir` is bound to a direction randomly chosen from `preferredDirs`, or from `freeDirs` if the first set is empty. Comparing the LHS and the RHS, one sees that the overall effect of rule `IP-main` is the production of a new effect at layer 1, `go(dir)`, and the update of the knowledge of the managed component of layer 0.

Notice that the rules of layer 0 (`m0`) are not affected by the rule: in fact in our implementation rules of layer 1 never trigger an adaptation phase on layer 0. This is just a design choice, as clearly our architecture does not forbid it.

Layer 2. A component of this layer corresponds to the entire state machine of Fig. 11. It monitors the environment, and at each step it triggers a reduction of the managed component of layer 1; if necessary, it also enforces a transition from the current state to a new one of the state machine by performing an adaptation phase, i.e. by changing the rules of the managed component.

The following rule is the main one governing this layer

```

cr1 [adaptAndExecute]:
  < oid : AC2 | K: nextEffect k2 , E: e2
    M: < oid : AC1 | K: k1 , E: e1 , R: m1 , A1 > , A2 >
=> < oid : AC2 | K:
      k2A, E: e2A schedule(event(oid,effect)),
    M: < oid : AC1 | K: k1b, E: e1A, R: m1A, A1b > , A2A >
if < oid : AC2 | K: k2A, E: e2A,
    M: < oid : AC1 | K: k1A, E: e1A, R: m1A, A1A > , A2A > :=
  computeAdaptationPhase( < oid : AC2 | K: k2 , E: e2 ,
    M: < oid : AC1 | K: k1 , E: e1 , R: m1 , A1 > , A2 > )
/\ < oid : AC1 | K: k1b, E: e1A effect, A1b > := execute(
  < oid : AC1 | K: oneStep update2To1(k2A,k1A), E: e1A, A1A > , m1A ) .

```

The rule is triggered by the token `nextEffect`, generated by the orchestrator, and propagated by the cell containing the robot. The execution of the rule consists of an adaptation phase followed by an execution phase, both on the managed component. The two phases are triggered by the two sub-conditions of the rule.

The adaptation phase is computed by the operation `computeAdaptationPhase`, using the knowledge of layer 2 (`k2`) to enact a state transition, if necessary. Among those defining the operation, the equation below encodes the transition of Fig. 11 from state `Aggregate` to state `Self_Assemble`, labeled with `Close` to `red`

```

ceq [AggToSA]: computeAdaptationPhase(
  < oid2 : AC2 | K: state(Aggregate)    k2, E: e2
    M: < oid1 : AC1 | R: m1 , E: e1 , A1 > , A2 >)
= < oid2 : AC2 | K: state(SelfAssemble) k2, E: emitt(green),
  M: < oid1 : AC1 | R: m1b, E: none, A1 > , A2 >
if seeEffect(led(red),k2)
/\ m1b := upModule('AC1-SELF_ASSEMBLE,false) .

```

The conditional equation states that if a robot in state `Aggregate` sees in its neighborhood a robot with red LEDs on, then it must pass to state `Self_Assemble` and turn on its green LEDs. Also the rules of the managed component are changed: the new module `m1b` is obtained with the operation `upModule`, producing the meta-representation of the Maude module passed as first parameter.

We specified one equation for each transition of Fig. 11 plus the following one where `owise` is a special attribute that tells the interpreter to apply the equation only if none of the others is applicable

```

eq [idle]: computeAdaptationPhase(obj) = obj [ owise ] .

```

Once the adaptation phase is concluded, the second sub-condition of rule `adaptAndExecute` takes care of the one step execution of the (possibly adapted) managed component, using operation `execute`. Finally, the effects generated by layer 1 are wrapped in the constructors `event` and `schedule`, and are added to the effects of layer 2, so that the cell containing it will propagate it to the scheduler.

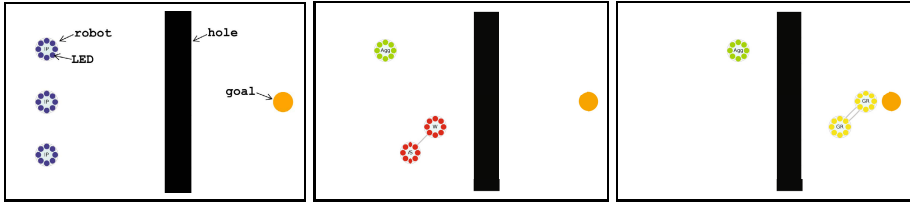


Fig. 7. Three states of a simulation: initial (left), assembly (middle), final (right)

6 Analysis of Adaptation Strategies

This section describes some of the analysis activities carried out with our implementation, available at <http://sysma.lab.imtlucca.it/tools/ensembles> together with some additional material such as animated simulations.

The analysis has been carried out in two phases: (§6.1) discrete event simulation; and (§6.2) statistical model checking. The rationale is the following.

In the early development phases we have mainly concentrated on performing single simulations that have been informally analyzed by observing the behavior of the assemblies in the automatically generated animations. A couple of trial-and-error iterations (where the model was fixed whenever some anomalous behavior was spotted) were enough for the model to acquire sufficient maturity to undergo a more rigorous analysis in terms of model checking.

Ordinary model checking is possible in the Maude framework (via Maude’s reachability analysis capabilities, or LTL model checker) but suffers from the state explosion problem and is limited to small scenarios and to *qualitative* properties. To tackle larger scenarios, and to gain more insight into the probabilistic model by reasoning about *probabilities* and *quantities* rather than *possibilities*, we resorted to statistical model checking techniques.

We now provide the details of these analysis phases, centered around one crucial question: *How many robots reach the goal by crossing the hole?*

6.1 Simulations

Simulations are performed thanks to the discrete-event simulator mentioned in §5.2 along the lines of the ones reported in [11,20]. Valuable help has been obtained implementing an exporter from Maude `Configuration` terms to DOT graphs¹, offering the automatic generation of images from states: they have greatly facilitated the debugging of our code.

Fig. 7 illustrates three states of a simulation in which robots execute the *basic self-assembly response strategy*. The initial state (left) consists of three robots (grey circles with small dots on their perimeter) in their initial state (emitting blue light), a wide hole (the black rectangle) and the goal of the robots, i.e. a source of light (the orange circle on the right). After some steps, where the robots execute the self-assembly strategy, two get assembled (middle of Fig. 7).

¹ DOT is a well-established graph description language (<http://www.graphviz.org/>).

The assembled robots can then safely cross the hole and reach the goal (right of Fig. 7), while the unassembled one is abandoned in the left part of the arena.

While performing such simulations with different scenarios, varying the location of the goal and number and distribution of the robots, and with different parameters for duration of timeouts and actions, we observed several *bizarre* behaviors. For instance, in various simulations we observed some unassembled robots erroneously believing to be part of an assembly, moving into the hole and disappearing. In other simulations we instead noticed pairs of robots grabbing each other. These observations triggered the following questions: *Is there an error in our implementation? Is there an error in the strategies defined in [16]?*

Examining carefully the description of the strategy, we discovered that the two behaviors are indeed not explicitly disallowed in [16] and originated by the two transitions (see Fig. 1 in §2) outgoing from the state `Assembly_Seed` (willing to be grabbed). The first transition leads to state `Wait`, triggered by the expiration of a timeout, while the second one leads to state `Aggregate` (willing to grab), triggered by the event `See red` (i.e. another robot willing to be grabbed). Considering the first behavior, a robot can change from state `Assembly_Seed` to state `Wait` even if no other robot is attached to it. The robot then evolves to state `Connected_phototaxis` believing to be assembled with other robots. Considering instead the second behaviour, once a robot i grabs a robot j , i becomes itself “willing to be grabbed” (turning on its red LEDs) to allow other robots to connect to the assembly. Now, it is clear that if j is grabbed while being in state `Assembly_Seed`, then its transition towards state `Aggregate` is allowed, leading to the second bizarre behaviour. Interestingly enough, we hence notice that the two bizarre behaviors strongly depend on the duration of the timeout: a short one favors the first behaviour, while a long one favors the second one.

Are these behaviors actually possible in real robots or are they forbidden by real life constraints (e.g. due to the physical structure of the robots or to some real-time aspects)? The answer to this question is being investigated within the ASCENS project [4]. However, our experience makes it evident that the self-assembly strategies described in [16] might be adequate for s-bots but not in general for self-assembly settings where other constraints might apply. Fortunately, both bizarre behaviors can be fixed easily by adding further conditions to the two mentioned transitions of the adaptation strategy. In particular, the transition from `Assembly_Seed` to `Aggregate` requires a further condition to ensure that the robot has been gripped. Conversely, the transition from `Assembly_Seed` to state `Aggregate` requires exactly the contrary, i.e. the robot must not be gripped.

6.2 Statistical Model Checking

A qualitative analysis can prove that an assembly strategy can result in different degrees of success, from full success (all robots reach the goal) to full failure (no robot reaches the goal). However, in the kind of scenario under study different levels of success are typically of interest. The really interesting question is how *likely* are they? Moreover, another interesting measure could be the *expected number* of robots reaching the goal.

An analysis based on statistical model checking (see e.g. [19,20,2]) is more appropriate in these cases. Such techniques do not yield the absolute confidence of qualitative model checking but allow to analyze (up to some statistical errors and at different levels of confidence) larger scenarios and to deal with the stochastic nature of probabilistic systems.

We consider the following properties: (P_0) *What is the probability that no robot reaches the goal?*; (P_1) *What is the probability that at least one robot reaches it?*; and (P_2) *What is the expected number of robots reaching the goal?*

We have used PVesta [2], a parallel statistical model checker, to perform some comparative analysis. The tool performs a statistical evaluation (Monte Carlo based) of properties expressed in the transient fragments of PCTL and CSL, and of quantitative temporal expressions (QuaTEx) [1], allowing to query about expected values of real-typed expressions of a probabilistic model.

We have performed a comparative analysis (w.r.t. the above properties) between two different strategies: namely the original *basic self-assembly response* and the variant that fixes the bizarre behaviors discussed above. For each experiment, where we fixed 120 as maximum number of system steps, all robots execute the same strategy. The aim of this preliminary assessment was neither to compare different strategies, nor to derive exact statistical measures, but to gain some intuition of the success and performance impact of the absence of the bizarre behaviors. The arena was configured as follows (cf. Fig. 8): an 11×7 grid containing 3 robots, the goal (a source of light) and a hole dividing the robots from the goal. We remind that a robot alone is not able to cross the hole, and hence needs to cooperate (assemble) with other robots to cross it.

Roughly, our variant of the strategy exhibits a better success rate. More precisely, the analysis of P_0 on the original strategy provides 0.48 (i.e. about half of the cases ends up without any robot reaching the goal), while for our variant we obtain 0.36. Regarding P_1 , our variant exhibits again a better rate (0.64) than the original one (0.52). Finally, the expected number of successful robots (P_2) is 1.07 in the original case, while in our variant case it is 1.38.

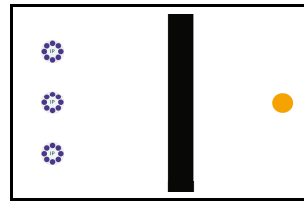


Fig. 8. An initial state

These preliminary data of the statistical analysis seem to confirm our intuition. Forthcoming experiments will consider other robot features and strategies, and will validate our results against the ones reported in [16].

A Sample QuaTEx Expression. We conclude this section discussing the quantitative temporal expression we defined to estimate the expected number of robots reaching the goal.

QuaTEx is a language to query quantitative aspects of probabilistic systems. Exactly as temporal logics allows to express temporal formulae, QuaTEx allows to write quantitative temporal expressions.

PVesta statistically evaluates quantitative temporal expressions w.r.t. two parameters: α and δ . Specifically, expected values are computed from n independent simulations, with n large enough to grant that if a QuaTE_X expression is estimated as \bar{x} , then, with probability $(1 - \alpha)$, its actual value belongs to the interval $[(1 - \delta)\bar{x}, (1 + \delta)\bar{x}]$. For the experiments in §6.2 we fixed $\alpha = \delta = 0.05$.

In the rest of this section we see how the mentioned QuaTE_X expression has been defined, and how its value is actually computed for single simulations. We do not detail how PVesta performs one-step executions, since this is out of the scope of this paper. Details can be found in [1].

Before defining our expression it is necessary to define real-typed Maude operations representing the states predicates we are interested in. We defined the state predicate `completed : Configuration -> Float`, reducing to 1.0 for terminal states, and to 0.0 otherwise. A terminal state is a state with no more robots, a state with all the robots in goal, or the state obtained after a given maximum number of steps. We also defined the state predicate `countRobotInGoal : Configuration -> Float`, counting the number of succesful robots.

Then we defined the equations necessary to PVesta to access such predicates (where `C` is a variable with sort `Configuration`): `eq val(0,C) = completed(C)`, and `eq val(1,C) = countRobotInGoal(C)`. Actually, QuaTE_X syntax requires to indicate the term “`val(n,s)`” with “`s.rval(n)`”, where `n` and `s` are respectively terms with sort `Natural` and `Configuration`.

Finally, the QuaTE_X expression to estimate the expected number of robots reaching the goal is easily expressed as

```
count_s-bots_in_goal() = if { s.rval(0) == 1.0 } then s.rval(1)
                        else #count_s-bots_in_goal() fi;
eval E[ count_s-bots_in_goal() ] ;
```

Informally, a QuaTE_X expression consists in a list of definitions of recursive temporal operators, followed by a query of the expected value of a path expression obtained (arithmetically) combining the temporal operators. Our formula defines the temporal operator `count_s-bots_in_goal()`, which also corresponds to the estimated path expression `eval E[count_s-bots_in_goal()]`.

The path expression is evaluated by PVesta in the initial state (`s`) of the system (e.g. the state depicted in Fig. 8). The tool first evaluates the guard of the `if_then_else` statement, i.e. `s.rval(0) == 1.0`. The condition reads as “is the state predicate `rval(0)` equal to 1.0 if evaluated in the state `s`?”, and corresponds to “is the current state a final state?”. If the guard is evaluated to true, then the path expression is evaluated as `s.rval(1)`, that is in the number of robots that reached the goal in the state `s`. If the guard is evaluated to false, then the path expression is evaluated as `#count_s-bots_in_goal()`, read “evaluate `count_s-bots_in_goal()` in the state obtained after one step of execution”. The symbol `#`, named “next”, is in fact a primitive temporal operator.

To conclude, the evaluation of the QuaTE_X expression consists in performing step-wise system simulations, and is evaluated as the (mean of the) number of robots that reached the goal in the terminal states of each simulation.

7 Conclusion

The contributions of our paper are: (i) a description (§4, §5) of how to realize in Maude our recently proposed approach to adaptive systems [6] in a simple and natural way; and (ii) a description (§6) of how to exploit the Maude toolset for the analysis of our models, and PVesta [2] in particular.

Our work is inspired by early approaches to coordination and adaptation based on distributed object reflection [14,21] and research efforts to apply formal analysis onto such kind of systems (e.g. [13]), with a particular focus on adaptive systems (e.g. [22,4]). Among those, the PAGODA project [22] is the closest in spirit and shape. Our work is original in its clear and neat representation and role of *control data* in the architecture, and in the fact that this is, as far as we know, the first analysis of self-assembly strategies based on statistical model checking.

The case study of self-assembly strategies for robot swarms [16] has contributed to assess our approach. Overall, the conducted experimentation demonstrates that Maude is well-suited for prototyping self-assembly systems in early development phases, and that the associated simulation can be useful to discover and resolve small ambiguities and bugs in self-assembly strategies. Furthermore, statistical model checking can provide preliminary estimations of success rate, that can be used to compare different strategies and also to validate/confute/refine analogous measures provided by other tools or in real world experiments.

We plan to further develop our work by considering other case studies, more realistic abstractions and more modular implementations. However, the key challenging question we want to tackle is: *can we exploit the proposed architecture to design smarter adaptation strategies or to facilitate their analysis?* We envision several interesting paths in this regard. First, we are investigating how logical reflection can be exploited at each layer of the architecture, for instance to equip components with dynamic planning capabilities based on symbolic reachability techniques. Second, we are developing a compositional reasoning technique that exploits the hierarchical structure of the layered architecture.

All in all, we believe that our work is a promising step towards the non-trivial challenges of building predictive adaptive systems, and to analyze them.

Acknowledgements. We are grateful to the anonymous reviewers for their fruitful criticisms and to the organizers of the *AWASS 2012 summer school* for the opportunity to mentor a case study based on the experience of this paper.

References

1. Agha, G.A., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. In: Cerone, A., Wiklicky, H. (eds.) QAPL 2005. ENTCS, vol. 153(2), pp. 213–239. Elsevier (2006)
2. AlTurki, M., Meseguer, J.: PVESTA: A Parallel Statistical Model Checking and Quantitative Analysis Tool. In: Corradini, A., Klin, B., Cirstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 386–392. Springer, Heidelberg (2011)

3. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Reflecting on self-adaptive software systems. In: SEAMS 2009, pp. 38–47. IEEE Computer Society (2009)
4. Autonomic Service Component Ensembles (ASCENS), <http://www.ascens-ist.eu>
5. Broy, M., Leuxner, C., Sitou, W., Spanfelner, B., Winter, S.: Formalizing the notion of adaptive system behavior. In: Shin, S.Y., Ossowski, S. (eds.) SAC 2009, pp. 1029–1033. ACM (2009)
6. Bruni, R., Corradini, A., Gadducci, F., Lluch Lafuente, A., Vandin, A.: A Conceptual Framework for Adaptation. In: de Lara, J., Zisman, A. (eds.) Fundamental Approaches to Software Engineering. LNCS, vol. 7212, pp. 240–254. Springer, Heidelberg (2012)
7. Cabri, G., Puviani, M., Zambonelli, F.: Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles. In: Smari, W.W., Fox, G.C. (eds.) CTS 2011, pp. 508–515. IEEE Computer Society (2011)
8. 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)
9. Horn, P.: Autonomic Computing: IBM’s perspective on the State of Information Technology (2001)
10. IBM Corporation: An Architectural Blueprint for Autonomic Computing (2006)
11. Karsai, G., Sztipanovits, J.: A model-based approach to self-adaptive software. *Intelligent Systems and their Applications* 14(3), 46–53 (1999)
12. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *Computer* 36(1), 41–50 (2003)
13. Meseguer, J., Sharykin, R.: Specification and Analysis of Distributed Object-Based Stochastic Hybrid Systems. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 460–475. Springer, Heidelberg (2006)
14. Meseguer, J., Talcott, C.: Semantic Models for Distributed Object Reflection. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 1–36. Springer, Heidelberg (2002)
15. Mondada, F., Pettinaro, G.C., Guignard, A., Kwee, I.W., Floreano, D., Deneubourg, J.L., Nolfi, S., Gambardella, L.M., Dorigo, M.: Swarm-bot: A new distributed robotic concept. *Autonomous Robots* 17(2-3), 193–221 (2004)
16. O’Grady, R., Groß, R., Christensen, A.L., Dorigo, M.: Self-assembly strategies in a group of autonomous mobile robots. *Autonomous Robots* 28(4), 439–455 (2010)
17. Pavlovic, D.: Towards Semantics of Self-Adaptive Software. In: Robertson, P., Shrobe, H.E., Laddaga, R. (eds.) IWSAS 2000. LNCS, vol. 1936, pp. 50–64. Springer, Heidelberg (2001)
18. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems* 4(2), 1–42 (2009)
19. Sen, K., Viswanathan, M., Agha, G.: On Statistical Model Checking of Stochastic Systems. In: Etesami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 266–280. Springer, Heidelberg (2005)
20. Sen, K., Viswanathan, M., Agha, G.A.: Vesta: A statistical model-checker and analyzer for probabilistic systems. In: Baier, C., Chiola, G., Smirni, E. (eds.) QEST 2005, pp. 251–252. IEEE Computer Society (2005)
21. Talcott, C.L.: Coordination models based on a formal model of distributed object reflection. In: Brim, L., Linden, I. (eds.) MTCoord 2005. ENTCS, vol. 150(1), pp. 143–157. Elsevier (2006)

22. Talcott, C.L.: Policy-based coordination in PAGODA: A case study. In: Boella, G., Dastani, M., Omicini, A., van der Torre, L.W., Cerna, I., Linden, I. (eds.) CoOrg 2006 & MTCoord 2006. ENTCS, vol. 181, pp. 97–112. Elsevier (2007)
23. Weyns, D., Malek, S., Andersson, J.: FORMS: a formal reference model for self-adaptation. In: Figueiredo, R., Kiciman, E. (eds.) ICAC 2010, pp. 205–214. ACM (2010)
24. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) ICSE 2006, pp. 371–380. ACM (2006)

Formal Modeling and Analysis of Human Body Exposure to Extreme Heat in HI-Maude

Muhammad Fadlisyah¹, Peter Csaba Ölveczky¹, and Erika Ábrahám²

¹ University of Oslo, Norway

² RWTH Aachen University, Germany

Abstract. In this paper we use HI-Maude to model and analyze the human thermoregulatory system and the effect of extreme heat exposure on the human body. This work is motivated by the 2010 Sauna World Championships, which ended in a tragedy when the last two finalists were severely burnt in surprisingly short time (one of them died the next day). HI-Maude is a recent rewriting-logic-based formal modeling language and analysis tool for complex hybrid systems whose components influence each others' continuous dynamics. One distinguishing feature of HI-Maude is that the user only needs to describe the continuous dynamics of *single* components and interactions, instead of having to explicitly define the continuous dynamics of the entire system. HI-Maude analyses are based on numerical approximations of the system's continuous behaviors. Our detailed models of human thermoregulation and the sauna used in the world championships allow us to use HI-Maude to formally analyze how long the human body can survive when experiencing extreme conditions, as well as analyzing possible explanations for the still unsolved tragedy at the 2010 Sauna World Championships.

1 Introduction

Experimentation on humans might be the best way to understand the human body and mind, and can provide enormously important medical, scientific, and psychological knowledge for the good of society. However, experimentation on humans is typically quite costly, does (hopefully) not allow studying the body's reaction to extreme stress, is ethically fraught, and has a history with some very dark episodes. Although animals can sometimes replace humans in such experiments, they often function quite differently than humans, and experimentation on animals must take the growing animal rights sentiment into account.

Computer-based simulation and analysis can be a cheap and useful way to study the human body's reaction to stimuli – even fairly extreme stress that would be unethical to perform on a human. The human thermoregulatory system is an important part of the human body, as it tries to keep the person at a comfortable temperature even in difficult environments. Understanding the human thermoregulatory system is crucial to understand how our body will

respond to hostile and extreme environments, such as when firefighting, mining, deep-sea diving, traveling in space, or just practicing sports.¹

Defining useful computer models of the human thermoregulatory system is a challenging task. We need to model as closely as possible the complex continuous dynamics of the various parts (skin, core, blood, etc.) as well as the complex continuous interactions between these parts, and between the body and its environment. We may also want to model behavioral, and therefore nondeterministic, aspects of the thermoregulatory system. Finally, we must be able to account for changing configurations, such as when a human jumps from an oppressively hot sauna into the cold snow; in this case, the thermal interactions change instantly, and the continuous dynamics of the entire system must be recomputed.

In this paper we use the rewriting-logic-based HI-Maude language and tool [8] to model the human thermoregulatory system according to established medical/physiological models and to analyze it under extreme conditions. In particular, our investigation is motivated by the 2010 Sauna World Championships, where both finalists collapsed after surprisingly short time. We therefore also present a fairly detailed model of the thermodynamics of a sauna, and use the HI-Maude tool to analyze how long people in different states of fitness can survive in different kinds of saunas and to analyze possible explanations for the tragedy in 2010 (the cause of which is still unknown).

HI-Maude is a recent extension of Real-Time Maude [13] to support the object-oriented formal modeling, simulation, and model checking of *hybrid systems* with combined discrete and continuous behaviors. The tool targets large and complex hybrid systems that typically have multiple physical entities that interact and influence each other's continuous behavior. For a thermal systems example, consider a cup of hot coffee which interacts with the surrounding room through different kinds of heat transfer, leading to a decrease in the coffee temperature and to a slight increase in the room temperature. One distinguishing feature of HI-Maude is the *modularity* and *compositionality* of the specification of the system's continuous dynamics. Non-compositional specification of the *whole* system is very hard, as it involves combining the ordinary differential equations (ODEs) that specify the dynamics of its components; it also requires redefining the system's continuous dynamics for each new configuration of interacting physical components. To achieve the desired modularity and compositionality, HI-Maude offers an object-oriented modeling methodology [5] that allows us to specify the continuous dynamics of *single physical entities* (such as the cup of coffee and the room) and of *single physical interactions* (such as thermal conduction and convection). Not only does this make it easier to specify the continuous dynamics of a system of interacting physical components, but it also means that the specification does not need to be redefined for each new configuration of physical entities. If we want to add a cup of coffee to the room, we just add a new coffee object and appropriate physical interaction objects to the state.

To analyze the system, whose continuous dynamics is usually defined by ordinary differential equations that are not analytically solvable, HI-Maude uses

¹ Heat stroke is the third leading cause of deaths among athletes in the U. S. [18].

adaptations of different numerical methods (the Euler method and Runge-Kutta methods of different order) to give fairly precise approximate solutions to coupled ordinary differential equations. These approximations are then used in HI-Maude simulation, reachability analysis, and linear temporal logic model checking.

In this paper we follow as much as possible established medical facts and models when defining our own model. The reference [11] gives an overview of some approaches to model the human thermoregulatory system. We choose the two-node Gagge model [9] as the basic model of the human body, since much scientific and engineering research on human thermoregulation is based on this model. For the formulas used to model the physiological aspects of the human thermoregulatory system, our main sources are [14,1]. We use [12] as the main source for some physics-related equations for modeling aspects of the interaction between the human body and its environment, and use [11,10] as main sources for modeling the behavioral aspect of human thermoregulatory system. Our main sources on how to model experimental subjects in different physiological conditions and degrees of preparedness for this competition are [16,2,15,14].

We have tried to model the sauna as closely as possible to the one used in the Sauna World Championships. Since we have not found any official description about the sauna used in that event, we rely on information gathered from stories, photographs, and videos available on the web, and from technical specifications of the equipment (e.g. the heater) and physical properties of the material used (e.g., the heat capacity of the rocks used for the heating). We use information from [4] for some physical properties of the environment.

In [8] we introduce the HI-Maude tool and illustrate its use on a fairly simple “proof-of-concept” model of a few aspects of the human thermoregulatory system. The model presented in this paper is completely different, and aims at being a fairly detailed model of the human thermoregulatory system and its interactions with equally faithfully modeled environments, including sophisticated models of the thermodynamics of different kinds of saunas. Just to mention a few differences: the model in [8] only considers the temperature of the body *core* (and hence only problems of hypothermia and hyperthermia), whereas this paper also takes into account the effect on the skin of exposure to heat (burn injuries) and the hydration state of the body (dehydration), which also needs a more refined model of the sweating rate; in [8], the blood vessels are modeled by three discrete states, whereas in this paper their dynamics is continuous; in [8], the dynamics of the heat flow by convection and radiation only considers the temperature difference between the body skin and the air temperature, whereas in this paper, the heat flow dynamics also considers the humidity factor of the surrounding environment, which also changes continuously because water poured periodically on the heating rocks during the sauna event will increase the humidity of the environment (and significantly increase the stress on the body, and is a crucial parameter that must be taken into account to seriously analyze the sauna accident); the model in [8] does not include heat exchange through respiration, whereas this paper includes the heat exchange through the air inhaled and exhaled during respiration; in [8] we only consider the physiological

aspects of the human thermoregulatory system, whereas the model presented in this paper considers both physiological and behavioral aspects; the model in [8] considered the body to have the form of a cylinder when computing the area of the skin, whereas we now use the Dubois equation to approximate the size of the area of the body; and so on. As for the environment, in [8] the model is very simple and only considers that the room is filled with air (with a controlled heater to keep the maximum temperature), whereas in the current case study we model the sauna room with realistic parameters used in the event: our model of the sauna includes the heater, the heating rocks, and the periodic pouring of water onto the heating rocks, which increases heat to the skin and decreases the ability of the body to lose heat by evaporation. In this paper we also model three kinds of persons: a normal person, a person who has practiced in such conditions (like the contestants in the Sauna World Championships), and an unhealthy person. In short, the current model is incomparably more faithful and detailed and includes many more aspects needed to be able to analyze the sauna accident with a reasonable degree of accuracy.

We can only provide a sampler of our model in a short paper. The entire executable formal HI-Maude model, the analysis commands, a long report, and the HI-Maude tool itself, are available at <http://folk.uio.no/mohamf/HI-Maude>.

Section 2 briefly introduces the HI-Maude tool and the effort/flow-based modeling methodology upon which it is based. Section 3 gives an overview of the human thermoregulatory system. Section 4 presents some parts of our model of the human thermoregulatory system. Finally, Section 5 uses HI-Maude to analyze how long humans can stay safely in saunas and tries to understand what happened at the 2010 Sauna World Championships.

2 Modeling and Analysis of Hybrid Systems in HI-Maude

This section gives a brief overview of the HI-Maude tool and the modeling methodology in [5], upon which the tool is based, which adapts the *effort/flow* method [17] to model a physical system as a network of *physical entities* and *physical interactions* between the entities.

2.1 Effort/Flow Modeling of Interacting Hybrid Systems

In effort/flow modeling of a physical system, a *physical entity* is described by a real-valued *effort* value, a set of *attribute* values, and the entity's *continuous dynamics* (see Fig. 1, top left). The effort variable represents a dynamic physical quantity, such as temperature, that evolves over time as given by the continuous dynamics in the form of an ordinary differential equation (ODE), where its time derivative is a function of both the entity's attribute values *and* the flows of connected interactions (i.e., the time derivative \dot{e} of the effort e can be described by an equation of the form $\dot{e} = f(\sum \text{flows}, \text{atts})$).

A *two-sided interaction* between two physical entities is described by a real-valued *flow*, a set of *attribute* values, and a *continuous dynamics*. The flow value

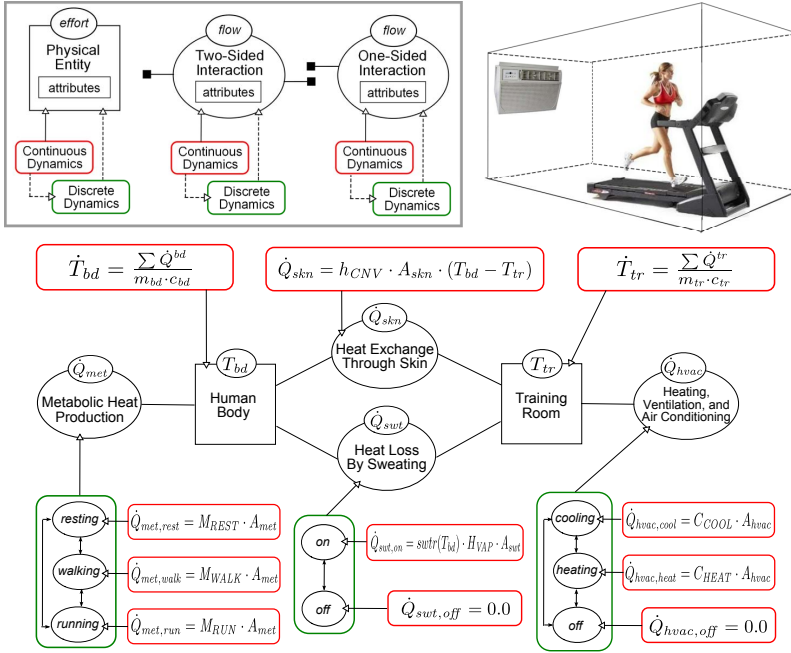


Fig. 1. A simple effort/flow model of the thermodynamics of a person in a gym

describes the dynamic interaction between two entities, whose evolution over time is specified by the continuous dynamics as an equation with the flow variable on the left-hand side and an expression referring to the interaction's attributes and the efforts of the connected entities on the right-hand side (i.e., $flow = g(effort_1, effort_2, atts)$). A *one-sided interaction* represents an interaction of a physical entity with its environment. The system may also exhibit discrete dynamics, for representing, e.g., the changes of physical states of the system components, explicit control behaviors, communication, etc.

Figure 1 illustrates our modeling methodology on a simplified *thermal* system consisting of a woman working out at a gym. There are two *physical entities* of interest for thermal reasoning: the human body and the training room, both with the temperature T as their effort variable. The body produces heat, and the heat production increases as the exercise gets harder. The body releases heat to the room through the skin (e.g., by convection), and through sweating (heat is released from the body as the sweat evaporates). These heat transfers are represented as *two-sided interactions* where the flow variable (\dot{Q}) denotes the heat flow rate of the interactions. The *one-sided interaction* is used to model the heat production inside the human body through metabolism, and also to model the system which handles heating and cooling of the gym. Beside continuous dynamics, there are some physical phenomena which are suitably modeled as discrete dynamics, e.g., the changes of activity during the training (from running, to walking, to resting), activation and deactivation of sweating, and so on.

2.2 The HI-Maude Tool

The HI-Maude tool [8] supports the object-oriented effort/flow modeling and approximation-based formal analysis of interacting hybrid systems. Since we target complex systems and therefore do not restrict to *linear* ODEs for describing the continuous dynamics of a component/interaction, the continuous dynamics of a system is in general not analytically solvable. HI-Maude therefore uses numerical techniques to *approximate* the continuous behaviors by advancing time in small discrete time increments, and approximating the values of the continuous variables at each “visited” point in time. We have adapted the *Euler* [5], the *Runge-Kutta 2nd order* (RK2), and the *Runge-Kutta 4th order* (RK4) methods [7] to the effort/flow framework. Once the dynamics of the single physical components has been defined, HI-Maude

1. automatically defines the continuous dynamics of the entire systems, and
2. provides the usual Real-Time Maude formal analysis commands, but where the desired built-in approximation algorithm and the desired time increments used by the approximations are additional parameters of the commands.

Modeling. Since HI-Maude is an extension of Maude [3], a *membership equational logic* [3] theory (Σ, E) , with Σ a signature² and E a set of *conditional equations* and *memberships*, specifies the system’s state space as an algebraic datatype. The system’s instantaneous transitions are specified by a set R of (possibly conditional) *labeled instantaneous rewrite rules* `cr1 [l] : t => t' if cond`, where l is a *label*, t and t' are two Σ -terms, and the condition *cond* is a conjunction of equations, memberships, and rewrites. We refer to [3] for more details on the syntax of Maude.

A declaration `class C | att1 : s1, ..., attn : sn` declares a *class* C with attributes att_1 to att_n of sorts s_1 to s_n . An *object* of class C is represented as a term `< O : C | att1 : val1, ..., attn : valn >` of sort `Object`, where O , of sort `Objid`, is the object’s *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n . The state is a term of sort `Configuration`, and has the structure of a *multiset* of objects and messages, with multiset union denoted by a juxtaposition operator that is declared associative and commutative, so that rewriting is *multiset rewriting* supported in Maude.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule

```
r1 [1] : < 0 : C | a1 : 0, a2 : y, a3 : w, a4 : z > =>
        < 0 : C | a1 : T, a2 : y, a3 : y + w, a4 : z >
```

defines a parametrized family of transitions which can be applied whenever the attribute `a1` of an object `0` of class `C` has the value `0`, with the effect of altering the attributes `a1` and `a3` of the object. “Irrelevant” attributes (such as `a4`, and the *right-hand side occurrence* of `a2`) need not be mentioned in a rule (or equation).

² i.e., Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols*.

A *subclass* inherits all the attributes and rules of its superclasses.

HI-Maude provides built-in classes for specifying physical entities and interactions. Concrete physical entities and interactions must then be defined as object instances of user-defined subclasses of these built-in classes. For modeling physical entities, the class `PhysicalEntity` is used:

```
class PhysicalEntity | effort : Float .
```

Sometimes we need additional continuous variables whose dynamics are time-derivative functions. The tool therefore provides the classes `PhysicalEntityAck`, where k denotes the number of additional continuous variables³

```
class PhysicalEntityAck | contvar1 : Float , ... , contvar $k$  : Float .
subclass PhysicalEntityAc1 ... PhysicalEntityAc $n$  < PhysicalEntity .
```

The attributes `contvari` denote the additional continuous variables.

Objects of the classes `TwoSidedInteraction` and `OneSidedInteraction` are used to model two-sided and one-sided physical interactions, respectively:

```
class PhysicalInteraction | flow : Float, contdyntype : ContDynType .
class TwoSidedInteraction | entity1 : Oid, entity2 : Oid .
class OneSidedInteraction | entity : Oid .
subclass TwoSidedInteraction OneSidedInteraction < PhysicalInteraction .
```

The `contdyntype` attribute denotes the type of continuous dynamics specified for the interaction. The `entity1` and `entity2` attributes denote the two physical entities involved in the two-sided interaction. The `entity` attribute of the class `OneSidedInteraction` denotes the entity interacting with the environment.

HI-Maude requires the user to define the continuous dynamics by defining the function `effortDyn` for each physical entity:

```
op effortDyn : Object Float -> Float .
```

The first argument of `effortDyn` is the entity object itself; the second argument is the sum of the values of the flows to/from the entity, and is provided by the tool. `effortDyn(object, $\sum \dot{Q}$)` therefore defines the time derivative of the effort variable of the object. That is, if $\dot{e} = f(\sum flows, atts)$, then we define `effortDyn(< O : C | atts >, X) = f(X, atts)`.

For physical entities with additional continuous variable(s), the functions `contvariDyn` define the continuous dynamics of those variables:

```
ops contvar1Dyn ... contvar $n$ Dyn : Object Configuration -> Float .
```

The first argument of the function is the entity object itself; the second argument is the entire multiset of objects in the system.

The function `flowDyn` defines the continuous dynamics of the physical interactions. To define the continuous dynamics of, respectively, two-sided interactions and one-sided interactions, the following formats are used:

³ The tool currently provides the entity class with two additional continuous variables.

```
op flowDyn : Object Float Float -> Float .
op flowDyn : Object Float -> Float .
```

The first argument of the function is the interaction object itself. The second (and third) arguments are the effort variable values of the interacting physical entity/entities. Sometimes attribute values from objects that are not directly related to the interaction must be used to define the continuous dynamics of an interaction, in which case the following function is used:

```
op flowDyn : Object Configuration -> Float .
```

The second argument is the entire multiset of objects in the system.

Discrete transitions are modeled as rewrite rules. To ensure that such a rule is applied in a timely manner, HI-Maude provides the function

```
op timeCanAdvance : Configuration -> Bool .
```

so that if the user does *not* want time to advance when an object is in a certain state, (s)he must define `timeCanAdvance` to be `false` for those object states.

Formal Analysis. HI-Maude extends Real-Time Maude’s analysis commands by allowing the user to select: (i) the numerical approximation technique used to approximate the continuous behaviors, (ii) the time increment used in the approximation, and (iii) discrete-switch-detection-based adaptive time increments. If we use a fixed time increment in the approximations, we may “miss” the time when the (approximated) effort value is such that a discrete event should take place (e.g., the body should go to state *hyperthermia* when its temperature reaches $38.9^{\circ}C$). We can therefore also use *adaptive* time increments in connection with the Euler method to stop time advance exactly when a given continuous attribute has a desired value.

HI-Maude’s hybrid *rewrite* command is used to simulate one behavior of the system from an initial state *initState* up to a certain duration *timeLimit*:

```
(hrew initState in time ~ timeLimit using numMethod stepsize stepSize
  discretewitch dswitchType .)
```

\sim is either ‘<=’ or ‘<’; *numMethod* \in {`euler`, `rk2`, `rk4`} is the numerical method used to approximate the continuous behaviors; *stepSize* is the time increment used in the approximation of the continuous behaviors; and (if *numMethod* is `euler`) *dswitchType* is `accurate` if adaptive step size should be used to stop time exactly when a discrete event must take place, and is `nonaccurate` otherwise.

HI-Maude’s search command searches for up to *n* states that are matched by a search pattern with a substitution that satisfies an (optional) condition and that can be reached from an initial state in a given time interval:

```
(hsearch [n] initState =>* searchPattern [such that cond] in time ~ timeLimit
  using numMethod stepsize stepSize discretewitch dswitchType .)
```

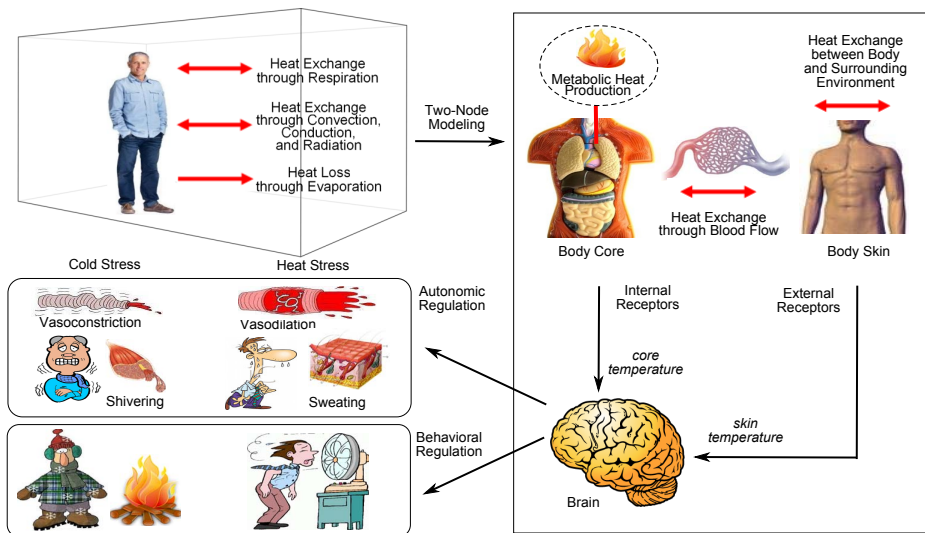


Fig. 2. The human thermoregulatory system

where $\sim \in \{<, \leq, >, \geq\}$, and *cond* is a condition on the variables in the search pattern. The following command finds the shortest time needed to reach a state:

```
(hfind earliest init => * pattern [such that cond] using numMethod stepsize sSize
discreteswitch dswitchType .)
```

Finally, HI-Maude’s model checker extends Real-Time Maude’s explicit-state time-bounded linear temporal logic model checker in the same way. The time-bounded hybrid model checking command is written with syntax

```
(hmc initState |t formula in time ~ timeLimit using numMethod stepsize sSize
discreteswitch dswitchType .)
```

3 The Human Thermoregulatory System

The human body needs to maintain a body temperature of around $37^{\circ}C$ to function normally. The metabolic heat production within the body is the only internal factor affecting body temperature of a healthy person. The environment surrounding the body affects the body temperature by heat loss or gain through physical processes such as radiation, evaporation, convection, and conduction. *Hyperthermia* and *hypothermia* occur, respectively, when the body temperature increases, resp. decreases, significantly beyond normal.

Physiological and *behavioral* thermoregulation respond to changing environments in an attempt to ensure human survival and comfort. The primary control center of physiological thermoregulation is located in a part of the brain called the hypothalamus. The hypothalamus enables mechanisms to support heat loss

from the body when the body temperature is increasing above normal levels that include: increasing the diameter of blood vessels to let more blood flow underneath the skin (*vasodilation*), which promotes heat loss by radiation, convection, and conduction; and increasing sweat production, which promotes heat loss by evaporation. When the body temperature is decreasing, the hypothalamus enables the following mechanisms to reduce heat loss and increase heat production: decreasing the diameter of blood vessels to let less blood flow underneath the skin (*vasoconstriction*), and stimulating the skeletal muscles to cause shivering, which increases heat production by the body. *Behavioral* response to heat or cold stress include taking off clothes or switch on a fan when the temperature is felt to be too hot, and putting on more clothes or moving closer to the fireplace when it feels too cold. Behavioral thermoregulation is related to a part of the brain called the cerebral cortex.

The Two-Node Modeling Approach. In the two-node Gage model [9] the human body is considered as consisting of two concentric layers where the inner layer is the central core, and the outer layer is the skin shell. Heat exchange between the body and the environment takes place continuously at the skin surface. Heat generated inside the body is transferred to the skin surface through blood flow. From the skin, heat is transferred to the environment by convection, conduction, radiation, and sweat evaporation. Heat in excess of that which can be dissipated is stored in the tissue, resulting in a rise of body temperature. As mentioned below, we extend this basic Gage model to also take heat exchange between the body and the environment through respiration into account.

4 Modeling the Human Thermoregulatory System

Using the two-node modeling approach, we model the body core, the body skin, and the surroundings as *thermal entities*, and the heat flow among these entities as *thermal interactions*, as shown in Fig 3. Heat flows between the core and the skin through blood vessels, and between the body and the environment through respiration. Heat flows between the skin and the environment through convection, radiation, and evaporation. The heat production inside the body through metabolic processes and the heat production by muscles through shivering are represented as one-sided thermal interactions.

4.1 Thermal Entities of the Human Body

The change of temperature of a thermal entity with mass m and specific heat capacity c is given by $\dot{T} = \frac{\sum \dot{Q}}{m \cdot c}$, where $\sum \dot{Q}$ is the amount of heat transferred per time unit. We therefore model a thermal entity by extending the built-in class `PhysicalEntity` with the entity's heat capacity and mass:

```
class ThermalEntity | mass : Float, heatCap : Float .
subclass ThermalEntity < PhysicalEntity .
```

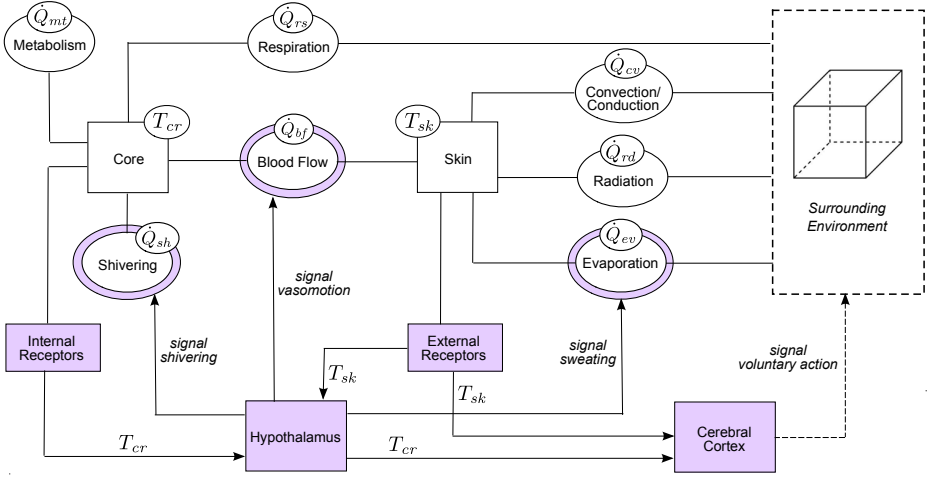


Fig. 3. Effort/flow model of the human thermoregulatory system

The continuous dynamics of the effort variable of the entity is defined as⁴

$$\text{eq effortDyn}(\langle \text{TE} : \text{ThermalEntity} \mid \text{mass} : \text{MASS}, \text{heatCap} : \text{HC} \rangle, \text{SF}) = \text{SF} / (\text{MASS} * \text{HC}) .$$

Since we want to add another continuous variable (the amount of water in a person) to the body core, we define the body core as a subclass of both `ThermalEntity` and `PhysicalEntityAC1`, where the new continuous attribute `contvar1` denotes the amount of water in the person. The body core component is defined by extending these classes with the entity's core state, body water state, the initial amount of water in the body, and some factor values for sweating, blood flow, and respiration:

```
class CoreHumanBody | coreState : CoreState, bWaterState : BWaterState,
    bWaterInit : Float, sweatRateFactor : Float,
    bloodFlowRateFactor : Float, respiRateFactor : Float .
subclass CoreHumanBody < ThermalEntity PhysicalEntityAC1 .
```

We define the temperature-related and body-water-related states of the core:

```
sort CoreState .
ops normal mildHyperthermia modHyperthermia sevHyperthermia mildHypothermia
    modHypothermia sevHypothermia death : -> CoreState [ctor] .
sort BWaterState .
ops normal modDehydration sevDehydration death : -> BWaterState [ctor] .
```

The continuous dynamics of the body water of the component is defined as

⁴ In this paper we follow the Maude convention that variables are written with (only) capital letters, and do not show the variable declarations.

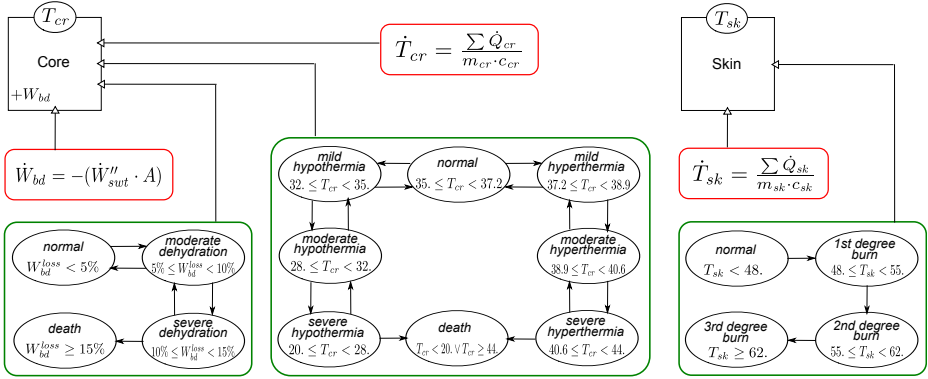


Fig. 4. The discrete and continuous dynamics of the core and the skin

```

eq contVar1Dyn(< CORE : CoreHumanBody | >,
  < BLF : BloodFlow | entity1 : CORE, entity2 : SKIN >
  < SKIN : SkinHumanBody | >
  < EVAP : Evaporation | entity1 : SKIN, sweatRate : SWTR, area : A >
  REST) = SWTR * -1.0 * A .

```

A thermal interaction between two thermal entities is a two-sided interaction, with an additional attribute that denotes the area of the flow. Similarly, the source of heat flow to a thermal entity is a one-sided interaction:

```

class ThermalInteraction | area : Float .
subclass ThermalInteraction < TwoSidedInteraction .
class ThermalFlowSource | area : Float .
subclass ThermalFlowSource < OneSidedInteraction .

```

The skin component is defined by extending the class `ThermalEntity` with an attribute for different degrees of burn injuries:

```

class SkinHumanBody | skinState : SkinState .
subclass SkinHumanBody < ThermalEntity .
sort SkinState .
ops normal firstDBurn secondDBurn thirdDBurn : -> SkinState [ctor] .

```

Changes in the body condition caused by temperature changes are represented as discrete events. For example, the core experiences *severe hyperthermia* if the core temperature exceeds 40.6°C ; this causes the sweating process to stop:

```

cr1 [modhyperthermia-to-sevhyperthermia] :
  < CORE : CoreHumanBody | effort : TEMP, coreState : modHyperthermia >
  < BLF : BloodFlow | entity1 : CORE, entity2 : SKIN >
  < SKIN : SkinHumanBody | >
  < EVAP : Evaporation | entity1 : SKIN >
  =>
  < CORE : CoreHumanBody | coreState : sevHyperthermia >   < BLF : BloodFlow | >
  < SKIN : SkinHumanBody | >   < EVAP : Evaporation | state : off >
  if TEMP > 40.6 .

```

To ensure that these rules are applied in a timely manner, we use the built-in `timeCanAdvance` function to define, for each core state, when time can advance *without* a rule having to be taken. For example:

```
eq timeCanAdvance(< CORE : CoreHumanBody | effort : TEMP, coreState : sevHyperthermia >
  = TEMP > 40.6 and TEMP <= 44.0 .
```

Changes in the volume of water in the body may cause discrete changes in the body core (see Fig. 4). For example, the body water state changes to *severe dehydration* if the body has lost more than 10% of its initial amount of water:

```
crl [moddehydration-to-sevhydration] :
  < CORE : CoreHumanBody | contVar1 : BWATERCUR, bWaterInit : BWATERINIT >
  =>
  < CORE : CoreHumanBody | bWaterState : sevDehydration >
  if bWaterLoss(BWATERCUR, BWATERINIT) >= 0.1 .
```

Change in skin temperature may change the state of the skin, e.g., to *second degree burn* if the skin temperature exceeds 55.0°C ; this causes the evaporation to stop (the skin experiences *third degree burn* if its temperature exceeds 62.0°C):

```
crl [1st-degree-burn-to-2nd-degree] :
  < SKIN : SkinHumanBody | effort : TEMP, skinState : firstDBurn >
  < SWEAT : Evaporation | entity1 : SKIN >
  =>
  < SKIN : SkinHumanBody | skinState : secondDBurn >
  < SWEAT : Evaporation | state : off > if TEMP >= 55.0 .
```

4.2 Thermal Interactions of the Human Body

Due to lack of space, we only explain the modeling of one of the thermal interactions in the system, and refer to [6] for an overview of the other interactions.

The heat flow rate between the core and the skin through the blood vessels per square meter of body area can be computed using the equation $\dot{Q}_{bf} = (K + \dot{m}_{bl} \cdot c_{bl})(T_{cr} - T_{sk})$, where K is the thermal conductivity between core and skin, \dot{m}_{bl} is the blood flow rate and c_{bl} is the specific heat capacity of blood [14]. The model of heat exchange through blood flow is shown in Fig 5. Two discrete states represent whether the blood flow is active or not, since blood stops flowing when a person is dead (since the heart cannot pump blood anymore). Increasing or decreasing blood flow through vasodilation or vasoconstriction is managed by the hypothalamus which determines the value of the blood flow rate.

The blood flow component is defined by extending the `ThermalInteraction` class with attributes for the blood flow rate, the blood thermal conductivity, and the blood heat capacity:

```
class BloodFlow | state : onOffStatusType, conduct : Float, heatCap : Float,
  bloodFlowRate : Float .
subclass BloodFlow < ThermalInteraction .
```

The continuous dynamics of the flow variable is defined in the usual way:

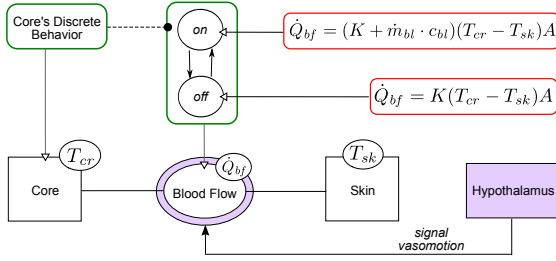


Fig. 5. The dynamics of heat exchange through the blood flow

```
eq flowDyn(< BF : BloodFlow | state : on, conduct : COND, heatCap : HC,
           bloodFlowRate : BFR, area : A >,
           TEMPCR, TEMPSK) = (COND + HC * BFR) * (TEMPCR - TEMPSK) * A .
```

```
eq flowDyn(< BF : BloodFlow | state : off, conduct : COND, area : A >, TEMPCR, TEMPSK)
= COND * (TEMPCR - TEMPSK) * A .
```

The following rule defines the behavior of the blood flow component when it receives a signal (message) from the hypothalamus containing the value of the desired blood flow rate:

```
rl [vaso] :
  signalVaso(BF, HYPOTHAL, BFR)
  < BF : BloodFlow | entity1 : CORE >
  < CORE : CoreHumanBody | coreState : CRST, bWaterState : BWST,
    bloodFlowRateFactor : BFRF >
=>
  < BF : BloodFlow | bloodFlowRate : if CRST /= dead and BWST /= death
    then BFRF * BFR else 0.0 fi >
  < CORE : CoreHumanBody | > fromActuator(HYPOTHAL, BF) .
```

4.3 The Controllers

To model the regulatory process in the human body, we have defined a control system with *controllers*, *sensors*, and *actuators*. A sensor is connected to a component in a physical system to monitor the value of some variable and to periodically send the value to one or more controllers. A controller receives information from one or more sensors, and performs the controlling actions by sending messages/signals to one or more actuators. We again refer to the longer report [6] for details about this control system infrastructure and the cortex.

The *hypothalamus* component is defined by extending the controller class with attributes for temperature set points for the core and the skin. The hypothalamus component is triggered when it receives the core and skin temperature values. If the hypothalamus is not inactive, it uses these values to compute the appropriate messages/signals to send to the sweating, shivering, and blood flow components, and to send the core temperature value to the cortex.


```

class Hypothalamus | setPointCore : Float, setPointSkin : Float .
subclass Hypothalamus < Controller .

rl [hypo-manages-involuntary-thermoreg] :
tempValCore(HYPOTHAL, RECEPTCR, TEMPCR)
tempValSkin(HYPOTHAL, RECEPTSK, TEMPSK)
< HYPOTHAL : Hypothalamus | status : waiting, setPointCore : SPCR,
    setPointSkin : SPSK >
< SWEAT : EvapSkinSweating | controller : HYPOTHAL >
< SHIVER : Shivering | controller : HYPOTHAL >
< BF : BloodFlow | controller : HYPOTHAL >
< CORTEX : Cortex | dataProvider : HYPOTHAL >
=>
< HYPOTHAL : Hypothalamus | status : running, sensors : RECEPTCR ; RECEPTSK,
    actuators : BF ; SWEAT ; SHIVER >
< SWEAT : EvapSkinSweating | > < SHIVER : Shivering | >
< BF : BloodFlow | > < CORTEX : Cortex | >
signalVaso(BF, HYPOTHAL, bloodFlowRate(TEMPCR, TEMPSK, SPCR, SPSK))
makeSignalSweating(SWEAT, HYPOTHAL, TEMPCR, TEMPSK, SPCR, SPSK)
makeSignalShivering(SHIVER, HYPOTHAL, TEMPCR, TEMPSK, SPCR, SPSK)
tempValCoreFromHypothalamus(CORTEX, HYPOTHAL, TEMPCR) .
    
```

Some of the “messages” above are functions which generate the appropriate message. For example, when the body temperature is considered too hot, a signal containing the on value and the sweat rate value is sent to the sweating component. If the body temperature is considered fine, only the off value is sent. The treatment is similar for messages to the shivering component:

```

op makeSignalSweating : Oid Oid Float Float Float Float -> Msg .
op makeSignalShivering : Oid Oid Float Float Float Float -> Msg .

msg signalVaso : Oid Oid Float -> Msg .
msgs signalSweating signalShivering : Oid Oid OnOffStatusType Float -> Msg .
msg hypothalamusInactive : Oid Oid -> Msg .

ceq makeSignalSweating(SWEAT, HYPOTHAL, TEMPCR, TEMPSK, SPCR, SPSK) =
    signalSweating(SWEAT, HYPOTHAL, if SWTR /= 0.0 then on else off fi, SWTR)
    if SWTR := sweatRate(TEMPCR, TEMPSK, SPCR, SPSK) .

ceq makeSignalShivering(SHIVER, HYPOTHAL, TEMPCR, TEMPSK, SPCR, SPSK) =
    signalShivering(SHIVER, HYPOTHAL, if SHVR /= 0.0 then on else off fi, SHVR)
    if SHVR := shiverRate(TEMPCR, TEMPSK, SPCR, SPSK) .
    
```

The function `bloodFlowRate`, governing vasodilation and vasoconstriction, is defined according to the equation for the blood flow rate for two-node models as $\dot{m}_{bl} = \frac{1}{3600} [6.3 + \frac{200 \cdot WSIG_{cr}}{1+0.5 \cdot CSIG_{sk}}]$, where $WSIG_{cr}$ and $CSIG_{sk}$ are the effector controlling signals for, respectively, vasodilation and vasoconstriction [14].

The equation for the shivering heat production rate for the two-node model is $\dot{Q}_{sh}'' = 19.4 \cdot CSIG_{sk} \cdot CSIG_{cr}$ (in W/m^2 ; we multiply by 10^{-3} since we use kiloWatts):

```

op shiverRate : Float Float Float Float -> Float .
eq shiverRate(TEMPCR, TEMPSK, SPCR, SPSK) =
    19.4 * cSigSK(TEMPSK, SPSK) * cSigCR(TEMPCR, SPCR) * 0.001 .
    
```



Fig. 6. The Sauna World Championships in Heinola, Finland

The function computing the sweat rate is based on [14] where it is defined as $\dot{m}_{sw} = 4.7 \cdot 10^{-5} \cdot WSIG_{body} \cdot \exp\left(\frac{WSIG_{sk}}{10.7}\right)$:

```
op sweatRate : Float Float Float Float -> Float .
eq sweatRate(TEMPCR, TEMPSK, SPCR, SPSK) =
  4.7 * 0.00001 * wSigBody(TEMPCR, TEMPSK, SPCR, SPSK) * exp(wSigSK(TEMPSK, SPSK)).
```

5 Extreme Exposure: The Sauna World Championships

The Sauna World Championships were an annual event held in Heinola, Finland. The winner is the contestant who can stay the longest in an oppressively hot sauna. Before the torturing game starts, the sauna is pre-heated to 110°C (warmer than the boiling point for blood). To make the conditions even worse, every thirty seconds half a liter of water is poured onto the hot sauna rocks which are the heat source of the sauna. The intense vapor from this water increases the humidity of the sauna, which makes it more difficult for the participants' sweat to evaporate. The world record of 18 minutes and 15 seconds was set in the 2008 championships. The championships in 2010 ended in a tragedy. The two last finalists collapsed with severe burn injuries after about six minutes. One of them died a day later, and the other, a five-time champion, survived after two months in coma, with serious damage to his skin, lungs and kidneys. They were both conscious but were unable to get out of the sauna on their own. The cause of this tragedy is still under investigation. The temperature in the sauna was similar to those in previous years and the times of the competitors were about the same, until this terrible final round.

This section shows how we can use HI-Maude and our model to formally analyze the ability of an unhealthy, a normal, and a trained human body to survive extreme conditions similar to those in the sauna world championships. We also

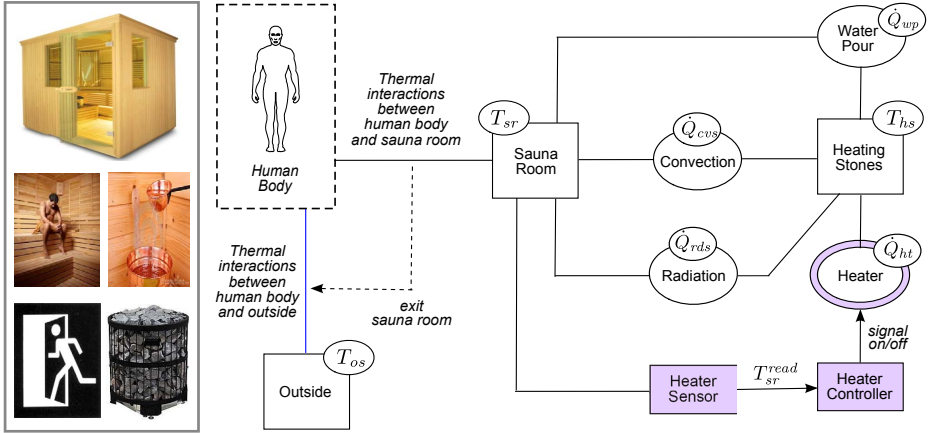


Fig. 7. The sauna room environment

try to find out what may have happened that fateful day in 2010. In particular, Section 5.1 gives an overview of our model of the thermodynamics of the sauna. We do not have any official information about the sauna environment used in the world championships, but after researching literature, product descriptions, etc., we have obtained information that we use to define the parameter values (see [6]). Section 5.2 then presents some snippets of our HI-Maude analyses.

5.1 Modeling the Sauna

The sauna is modeled as a room which uses special rocks to provide heat to the room, as shown in Fig. 7. The rocks are heated by a heater, which is connected to a control system that manages the temperature of the room. Some amount of water is periodically poured on the rocks.

Thermal Entities. The thermal entity for the sauna room is modeled in a different way than the other thermal entity components. For example, there are two attributes for the mass: for the dry air and the water vapor, since we have to model the effect of pouring water on the rocks.

```
class SaunaRoom | massDryAir : Float, spHeatAir : Float, massWaterVap : Float,
                  spHeatWater : Float, relHumid : Float, timer : Int, period : Int,
                  vol : Float, gcDryAir : Float, gcWaterVap : Float .
subclass SaunaRoom < PhysicalEntity .
```

The attributes `massDryAir` and `massWaterVap` specify the mass of dry air and of water vapor, respectively; `spHeatAir` and `spHeatWater` specify the specific heat of air and water, respectively; `relHumid` keeps the value of relative humidity of the room; `timer` is used for triggering periodical events; `vol` denotes the volume

of the sauna room; and `gcDryAir` and `gcWaterVap` are the gas constants of dry air and water vapor, respectively.

The continuous dynamics of the effort variable of the sauna is defined by

```
ceq effortDyn(< ROOM : SaunaRoom | massDryAir : MASSDA, spHeatAir : SHDA,
              massWaterVap : MASSWV, spHeatWater : SHWV >, SF)
= SF / ((MASSDA + MASSWV) * specHeatHumid(SHDA, SHWV, HUMSPEC))
if HUMSPEC := humidRatio2(MASSDA, MASSWV) .
```

This equation is the basic equation for thermal entities, taking into account that the mass of the room is a combination of the mass of the dry air and of the water vapor. The function `specHeatHumid` computes the specific heat of a mix of dry air and water vapor, and `humidRatio` computes the humidity ratio of the mix.

The *sauna rocks* component is defined as a simple thermal entity:

```
class SaunaRocks .      subclass SaunaRocks < ThermalEntity .
```

Heat transfer from the rocks occurs through convection and radiation. We use the basic forms of these heat transfers:

```
class ConvectionBasic | convectCoeff : Float .
class RadiationBasic | emissiv : Float .
subclass ConvectionBasic RadiationBasic < ThermalInteraction .

eq flowDyn(< CONV : ConvectionBasic | convectCoeff : COEFF, area : A >, TEMP1, TEMP2)
= COEFF * (TEMP1 - TEMP2) * A .

eq flowDyn(< RAD : RadiationBasic | emissiv : EMMI, area : A >, TEMP1, TEMP2)
= EMMI * stefBoltzConst * A * ((TEMP1 ^ 4.0) - (TEMP2 ^ 4.0)) .
```

Pouring Water. We make some simplifying assumptions when considering the thermal effect of pouring water on the rocks: the effect of pouring water is a heat loss for the heating rocks and a heat gain for the sauna; all the water is vaporized; and the vaporization is instantaneous. The *water pouring* component is defined as a thermal interaction between the rocks and the sauna room:

```
class WaterPouringHeatLoss | mass : Float, temp : Float, heatCap : Float,
                           heatEvap : Float, timer : Int, period : Int .
subclass WaterPouringHeatLoss < PhysicalInteraction .
```

The attribute `mass` specifies the amount of water poured; `temp` defines the temperature of the poured water; `heatCap` and `heatEvap` represent the heat capacity and the latent heat evaporation of the water, respectively; and `timer` and the `period` are used to trigger the water pouring event periodically.

The flow variable of the water pouring component represents the rate of heat flow from the rocks to the sauna room. We separate between the case when the water is poured, and when nothing happens between two water pouring events:

```
eq flowDyn(< WPHL : WaterPouringHeatLoss | mass : MASS, temp : TEMP, heatCap : HC,
          heatEvap : HEVAP, timer : TMR >,
          EFF1, EFF2)
= if TMR /= 0 then 0.0
  else MASS * HC * (100.0 - TEMP) + MASS * HEVAP * factorPourWater(EFF1, TEMP) fi .
```

When the water is poured, assuming that the water is vaporized at once, the heat transferred from the rocks to the room is the sum of the heat needed to increase the water to the boiling point and the heat needed to change the water from the liquid to vapor.

The water is poured periodically. When the associated timer expires (i.e., becomes 0), water is poured on the rocks, and the vapor content of the sauna room increases by the amount of water poured (half a litre):

```
rl [add-watervap-to-sauna] :
  < ROOM : SaunaRoom | timer : 0, massWaterVap : MASSWV, period : PER >
=>
  < ROOM : SaunaRoom | timer : PER, massWaterVap : MASSWV + 0.5 > .
```

The function `timeCanAdvance` must be used to ensure that the rule is applied at the moment when the timer expires:

```
eq timeCanAdvance(< ROOM : SaunaRoom | timer : TMR >) = TMR > 0 .
```

The other effect of pouring water on the heating rocks is an increase in the relative humidity of the sauna room. The function `computeAfterEF` is used to update the value of relative humidity, and also the timer (which should be synchronized with the timer of the pouring water component):

```
ceq computeAfterEF(< ROOM : SaunaRoom | timer : TMR, effort : TEMPAM, vol : VOL,
                  massWaterVap : MASSWV, gcWaterVap : GCWV,
                  massDryAir : MASSDA, gcDryAir : GCDA > REST)
= < ROOM : SaunaRoom | timer : TMR - 1,
    relHumid : relativeHumidity(PRESTOT, PRESSAT, HUMSPEC) >
  computeAfterEF(REST)
if TMR > 0 /\ PRESWV := gasPressure(MASSWV, GCWV, TEMPAM, VOL)
           /\ PRESDA := gasPressure(MASSDA, GCDA, TEMPAM, VOL)
           /\ PRESTOT := PRESWV + PRESDA
           /\ PRESSAT := waterVaporPressure(TEMPAM)
           /\ HUMSPEC := humidRatio(MASSDA, MASSWV) .
```

The Sauna Heating System. The heating system considered here is an automatic control system which manages to keep the temperature of the sauna room at a specified value. The room temperature read by a sensor determines the activation and deactivation of the heater. The specification is fairly standard (in the context of this paper!) and is not further explained.

5.2 HI-Maude Analysis

We are now ready to use HI-Maude to analyze how long people in different states of fitness for the event can endure in different kinds of saunas, as well as to analyze some of our own hypotheses for what may have happened at the 2010 championships. As described in [6], we model the persons according to known medical facts/assumptions and models. We use physical parameter values for the sauna heating system based on real values found in product descriptions of

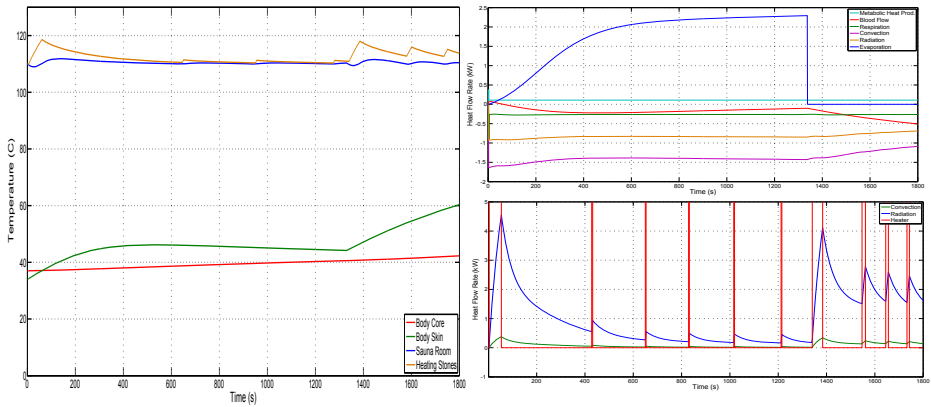


Fig. 8. Simulation results for the dry sauna: temperatures of the human body and the sauna environment thermal entities (left), and heat flow rates of the human body (top right) and of the sauna (bottom right) thermal interactions

commercial sauna heating systems (see [6]). The experiments were performed on a computer with an Intel Pentium 4 CPU 3.00 GHz and 3 GB of RAM. All analyses are carried out using Euler’s numerical method with fixed time increment size one.

We model three kinds of saunas:

- a dry sauna, where no water is poured on the heating rocks;
- a moderate wet sauna, where half a liter of water is poured every 5 minutes;
- an extreme wet sauna, where half a liter of water is poured every 30 seconds.

To analyze what happens to our virtual experimental subject after 30 minutes in the sauna, we can use the HI-Maude simulation command

```
(hrew cs1 in time <= 1800 using euler stepsize 1.0 discretewitch nonaccurate .)
```

where `cs1` is an initial state consisting of all appropriate physical entity objects and physical interaction objects with their respective initial values (see again [6]). Figure 8 shows the simulation results for the dry sauna for the average person up to 30 minutes. In the beginning the skin can handle the heat from the sauna room well, while the core temperature increases slowly. However at some point between minute 20 and 23, the skin temperature increases drastically. As we see in the graph on the right, at that point the sweating stops, possibly due to severe hyperthermia or second degree skin burn. At the same point, the heat flow from the blood vessels transfers heat from the skin to the core at an increasing rate. We next use the hybrid find earliest command to find out when a person encounters severe hyperthermia:

```
(hfind earliest cs1 =>*
  {REST:Configuration < personCore : CoreHumanBody | coreState : CRST >}
  such that (CRST == sevHyperthermia)
  using euler stepsize 1.0 discretewitch nonaccurate .)
```

The following table shows the results of the analysis command above for different sauna environments and different persons, for severe hyperthermia, severe dehydration, and second degree skin burn:

Person	Body condition	Sauna		
		dry	moderate wet	extreme wet
Normal	sev.hyperthermia	1177 s CPU: 497756ms	1104 s CPU: 472856ms	719 s CPU: 316259ms
	sev.dehydration	no result CPU: –	no result CPU: –	no result CPU: –
	skin 2nd burn	1366 CPU: 651181ms	1258 s CPU: 606713ms	770 s CPU: 394679ms
Trained	sev.hyperthermia	2436 s CPU: 1404685ms	2160 s CPU: 1221302ms	1266 s CPU: 712206ms
	sev.dehydration	no result CPU: –	no result CPU: –	no result CPU: –
	skin 2nd burn	2792 s CPU: 1753605ms	2457 s CPU: 1790829ms	1392 s CPU: 821230ms
Unhealthy	sev.hyperthermia	780 s CPU: 330182ms	728 s CPU: 316502ms	466 s CPU: 298229ms
	sev.dehydration	no result CPU: –	no result CPU: –	no result CPU: –
	skin 2nd burn	784 s CPU: 335146ms	597 s CPU: 250891ms	340 s CPU: 159359ms

Our analyses show that even the average person should endure 12 minutes in the wet sauna before the onset of major injuries (and that all persons die from hyperthermia before becoming severely dehydrated). We next propose and analyze some possible explanations for the still unsolved tragedy that could cause major injuries to a five-time world champion in around 6 minutes:

- The initial sauna room temperature is, as expected, 110°C . But the temperature of the heating rocks is 250°C . We understand from the manual of the heating system of the product used in Heinola that the temperature sensor only monitors the air temperature of sauna room and not the heating rocks.
- The temperature sensor is wrong and the temperature is higher. It turns out that even at 150°C , they should be fine for more than 10 minutes. A temperature of around 210°C is needed to explain the outcome.
- The humidity of the sauna is extremely high from the start. We start with 39 liters of water vapor instead of the expected 10 liters.

The results of our HI-Maude analyses of these hypothesis are:

Person	Body condition	Possible problems		
		Heating rocks 250°C	Room air 210°C	Very high humidity
Trained	sev.hyperthermia	349 s	328 s	785 s
	sev.dehydration	no result	no result	no result
	skin 2nd burn	414 s	393 s	311 s

Additional analyses are given in [6], including an analysis of whether a person with high tolerance level will be badly burnt or dehydrated before thinking about exiting the sauna.

6 Concluding Remarks

We have presented a detailed and realistic formal model, with all assumptions based on scientific/medical knowledge, of the human thermoregulatory system and its interactions with different kinds of environments, including oppressive saunas, and have simulated and further formally analyzed the reaction of different kinds of people to various saunas. The results of our analyses are decently close to what we know about how long both professionals and amateurs (sports writers, rock stars, etc.) can endure in the sauna.

We have also used HI-Maude to analyze some possible explanations for the still unresolved tragedy at the 2010 Sauna World Championships.

Acknowledgments. We gratefully acknowledge financial support for this work by The Research Council of Norway through the Rhytm project and by The Research Council of Norway and the German Academic Exchange Service through the DAAD ppp project HySmart.

References

1. ASHRAE Handbook: Fundamentals; SI Edition. American Society of Heating, Refrigerating and Air-Conditioning Engineers, Inc. (2005)
2. Bloomfield, L.A.: How things work: the physics of everyday life. Wiley, Hoboken (2006)
3. 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)
4. The Engineering Toolbox, <http://www.engineeringtoolbox.com/>
5. Fadlisyah, M., Ábrahám, E., Lepri, D., Ölveczky, P.C.: A rewriting-logic-based technique for modeling thermal systems. In: Proc. RTRTS 2010. Electronic Proceedings in Theoretical Computer Science, vol. 36 (2010)
6. Fadlisyah, M., Ölveczky, P.C., Ábrahám, E.: Formal modeling and analysis of extreme heat exposure to the human body in HI-Maude: A case study inspired by the tragedy at the 2010 Sauna World Championships. Tech. rep., Dept. of Informatics, Univ. of Oslo (2011), <http://heim.ifi.uio.no/mohamf/HI-Maude/techreport-casestudy.pdf>
7. Fadlisyah, M., Ölveczky, P.C., Ábrahám, E.: Formal modeling and analysis of hybrid systems in rewriting logic using higher order numerical methods and discrete-event detection. In: Proc. CSSE 2011. IEEE (2011)
8. Fadlisyah, M., Ölveczky, P.C., Ábrahám, E.: Object-Oriented Formal Modeling and Analysis of Interacting Hybrid Systems in HI-Maude. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 415–430. Springer, Heidelberg (2011)

9. Gagge, A., Stolwijk, J., Nishi, Y.: An effective temperature scale based on a simple model of human physiological regulatory response. *ASHRAE Trans.* 77(1), 247–262 (1971)
10. Hensel, H.: Thermoreception and temperature regulation. *Monogr. Physiol. Soc.* 38 (1981)
11. Hwang, C.L., Konz, S.A.: Engineering models of the human thermoregulatory system—a review. *IEEE Transactions on Biomedical Engineering BME-24*(4), 309–325 (1977)
12. McPherson, M.: *Subsurface Ventilation and Environmental Engineering*. Chapman & Hall (1993)
13. Ölveczky, P.C., Meseguer, J.: The Real-Time Maude Tool. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008*. LNCS, vol. 4963, pp. 332–336. Springer, Heidelberg (2008)
14. Parsons, K.: *Human Thermal Environments: The effects of hot, moderate, and cold environments on human health, comfort and performance*. Taylor & Francis, London (2003)
15. Plantadosi, C.: *The Biology of Human Survival: Life and Death in Extreme Environments*. Oxford University Press (2003)
16. Tipton, C.: *ACSM’s Advanced Exercise Physiology*. Lippincott Williams & Wilkins (2006)
17. Wellstead, P.E.: *Introduction to physical system modelling*. Academic Press (1979)
18. Winkeljohn, M.: Heat injuries forcing new technologies (2010), <http://sports.espn.go.com/ncaa/recruiting/football/news/story?id=4372652> (accessed February 2, 2012)

Order-Sorted Equality Enrichments Modulo Axioms^{*}

Raúl Gutiérrez, José Meseguer, and Camilo Rocha

Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract. Built-in equality and inequality predicates based on comparison of canonical forms in algebraic specifications are frequently used because they are handy and efficient. However, their use places algebraic specifications with initial algebra semantics beyond the pale of theorem proving tools based, for example, on explicit or inductionless induction techniques, and of other formal tools for checking key properties such as confluence, termination, and sufficient completeness. Such specifications would instead be amenable to formal analysis if an equationally-defined equality predicate enriching the algebraic data types were to be added to them. Furthermore, having an equationally-defined equality predicate is very useful in its own right, particularly in inductive theorem proving. Is it possible to *effectively* define a theory transformation $\mathcal{E} \mapsto \mathcal{E} \simeq$ that extends an algebraic specification \mathcal{E} to a specification $\mathcal{E} \simeq$ having an equationally-defined equality predicate? This paper answers this question in the affirmative for a broad class of order-sorted conditional specifications \mathcal{E} that are sort-decreasing, ground confluent, and operationally terminating modulo axioms B and have a subsignature of constructors. The axioms B can consist of associativity, or commutativity, or associativity-commutativity axioms, so that the constructors are *free modulo B*. We prove that the transformation $\mathcal{E} \mapsto \mathcal{E} \simeq$ preserves all the just-mentioned properties of \mathcal{E} . The transformation has been automated in Maude using reflection and is used in several Maude formal tools.

1 Introduction

It can be extremely useful, when reasoning about equational specifications with initial semantics, to have an explicit equational specification of the *equality predicate* as a binary Boolean-valued operator ‘ \simeq ’. For example, in *theorem proving* where the logic of universal quantifier-free formulas is automatically reduced to unconditional equational logic so that the formula $(u \neq v \vee w = r) \wedge q = t$ becomes equivalent to the equation $(\text{not}(u \simeq v) \text{ or } w \simeq r) \text{ and } q \simeq t = \text{true}$,

^{*} This work has been supported in part by NSF Grant CCF 09-05584, the “Programa de Apoyo a la Investigación y Desarrollo” (PAID-02-11) of the Universitat Politècnica de València, the EU (FEDER), the spanish MICINN/MINECO under Grant TIN2010-21062-C02 and by the Generalitat Valenciana, ref. PROM-ETEO/2011/052.

and in *inductionless induction* where inductive proofs are reduced to proofs by consistency because any equation not holding inductively makes *true = false*.

An equationally-defined predicate can as well be useful in the *elimination of built-in equalities and inequalities* that often are introduced in algebraic specifications through built-in operators. Such built-in equalities and inequalities are not defined logically but operationally, for both expressiveness and efficiency reasons, by comparison of canonical forms. However, their non-logical character renders any formal reasoning about specifications using them impossible. In particular, the *use of formal tools* such as those checking termination, local confluence, or sufficient completeness of an algebraic specification is impossible with built-in equalities and inequalities, but becomes possible when they are replaced by an equationally axiomatized equality predicate ‘ \simeq ’. That is, the equality between t and t' is now expressed as $t \simeq t' = \text{true}$, and their inequality as $t \simeq t' = \text{false}$. Furthermore, the equality $t \simeq t'$ will *still be correct* when t and t' are *terms with variables*, whereas a built-in equality predicate will often give a *false negative* answer for such terms, even when the equations are confluent and terminating. For example, for natural number addition ‘+’, defined by equations $x + 0 = x$ and $x + s(y) = s(x + y)$, the terms $x + y$ and $y + x$ are *already* in canonical form and a built-in equality predicate ‘ \equiv ’ will evaluate $x + y \equiv y + x$ to *false*. Instead, $x + y \simeq y + x$ will remain in canonical form with ‘ \simeq ’ and one can then inductively prove $x + y \simeq y + x = \text{true}$ using the equations defining ‘+’ and ‘ \simeq ’.

In principle, the meta-theorem of Bergstra and Tucker [2] ensures that any computable data type can be axiomatized as an initial algebra defined by a finite number of Church-Rosser and terminating equations. This also means that such a computable data type *plus* its equality predicate is also finitely axiomatizable by a finite set of Church-Rosser and terminating equations. However, the Bergstra-Tucker result is *non-constructive* in the sense that it does not give an algorithm to actually obtain the equational specification of the data type with its equality predicate. Therefore, what would be highly desirable in practice is a general *constructive theory transformation* $\mathcal{E} \mapsto \mathcal{E} \simeq$ that adds equationally-axiomatized equality predicates to an algebraic data type specification \mathcal{E} .

Such a transformation should be *as general as possible* for it to be useful in practice. For example, a transformation applicable only to “vanilla-flavored” specifications without support for types and subtypes, or that excludes conditional equations and rewriting modulo axioms would be extremely limited. The transformation should also come with *strong preservation properties*. For example, if \mathcal{E} is ground confluent, ground operationally terminating, and sufficiently complete, then $\mathcal{E} \simeq$ should also enjoy these same properties that are often essential both for executability and for a variety of formal reasoning forms.

These generality and property-preservation requirements on the transformation $\mathcal{E} \mapsto \mathcal{E} \simeq$ are a tall order. For instance, if f is a free constructor symbol, then the equations $f(x_1, \dots, x_n) \simeq f(y_1, \dots, y_n) = x_1 \simeq y_1 \text{ and } \dots \text{ and } x_n \simeq y_n$ and $f(x_1, \dots, x_n) \simeq g(y_1, \dots, y_m) = \text{false}$, for each constructor $g \neq f$ of same type, give a perfectly good and straightforward axiomatization of equality for f . But how can the equality predicate be defined when f satisfies, e.g., associativity and

commutativity axioms? Also, how should sorts and subsorts be dealt with? An even harder issue is the preservation of properties such as ground confluence, operational termination, and sufficient completeness. The difficulty is that for any given specification there are tools that can be used to prove such properties, but we need a proof that will work for *all* specifications in a very wide class. What we actually need are *metatheorems* ensuring that these properties are preserved under the transformation for *any* equational specification in the input class.

We present in this paper an effective theory transformation $\mathcal{E} \mapsto \mathcal{E} \simeq$ that satisfies the above-mentioned preservation properties. The class of equational theories \mathcal{E} accepted as inputs to the transformation is quite general. Modulo mild syntactic requirements, it consists of all order-sorted theories \mathcal{E} of the form $(\Sigma, E \uplus B)$ having a subsignature Ω of constructors and such that: (i) B is a set of associativity, or commutativity, or associativity-commutativity axioms¹; (ii) the equations E can be conditional and are sort-decreasing, ground confluent, and ground operationally terminating; and (iii) the constructors Ω are *free modulo* B , i.e., there is an isomorphism $\mathcal{T}_{\Sigma/E \uplus B}|_{\Omega} \cong \mathcal{T}_{\Omega/B}$ of initial algebras.

Outline. Preliminaries on order-sorted equational specifications are presented in Section 2. Section 3 contains the definition and fundamental properties of an equality enrichment. Sections 4 and 5 present the transformation $\mathcal{E} \mapsto \mathcal{E} \simeq$ and state its basic metatheorems. The implementation of the transformation, some of its practical consequences, and a case study are presented in Section 6. The extended version of this paper [9] contains the details and proofs of the mathematical development. The implementation of the transformation, the case study, and more examples are available at <http://maude.cs.uiuc.edu/tools/eq-enrich/>.

2 Preliminaries

We assume basic knowledge on term rewriting [1] and order-sorted algebra [7].

Order-Sorted Signatures and Terms. We assume an *order-sorted signature* $\Sigma = (S, \leq, F)$ with a finite poset of sorts (S, \leq) and a finite set of function symbols F . We also assume that the function symbols in F can be subsort overloaded and satisfy that if $f \in F_{w,s} \cap F_{w',s'}$, then $w \equiv_{\leq} w'$ implies $s \equiv_{\leq} s'$, where \equiv_{\leq} denotes the equivalence relation generated by \leq on S and $(w, s), (w', s') \in S^* \times S$. We say that $f : s_1 \cdots s_n \rightarrow s \in F$ is a *maximal typing* of f in Σ if there is no other $f : s'_1 \cdots s'_n \rightarrow s' \in F$ such that $s_i \leq s'_i$, $1 \leq i \leq n$, and $s \leq s'$. We let $X = \{X_s\}_{s \in S}$ be an S -sorted family of disjoint sets of variables with each X_s countably infinite. The set of Σ -terms of sort s is denoted by $T_{\Sigma}(X)_s$ and the set of ground terms of sort s is denoted by $T_{\Sigma,s}$, which we assume nonempty for each s . We let $\mathcal{T}_{\Sigma}(X)$ and \mathcal{T}_{Σ} denote the corresponding order-sorted term algebras. The set of variables of a term t is written $\text{Var}(t)$ and is extended to

¹ Identity axioms are excluded from our transformation. However, by using the transformation described in [5] and subsort-overloaded operators, one can often extend our transformation to specifications that also include identity axioms.

sets of terms in the natural way. A *substitution* θ is a sorted mapping from a finite subset $\text{Dom}(\theta) \subseteq X$ to $T_\Sigma(X)$ and extends homomorphically in the natural way; $\text{Ran}(\theta)$ denotes the set of variables introduced by θ . The application of a substitution θ to a term t is denoted by $t\theta$ and the composition of two substitutions θ_1 and θ_2 is denoted by $\theta_1\theta_2$. A substitution θ is called *ground* iff $\text{Ran}(\theta) = \emptyset$. We assume that all order-sorted signatures are *preregular* [7], so that each Σ -term t has a *least sort* $ls(t) \in S$ such that $t \in T_\Sigma(X)_{ls(t)}$.

Order-Sorted Equational Theories. A Σ -*equation* is an expression $t = t'$ with $t \in T_\Sigma(X)_s$, $t' \in T_\Sigma(X)_{s'}$, and $s \equiv_{\leq} s'$. A *conditional Σ -equation* is a Horn clause $t = t'$ if C with $t = t'$ a Σ -equation and $C = \bigwedge_i u_i = v_i$ a finite conjunction of Σ -equations. An *equational theory* is a tuple (Σ, E) with Σ an order-sorted signature and E a finite set of conditional Σ -equations. For φ a conditional Σ -equation, $(\Sigma, E) \vdash \varphi$ iff φ can be proved from (Σ, E) by the deduction rules in [13] iff φ is valid in all models of (Σ, E) [13]. An equational theory (Σ, E) induces the congruence relation $=_E$ on $T_\Sigma(X)$ defined for any $t, u \in T_\Sigma(X)$ by $t =_E u$ iff $(\Sigma, E) \vdash (\forall X) t = u$. We let $\mathcal{T}_{\Sigma/E}(X)$ and $\mathcal{T}_{\Sigma/E}$ denote the quotient algebras induced by $=_E$ on the algebras $\mathcal{T}_\Sigma(X)$ and \mathcal{T}_Σ , respectively. We call $\mathcal{T}_{\Sigma/E}$ the *initial algebra* of (Σ, E) and call a conditional Σ -equation φ an *inductive consequence* of (Σ, E) iff $\mathcal{T}_{\Sigma/E} \models \varphi$, i.e., iff $(\forall \theta : X \rightarrow T_\Sigma)(\Sigma, E) \vdash \varphi\theta$. A theory inclusion $(\Sigma, E) \subseteq (\Sigma', E')$, with $\Sigma \subseteq \Sigma'$ and $E \subseteq E'$, is called *protecting* iff the unique Σ -homomorphism $\mathcal{T}_{\Sigma/E} \rightarrow \mathcal{T}_{\Sigma'/E'}|_\Sigma$ to the Σ -reduct of the initial algebra $\mathcal{T}_{\Sigma'/E'}$ is a Σ -isomorphism.

Executability Conditions. We assume that the set of equations of an equational theory can be decomposed into a disjoint union $E \uplus B$, with B a collection of 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 substitutions, or failing otherwise. Furthermore, we assume that all axioms in B are *sort-preserving*, i.e., for each $u = v \in B$ and substitution θ we have $ls(u\theta) = ls(v\theta)$. The conditional equations E can be oriented into a set of (possibly conditional) (*ground*) *sort-decreasing*, (*ground*) *operationally terminating* [12], and (*ground*) *confluent conditional* rewrite rules \vec{E} modulo B . We let $\rightarrow_{E/B}$ denote the one-step rewrite relation induced by \vec{E} modulo B on $T_\Sigma(X)$, and let $\rightarrow_{E/B}^*$ denote its reflexive and transitive closure. A set of rewrite rules \vec{E} modulo B is: (i) *sort-decreasing* iff for each $t = t'$ if $C \in E$ and substitution θ we have $ls(t\theta) \geq ls(t'\theta)$ if $(\Sigma, E \uplus B) \vdash C\theta$; (ii) *operationally terminating* iff there is no infinite well-formed proof tree modulo B in \vec{E} [5]; and (iii) *confluent* if for all $t, t', t'' \in T_\Sigma(X)$, if $t \rightarrow_{E/B}^* t'$ and $t \rightarrow_{E/B}^* t''$, then there is $u \in T_\Sigma(X)$ such that $t' \rightarrow_{E/B}^* u$ and $t'' \rightarrow_{E/B}^* u$. A set of rewrite rules \vec{E} modulo B is *ground sort-decreasing*, *ground operationally terminating*, and *ground confluent* iff it is, respectively, sort-decreasing, operationally terminating, and confluent for ground terms. We let $t \downarrow_{E/B} \in T_{\Sigma,s}(X)$ denote the *E-canonical form* of t modulo B , i.e., $t \rightarrow_{E/B}^* t \downarrow_{E/B}$ and $t \downarrow_{E/B}$ cannot be further rewritten. Under the above assumptions $t \downarrow_{E/B}$ is unique up to B -equality.

Free Constructors Modulo. Given $\mathcal{E} = (\Sigma, E \uplus B)$ ground sort-decreasing, ground confluent, and ground operationally terminating modulo B , we say that $\Omega \subseteq \Sigma$ is a subsignature of *free constructors* modulo B iff Ω has the same poset of sorts as Σ and for each sort s in Σ and ground term $t \in T_{\Sigma,s}$ there is a $u \in T_{\Omega,s}$ satisfying $t =_{E \uplus B} u$ and, moreover, $v \downarrow_{E/B} =_B v$ for each $v \in T_{\Omega,s}$.

3 Equality Enrichments

An *equality enrichment* [14] of an equational theory \mathcal{E} is an equational theory $\mathcal{E} \simeq$ extending \mathcal{E} that defines equality in $\mathcal{T}_{\mathcal{E}}$ as a Boolean-valued function, as stated in Definition 1. In this section we fix an order-sorted signature $\Sigma = (S, \leq, F)$ and an order-sorted equational theory $\mathcal{E} = (\Sigma, E)$ with initial algebra $\mathcal{T}_{\mathcal{E}}$.

Definition 1 (Equality Enrichment) (generalizes [14, Definition 68]). An equational theory $\mathcal{E} \simeq = (\Sigma \simeq, E \simeq)$ is called an equality enrichment of \mathcal{E} , with $\Sigma \simeq = (S \simeq, \leq \simeq, F \simeq)$ and $\Sigma = (S, \leq, F)$, iff

- $\mathcal{E} \simeq$ is a protecting extension of \mathcal{E} ;
- the poset of sorts of $\Sigma \simeq$ extends (S, \leq) by adding a new sort *Bool* that belongs to a new connected component, with constants \top and \perp such that $\mathcal{T}_{\mathcal{E} \simeq, \text{Bool}} = \{\{\top, \perp\}\}$, with $\top \neq_{E \simeq} \perp$; and
- for each connected component in (S, \leq) there is a top sort $k \in S \simeq$ and a binary commutative operator $\simeq : k \ k \longrightarrow \text{Bool}$ in $\Sigma \simeq$, such that the following holds for any ground terms $t, u \in T_{\Sigma, k}$:

$$\mathcal{E} \vdash t = u \iff \mathcal{E} \simeq \vdash (t \simeq u) = \top, \quad (1)$$

$$\mathcal{E} \not\vdash t = u \iff \mathcal{E} \simeq \vdash (t \simeq u) = \perp. \quad (2)$$

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

The equality predicate \simeq in $\mathcal{E} \simeq$ is sound for inferring equalities and inequalities in the initial algebra $\mathcal{T}_{\mathcal{E}}$, even for terms with variables. The precise meaning of this claim is given by Proposition 1.

Proposition 1 (Equality Enrichment Properties). Let $\mathcal{E} \simeq = (\Sigma \simeq, E \simeq)$ be an equality enrichment of \mathcal{E} . Then, for any Σ -equation $t = u$ with $X = \text{Var}(t) \cup \text{Var}(u)$:

$$\mathcal{T}_{\mathcal{E}} \models (\forall X) t = u \iff \mathcal{T}_{\mathcal{E} \simeq} \models (\forall X) (t \simeq u) = \top, \quad (3)$$

$$\mathcal{T}_{\mathcal{E}} \models (\exists X) \neg(t = u) \iff \mathcal{T}_{\mathcal{E} \simeq} \models (\exists X) (t \simeq u) = \perp, \quad (4)$$

$$\mathcal{T}_{\mathcal{E}} \models (\forall X) \neg(t = u) \iff \mathcal{T}_{\mathcal{E} \simeq} \models (\forall X) (t \simeq u) = \perp. \quad (5)$$

Note that by using an equality enrichment $\mathcal{E} \simeq$ of \mathcal{E} , the problem of reasoning in $\mathcal{T}_{\mathcal{E}}$ about a universally quantified inequality $\neg(t = u)$ (abbreviated $t \neq u$) can be reduced to reasoning in $\mathcal{T}_{\mathcal{E} \simeq}$ about the universally quantified equality $(t \simeq u) = \perp$. A considerably more general reduction, not just for inequalities but for *arbitrary quantifier-free first-order formulae*, can be obtained with Boolean equality enrichments, as stated in Corollary 1.

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

$$\mathcal{T}_{\mathcal{E}} \models (\forall X)\varphi \iff \mathcal{T}_{\mathcal{E} \simeq} \models (\forall X)\widehat{\varphi}(t_1 \simeq u_1, \dots, t_n \simeq u_n) = \top, \quad (6)$$

where $\widehat{\varphi}(t_1 \simeq u_1, \dots, t_n \simeq u_n)$ is the $\Sigma \simeq$ -term of sort *Bool* obtained from φ by replacing every occurrence of the logical connectives \neg , \vee , and \wedge by, respectively, the function symbols \neg , \sqcup , and \sqcap in \mathcal{E}^{Bool} (making $\mathcal{T}_{\mathcal{E}, Bool}$ a Boolean algebra) and every occurrence of an atom $t_i = u_i$ by the *Bool* term $t_i \simeq u_i$, for $1 \leq i \leq n$.

A key property of an equality enrichment $\mathcal{E} \simeq$ of \mathcal{E} is that, if $\mathcal{E} \simeq$ is extended with any set E' of Σ -equations that are *not* satisfiable in $\mathcal{T}_{\mathcal{E}}$, then the resulting extension is inconsistent so that one can derive the *contradiction* $\top = \perp$. Conversely, if the set E' of Σ -equations extending $\mathcal{E} \simeq$ is satisfiable in $\mathcal{T}_{\mathcal{E}}$, then the resulting extension is consistent and therefore cannot yield a proof of contradiction. Statements (7) and (8) in Corollary 2 account for these two facts.

Corollary 2 (generalizes [14, Theorem 74]). *Let $\mathcal{E} \simeq = (\Sigma \simeq, E \simeq)$ be an equality enrichment of \mathcal{E} and let E' be a collection of Σ -equations. Then the following holds*

$$\mathcal{T}_{\mathcal{E}} \not\models E' \iff (\Sigma \simeq, E \simeq \cup E') \vdash \top = \perp, \quad (7)$$

$$\mathcal{T}_{\mathcal{E}} \models E' \iff (\Sigma \simeq, E \simeq \cup E') \not\vdash \top = \perp. \quad (8)$$

4 Equality Enrichments of Theories with Free Constructors Modulo

In this section we present the effective theory transformation $\mathcal{E} \mapsto \mathcal{E} \simeq$ for enriching order-sorted equational theories having free constructors modulo structural axioms with an equality predicate. We fix an order-sorted equational theory $\mathcal{E} = (\Sigma, E \uplus B)$, with $\Sigma = (S, \leq, F)$, and assume that $\Omega \subseteq \Sigma$ is a subsignature of free constructors modulo B , where B is a union of associative (A), commutative (C), and associative-commutative (AC) axioms. Furthermore, the following convention is adopted: for x a variable and s a sort, the expression x_s indicates that x has sort s , i.e., $x \in X_s$.

The theory transformation performed by $\mathcal{E} \mapsto \mathcal{E} \simeq$ consists of two main tasks or subtransformations. On input \mathcal{E} , it first extends \mathcal{E} by adding new sorts, the equational theory \mathcal{E}^{Bool} of Booleans with constructors \top and \perp (and with the other usual Boolean connectives equationally defined), some auxiliary functions, and the predicate $_ \simeq _$ for each top sort in the input theory \mathcal{E} . Then, it generates a set of equations defining $_ \simeq _$ that depend on the structural axioms of the symbols in Ω . More precisely,

Transformation 1: extends the input theory \mathcal{E} by:

1. generating a fresh top sort for each connected component in Σ that does not have it,
2. adding the theory \mathcal{E}^{Bool} with fresh new sort $Bool$,
3. adding a Boolean-valued (binary) commutative operator \simeq for the top sort of each connected component of \mathcal{E} , and
4. adding for each $f \in \Omega$ with A or AC structural axioms the Boolean-valued unary operator $root_f^k$, and if f is AC adding also the Boolean-valued binary operator in_f^k .

Transformation 2: for each $f \in \Omega$, and depending on the structural axioms of f , generates a suitable set of equations defining \simeq , $root_f^k$, and in_f^k .

Auxiliary Boolean-valued operators $root_f^k$ and in_f^k are useful for respectively checking if a term t is rooted by the constructor symbol f , or if a term t is an alien subterm of an f -rooted term t' . In this paper we use the Boolean theory \mathcal{E}^{Bool} specified in [3, Subsection 9.1]. The theory \mathcal{E}^{Bool} has free constructors modulo B^{Bool} , it is sort-decreasing, confluent, and operationally terminating modulo AC, and hence provides a Boolean decision procedure. It has signature of free constructors $\Omega^{Bool} = \{\top, \perp\}$, set of defined symbols $\Sigma^{Bool} \setminus \Omega^{Bool} = \{\neg, \sqcap, \sqcup, \oplus, \supset\}$, and satisfies $\mathcal{T}_{\mathcal{E}^{Bool}} \models \top \neq \perp$. The choice of \mathcal{E}^{Bool} is somewhat arbitrary: any equational theory implementing a Boolean decision procedure should suffice for our purpose (for instance, see [17] for other equational Boolean decision procedures).

Definition 2 spells out in detail Transformation 1 and prepares the ground for Transformation 2.

Definition 2 (Enrich). *Given \mathcal{E} , the transformation $\mathcal{E} \mapsto \mathcal{E}^\simeq$ generates the smallest equational theory $\mathcal{E}^\simeq = (\Sigma^\simeq, E^\simeq \uplus B^\simeq)$ satisfying:*

- $\mathcal{E} \cup \mathcal{E}^{Bool} \subseteq \mathcal{E}^\simeq$;
- the poset of sorts of \mathcal{E}^\simeq extends that of \mathcal{E} by adding a new connected component $\{Bool\}$, and by adding a fresh top sort to any connected component of the poset of sorts of \mathcal{E} lacking a top sort;
- for each top sort k in Σ^\simeq of a connected component of Σ , Σ^\simeq contains a commutative operator:

$$(\simeq) : k \ k \rightarrow Bool,$$

B^\simeq contains the commutative structural axiom:

$$x_k \simeq y_k = y_k \simeq x_k,$$

and E^\simeq contains the equation:

$$x_k \simeq x_k = \top;$$

- for each top sort k in Σ^\simeq of a connected component of Σ and for each function symbol $f : s \ s' \rightarrow s'' \in \Omega$, with $s \leq k$, $s' \leq k$, and $s'' \leq k$:
 - if f has axioms A or AC, then Σ^\simeq contains the symbol:

$$root_f^k : k \rightarrow Bool,$$

- if f has axioms AC , then $\Sigma \simeq$ contains the symbol:

$$\text{in}_f^k : k \ k \rightarrow \text{Bool};$$

- for each function symbol $f \in \Omega$, $E \simeq$ contains the equations $\text{enrich}_E(f)$ (see the upcoming definitions in this section).

Function enrich_E in Definition 2 formally specifies Transformation 2 and is defined by cases for each constructor symbol, depending on its structural axioms. We start with the definition of enrich_E for the case in which the constructor symbol has no structural axioms; we call such a symbol *absolutely free*.

Definition 3 (Absolutely Free Enrich). Assume $f \in \Omega$ is an absolutely free symbol. Then, for each maximal typing $f : s_1 \dots s_n \rightarrow s$ of $f \in \Omega$, $\text{enrich}_E(f)$ adds the following equations:

- for each $g : s'_1 \dots s'_m \rightarrow s' \in \Omega$ a maximal typing of g such that $s \equiv_{\leq} s'$ and $f \neq g$:

$$f(x_{s_1}^1, \dots, x_{s_n}^1) \simeq g(y_{s'_1}^1, \dots, y_{s'_m}^m) = \perp,$$

- for f itself:

$$f(x_{s_1}^1, \dots, x_{s_n}^n) \simeq f(y_{s_1}^1, \dots, y_{s_n}^n) = \prod_{1 \leq i \leq n} x_{s_i}^i \simeq y_{s_i}^i,$$

- for each $1 \leq i \leq n$ such that $s_i \equiv_{\leq} s$:

$$f(x_{s_1}^1, \dots, x_{s_n}^n) \simeq x_{s_i}^i = \perp.$$

In Definition 3, some equations use the Boolean operator \prod in $\mathcal{E}^{\text{Bool}}$ to obtain a recursive definition of $_ \simeq _$. Example 1 shows the results of applying Definition 2 and Definition 3 to a concrete specification.

Example 1. Consider the equational theory $\mathcal{E}^{\text{NATURAL}}$ in Figure 1 (left) that represents the natural numbers in Peano notation. An equality enrichment consists of $\mathcal{E}^{\text{NATURAL}}$ extended with the equational theory $\mathcal{E}^{\text{Bool}}$ and an equational definition of $_ \simeq _$. The equational theory in Figure 1 (right) is an equality enrichment of $\mathcal{E}^{\text{NATURAL}}$. The last equation is not essential, but it is useful for detecting a greater number of inequalities between terms with variables.

Definition 4 presents the definition of enrich_E for the case in which the input symbol is commutative. For the definition of enrich_E in the case of a commutative function symbol f with maximal typing of sort s' , it is assumed that its two arguments have the same sort s .

Definition 4 (C-Enrich). Assume $f \in \Omega$ is commutative and non-associative. Then, for each maximal typing $f : s \ s \rightarrow s'$ of $f \in \Omega$, $\text{enrich}_E(f)$ adds the following equations:

- for each $g : s'_1 \dots s'_m \rightarrow s'' \in \Omega$ a maximal typing of g such that $s' \equiv_{\leq} s''$ and $f \neq g$:

$$f(x_s^1, x_s^2) \simeq g(y_{s'_1}^1, \dots, y_{s'_m}^m) = \perp,$$

<pre>fmod NATURAL is sort Nat . op 0 : -> Nat [ctor] . op s : Nat -> Nat [ctor] . endfm</pre>	<pre>fmod EQ-NATURAL is protecting NATURAL . protecting BOOL . op ~_ : Nat Nat -> Bool [comm] . vars N M : Nat . eq N ~ N = true . eq 0 ~ s(N) = false . eq s(N) ~ s(M) = N ~ M . eq s(N) ~ N = false . endfm</pre>
---	--

Fig. 1. Equality Enrichment for $\mathcal{E}^{\text{NATURAL}}$

– for f itself:

$$f(x_s^1, x_s^2) \simeq f(y_s^1, y_s^2) = (x_s^1 \simeq y_s^1 \sqcap x_s^2 \simeq y_s^2) \sqcup (x_s^1 \simeq y_s^2 \sqcap x_s^2 \simeq y_s^1),$$

– if $s \equiv_{\leq} s'$ we add the equation:

$$f(x_s^1, x_s^2) \simeq x_s^1 = \perp.$$

For the definition of $\text{enrich}_{\mathcal{E}}$ in the case of an associative function symbol f with maximal typing of sort s , it is assumed that its two arguments have also sort s . Furthermore, a *top typing* for such an f is also assumed, i.e., a typing $f : s' s' \rightarrow s'$ satisfying that if $f : s s \rightarrow s$ is another typing with $s \equiv_{\leq} s'$, then $s' \geq s$ (note that a top typing of f may not belong to Ω , as in Example 2 below).

Definition 5 (A-Enrich). Assume $f \in \Omega$ is associative but not commutative. Then for each maximal typing $f : s s \rightarrow s$ of $f \in \Omega$, $\text{enrich}_{\mathcal{E}}(f)$ adds the following equations:

– for each $g : s'_1 \dots s'_m \rightarrow s'$ a maximal typing of $g \in \Omega$ such that $s \equiv_{\leq} s'$ and $f \neq g$:

$$\begin{aligned} f(x_s^1, x_s^2) \simeq g(y_{s'_1}, \dots, y_{s'_m}^m) &= \perp, \\ \text{root}_f^k(g(x_{s'_1}^1, \dots, x_{s'_m}^m)) &= \perp, \end{aligned}$$

– for f itself:

$$\begin{aligned} \text{root}_f^k(f(x_s^1, x_s^2)) &= \top, \\ f(x_s^1, x_s^2) \simeq f(x_s^1, y_s^2) &= x_s^2 \simeq y_s^2, \\ f(x_s^1, x_s^2) \simeq f(y_s^1, x_s^2) &= x_s^1 \simeq y_s^1, \\ f(x_s^1, x_s^2) \simeq f(y_s^1, y_s^2) &= \perp \quad \text{if } \neg (\text{root}_f^k(x_s^1)) \sqcap \\ &\quad \neg (\text{root}_f^k(y_s^1)) \sqcap \\ &\quad \neg (x_s^1 \simeq y_s^1) = \top, \end{aligned}$$

<pre> fmod LIST is protecting NATURAL . sorts NeNatList NatList . subsorts Nat < NeNatList < NatList . op nil : -> NatList [ctor] . op _;- : NeNatList NeNatList -> NeNatList [ctor assoc] . op _;_ : NatList NatList -> NatList [assoc] . var L : NatList . eq L ; nil = L . eq nil ; L = L . endfm fmod EQ-LIST is protecting LIST . protecting BOOL . protecting EQ-NATURAL . op ;-NeNL-root : NatList -> Bool . op _~_ : NatList NatList -> Bool [comm] . </pre>	<pre> vars P Q R S : NeNatList . var N : Nat . eq ;-NeNL-root(0) = false . eq ;-NeNL-root(s(N)) = false . eq ;-NeNL-root(nil) = false . eq ;-NeNL-root(P ; Q) = true . eq P ~ P = true . eq 0 ~ nil = false . eq s(N) ~ nil = false . eq (P ; Q) ~ 0 = false . eq (P ; Q) ~ s(N) = false . eq (P ; Q) ~ nil = false . eq (P ; Q) ~ P = false . eq (P ; Q) ~ Q = false . eq (P ; Q) ~ (P ; R) = Q ~ R . eq (P ; Q) ~ (R ; Q) = P ~ R . ceq (P ; Q) ~ (R ; S) = false if (not(; -NeNL-root(P)) and not(; -NeNL-root(R)) and not(P ~ R)) = true . endfm </pre>
--	---

Fig. 2. Equality Enrichment for $\mathcal{E}^{\text{LIST}}$

– for each $1 \leq i \leq 2$:

$$f(x_s^1, x_s^2) \simeq x_s^i = \perp.$$

Example 2. Consider the equational theory $\mathcal{E}^{\text{LIST}}$ in Figure 2 that specifies lists of natural numbers in Peano notation. Note that $_;-$ is a constructor symbol only when its arguments are non-empty lists. Therefore, the signature of free constructors modulo B of the theory $\mathcal{E}^{\text{LIST}}$ is:

$$\{\text{nil} : \rightarrow \text{NatList}, \text{;-} : \text{NeNatList NeNatList} \rightarrow \text{NeNatList}\}.$$

In order to have a recursive definition of equality for lists, $\text{enrich}_{\mathcal{E}}(f)$ uses the auxiliary function root_f^k that checks if a term of sort k is rooted by the constructor symbol f . Figure 2 presents $\mathcal{E}^{\text{EQ-LIST}}$, an equality enrichment for $\mathcal{E}^{\text{LIST}}$. We illustrate the use of $\mathcal{E}^{\text{EQ-LIST}}$ with the following two cases:

– for $((0;0);0) \simeq ((0;0);0)$, the only applicable equations are $P \simeq P = \top$, $(P;Q) \simeq (P;R) = Q;R$, and $(P;Q) \simeq (R;Q) = P;R$ by proper associative commutations, and the result is always \top independently of their application order; and

- for $((0; s(0)); 0) \simeq (s(0); 0)$ the last equation defined for $\mathcal{E}^{\text{EQ-LIST}}$ is applicable under substitution $\sigma(P) = \sigma(S) = 0$, $\sigma(R) = s(0)$, and $\sigma(Q) = s(0); 0$ by proper associative commutations since it is the only equation that satisfies $\perp \text{root}_f^{\text{NeNatList}}(0) \sqcap \perp \text{root}_f^{\text{NeNatList}}(s(0)) \sqcap \perp 0 \simeq s(0) = \top$, thus obtaining $((0; s(0)); 0) \simeq (s(0); 0) = \perp$.

In the case in which the input symbol of $\text{enrich}_{\mathcal{E}}$ with maximal typing of sort s is associative-commutative, it is assumed that its two arguments also have sort s and there is a *top typing* for f , as in the associative case.

Definition 6 (AC-Enrich). *Assume $f \in \Omega$ is associative-commutative. Then for each maximal typing $f : s s \rightarrow s$ of $f \in \Omega$, $\text{enrich}_{\mathcal{E}}(f)$ adds the following equations:*

- for each $g : s'_1 \dots s'_m \rightarrow s'$ a maximal typing of $g \in \Omega$ such that $s \equiv_{\leq} s'$ and $f \neq g$:

$$\begin{aligned} f(x_s^1, x_s^2) &\simeq g(y_{s'_1}^1, \dots, y_{s'_m}^m) = \perp, \\ \text{root}_f^k(g(x_{s'_1}^1, \dots, x_{s'_m}^m)) &= \perp, \end{aligned}$$

- for f itself:

$$\begin{aligned} \text{root}_f^k(f(x_s^1, x_s^2)) &= \top, & \text{if } \text{root}_f^k(x_s) &= \top, \\ \text{in}_f^k(x_s, y_k) &= \perp & \text{if } \perp(\text{root}_f^k(x_s)) &= \top, \\ \text{in}_f^k(x_s, f(x_s, y_s)) &= \top & \text{if } \perp(\text{root}_f^k(x_k)) &\sqcap \\ \text{in}_f^k(x_k, f(y_s^1, y_s^2)) &= (x_k \simeq y_s^1) \sqcup & \perp(\text{root}_f^k(y_s^1)) &= \top, \\ & \text{in}_f^k(x_k, y_s^2) & \text{if } \perp(\text{root}_f^k(x_k)) &\sqcap \\ & & \perp(\text{root}_f^k(y_k)) &= \top, \\ \text{in}_f^k(x_k, y_k) &= x_k \simeq y_k & \text{if } \perp(\text{root}_f^k(x_s^1)) &\sqcap \\ & & \perp(\text{in}_f^k(x_s^1, f(y_s^1, y_s^2))) &= \top, \\ f(x_s, y_s) &\simeq f(x_s, z_s) = y_s \simeq z_s, \\ f(x_s^1, x_s^2) &\simeq f(y_s^1, y_s^2) = \perp \\ f(x_s^1, x_s^2) &\simeq x_s^1 = \perp. \end{aligned}$$

Intuitively, if a term of sort k rooted by an associative-commutative symbol f is viewed as a multiset with union operator f , then function in_f^k in Definition 6 helps in identifying the cases in which an element (a term not rooted by f) belongs to the multiset.

Example 3. Consider the equational theory $\mathcal{E}^{\text{MSET}}$ in Figure 3 defining multisets of natural numbers in Peano notation. Theory $\mathcal{E}^{\text{EQ-MSET}}$ in Figure 3 is an equality enrichment for $\mathcal{E}^{\text{MSET}}$, where auxiliary functions root_f^k and in_f^k are used to give a recursive comparison of equality for constructor terms rooted by AC-symbols.

Consider the following two cases:

- for $((0 0) 1) \simeq ((0 1) 0)$, the only applicable equations are $P \simeq P = \top$ and $(P Q) \simeq (P R) = Q R$ by proper associative-commutative commutations, and the result is always \top independently of their application order; and

<pre> fmod MSET is protecting NATURAL . sorts NeNatMSet NatMSet . subsort Nat < NeNatMSet < NatMSet . op empty : -> NatMSet [ctor] . op -- : NeNatMSet NeNatMSet -> NeNatMSet [ctor assoc comm] . op _ : NatMSet NatMSet -> NatMSet [assoc comm] . var T : NatMSet . eq empty T = T . endfm fmod EQ-MSET is protecting MSET . protecting BOOL . protecting EQ-NATURAL . op -NeNMS-root : NatMSet -> Bool . op in--NeNMS : NatMSet NatMSet -> Bool . op ~_ : NatMSet NatMSet -> Bool [comm] . vars P Q R S : NeNatMSet . var N : Nat . vars T U : NatMSet . eq -NeNMS-root(0) = false . eq -NeNMS-root(s(N)) = false . </pre>	<pre> eq -NeNMS-root(empty) = false . eq -NeNMS-root(P Q) = true . ceq in--NeNMS(P,Q) = false if -NeNMS-root(P) = true . ceq in--NeNMS(P, (P Q)) = true if not(-NeNMS-root(P)) = true . ceq in--NeNMS(T, (Q R)) = (T ~ Q) or in--NeNMS(T,R) if (not(-NeNMS-root(T)) and not(-NeNMS-root(Q))) = true . ceq in--NeNMS(T,U) = T ~ U if (not(-NeNMS-root(T)) and not(-NeNMS-root(U))) = true . eq P ~ P = true . eq 0 ~ empty = false . eq s(N) ~ empty = false . eq (P Q) ~ 0 = false . eq (P Q) ~ empty = false . eq (P Q) ~ s(N) = false . eq (P Q) ~ P = false . eq (P Q) ~ (P R) = Q ~ R . ceq (P Q) ~ (R S) = false if (not(-NeNMS-root(P)) and not(in--NeNMS(P, R S))) = true . endfm </pre>
---	---

Fig. 3. Equality Enrichment for $\mathcal{E}^{\text{MSET}}$

- for $((0 \ s(0)) \ 0) \simeq (s(0) \ 0)$, we can apply the following equations:
- $(P \ Q) \simeq P = \perp$ under substitution $\sigma(P) = 0 \ s(0)$ and $\sigma(Q) = 0$ by proper associative-commutative commutations;
 - $(P \ Q) \simeq (P \ R) = Q \ R$ under substitutions:
 - * $\sigma(P) = 0, \sigma(Q) = 0 \ s(0)$, and $\sigma(R) = s(0)$, obtaining $s(0) \ 0 \simeq s(0)$, and hence the only applicable equations now are:
 - $P \ Q \simeq s(N) = \perp$ using the substitution $\sigma(P) = s(0), \sigma(Q) = 0$ (or vice versa), and $\sigma(N) = 0$
 - $P \ Q \simeq P = \perp$ under substitution $\sigma(P) = s(0)$ and $\sigma(Q) = 0$,
 - * $\sigma(P) = s(0), \sigma(Q) = 0 \ 0$, and $\sigma(R) = 0$, obtaining $0 \ 0 \simeq 0$, and hence the only applicable equations now are:
 - $P \ Q \simeq 0 = \perp$ using the substitution $\sigma(P) = 0$ and $\sigma(Q) = 0$,
 - $P \ Q \simeq P = \perp$ under substitution $\sigma(P) = 0$ and $\sigma(Q) = 0$.

5 Executability Properties of $\mathcal{E} \simeq$

It would be enormously useful, both from theoretical and practical points of view, that if a theory \mathcal{E} satisfies some executability properties, then the equality enrichment $\mathcal{E} \simeq$ of \mathcal{E} obtained from the transformation in Section 4 could inherit these properties. In particular, if the original theory \mathcal{E} is sort-decreasing (resp., ground sort-decreasing), confluent (resp., ground confluent), and operationally terminating (resp., ground operationally terminating), then $\mathcal{E} \simeq$ should be so. Also, the subsignature of free constructors of $\mathcal{E} \simeq$ must be an extension of the subsignature of free constructors of \mathcal{E} (modulo the structural axioms). In this way, full agreement between mathematical and operational semantics is preserved in the equality enrichment $\mathcal{E} \simeq$ of \mathcal{E} .

Note that the domain of the transformation $\mathcal{E} \mapsto \mathcal{E} \simeq$ includes exactly those equational theories whose structural axioms are any combination of A and/or C axioms for some of its symbols. However, if the input theory \mathcal{E} has symbols with identity axioms, one could use the results in 5 to remove them and instead add them as equations, provided that the constructors remain free after the transformation. Note that, as illustrated by the LIST and MSET examples, where identities for lists and multisets are specified as oriented equations and not as axioms, this is often possible in practice.

In what follows, $\mathcal{E} = (\Sigma, E \uplus B)$ is an order-sorted equational theory with signature of free constructors $\Omega \subseteq \Sigma$ modulo B and $\mathcal{E} \simeq = (\Sigma \simeq, E \simeq \uplus B \simeq)$ is the Boolean equality enrichment $\mathcal{E} \simeq$ obtained by the transformation $\mathcal{E} \mapsto \mathcal{E} \simeq$. Moreover, the axioms B are any combination of A and/or C axioms for some of the function symbols in Σ .

5.1 Preservation of Executability Properties and Free Constructors

Recall from Section 2 that the equational theory $\mathcal{E} = (\Sigma, E \uplus B)$ is sort-decreasing (resp., ground sort-decreasing) iff for each $t = t'$ if $C \in E$, and substitution (resp., ground substitution) θ with $(\Sigma, E \uplus B) \vdash C\theta$, we have $ls(t\theta) \geq ls(t'\theta)$. The key observation is that since $Bool$ is a fresh sort in a new connected component of $\mathcal{E} \simeq$ and all equations in $\bigcup_{f \in \Omega} \text{enrich}_{\mathcal{E}}(f)$ are of sort $Bool$, it is impossible that

the equations in \mathcal{E}^{Bool} or in $\bigcup_{f \in \Omega} \text{enrich}_{\mathcal{E}}(f)$ can be applied to terms in T_{Σ} .

Theorem 1. *If \mathcal{E} is sort-decreasing (resp., ground sort-decreasing), then $\mathcal{E} \simeq$ is sort-decreasing (resp., ground sort-decreasing).*

The notion of reductive theory is used in proving $\mathcal{E} \simeq$ operationally terminating.

Definition 7 (Reductive Theory Modulo Axioms). *Let \triangleright be the strict subterm relation on terms. An equational theory $\mathcal{E} = (\Sigma, E \uplus B)$ is reductive modulo B iff there exists a reduction ordering \succ and a symmetric, stable, and monotonic relation \sim such that:*

1. $l \notin X$ for each equation $l = r$ if $\bigwedge_{i=1..n} t_i = u_i \in E$.
2. $l \succ r$ for each equation $l = r$ if $\bigwedge_{i=1..n} t_i = u_i \in E$.
3. $l(\succ \cup \triangleright)^+ t_i$ and $l(\succ \cup \triangleright)^+ u_i$ for each equation $l = r$ if $\bigwedge_{i=1..n} t_i = u_i \in E$.
4. $u \sim v$ for each equation $u = v \in B$
5. $\sim ; \succ \subseteq \succ$.

Lemma 1. *If an equational theory $\mathcal{E} = (\Sigma, E \uplus B)$ is reductive modulo B , then it is operationally terminating modulo B .*

In general, the union of two operationally terminating theories may not be operationally terminating. Furthermore, the fact of dealing with arbitrary theories whose rules are unknown makes the task of proving operational termination of the union more involved. However, sort information can be used to obtain a proof of operational termination in the following way:

1. first, we prove that $(\Sigma \simeq, E \uplus B \simeq)$ is operationally terminating (resp., ground operationally terminating) and confluent (resp., ground confluent) [\[2\]](#),
2. then, we prove that $(\Sigma \simeq, (E \simeq \setminus E) \uplus B \simeq)$ is operationally terminating (resp., ground operationally terminating),
3. finally, we prove that the union of both theories is operationally terminating (resp., ground operationally terminating).

Theorem 2. *If \mathcal{E} is sort-decreasing (resp., ground sort-decreasing), confluent (resp., ground confluent), and operationally terminating (resp., ground operationally terminating) in a Σ -extensible way, then $\mathcal{E} \simeq$ is operationally terminating (resp., ground operationally terminating).*

Since $\mathcal{E} \simeq$ is sort-decreasing by Theorem [\[1\]](#) and operationally terminating by Theorem [\[2\]](#), the confluence of $\mathcal{E} \simeq$ follows from its local confluence. Similarly, ground local confluence follows in the ground case.

Theorem 3. *If \mathcal{E} is sort-decreasing (resp., ground sort-decreasing), operationally terminating (resp., ground operationally terminating) in a Σ -extensible way, and confluent (resp., ground confluent), then $\mathcal{E} \simeq$ is confluent (resp., ground confluent).*

The proof of Theorem [\[3\]](#) is obtained by case analysis. It considers the conditional critical pairs of E that are joinable by assumption, the critical pairs of E^{Bool} that are also joinable by the choice of \mathcal{E}^{Bool} , and the conditional critical pairs of $E \simeq \setminus (E \cup E^{Bool})$. Note that, since we may have $B \simeq$ containing associative axioms, $B \simeq$ -unification is infinitary in general. In this case, we reason inductively about the possible form of any $B \simeq$ -unifier that can involve a critical pair between two oriented equations with A symbols to conclude the local confluence of $\mathcal{E} \simeq$.

Lemma [\[2\]](#) is used for reasoning about the signature of constructors of $\mathcal{E} \simeq$.

² We assume that operational termination (resp., ground operational termination) of $(\Sigma, E \uplus B)$ is Σ -extensible, i.e., if $(\Sigma, E \uplus B)$ is operationally terminating (resp., ground operationally terminating) then $(\Sigma \cup \Delta, E \uplus B)$ is so too. This is not a strong restriction in practice, since all the actual existing tools for proving termination properties on rewriting theories generate Σ -extensible orderings.

Lemma 2. *Let \mathcal{E}^\simeq be obtained by using the transformation $\mathcal{E} \mapsto \mathcal{E}^\simeq$, where \mathcal{E} is ground sort-decreasing, ground confluent, and ground operationally terminating in a Σ -extensible way, and Ω is the signature of free constructors modulo B of \mathcal{E} . Then, if $t, t' \in T_\Omega$, then $t \simeq t' \rightarrow_{E^\simeq/B^\simeq}^+ \top$ iff $t =_B t'$.*

Intuitively, Lemma 2 states that \mathcal{E}^\simeq is a conservative extension of \mathcal{E} for ground terms in Σ and the equationally defined equality predicate in \mathcal{E}^\simeq is well-defined.

We identify $\Omega^\simeq = \Omega \uplus \{\top, \perp\} \subseteq \Sigma^\simeq$ as a signature of constructors for \mathcal{E}^\simeq and prove that the constructors in Ω^\simeq are free modulo B^\simeq .

Theorem 4. *If \mathcal{E} is ground sort-decreasing, ground confluent, and ground operationally terminating in a Σ -extensible way, and Ω is the signature of free constructors modulo B of \mathcal{E} , then \mathcal{E}^\simeq has $\Omega^\simeq = \Omega \uplus \{\top, \perp\} \subseteq \Sigma^\simeq$ as a signature of free constructors modulo B^\simeq .*

5.2 \mathcal{E}^\simeq Is an Equality Enrichment

The properties inherited from \mathcal{E} are sufficient for proving that \mathcal{E}^\simeq is indeed an equality enrichment of \mathcal{E}^\simeq .

Theorem 5. *If \mathcal{E} is ground sort-decreasing, ground operationally terminating in a Σ -extensible way, and ground confluent modulo B , then \mathcal{E}^\simeq is a Boolean equality enrichment of \mathcal{E} .*

6 Automation and Applications of $\mathcal{E} \mapsto \mathcal{E}^\simeq$

The transformation $\mathcal{E} \mapsto \mathcal{E}^\simeq$ is obviously *constructive* and has been automated in Maude using its reflective features: it takes the meta-representation of \mathcal{E} in Maude as input and constructs a meta-representation of \mathcal{E}^\simeq as output. The transformation itself has already been incorporated into Maude formal tools, including the Maude Church-Rosser and Coherence Checker [6] (CRC-ChC), and the Maude Invariant Analyzer tool [18].

6.1 A Case Study

We present a case study in which the transformation $\mathcal{E} \mapsto \mathcal{E}^\simeq$ is used in the Maude Invariant Analyzer (InvA) tool [18]. The InvA tool mechanizes an inference system for deductively proving safety properties of rewrite theories: it transforms all formal temporal reasoning about safety properties of concurrent transitions to purely equational inductive reasoning. The InvA tool provides a substantial degree of mechanization and can automatically discharge many proof obligations without user intervention. In this section, we illustrate how equality enrichments can be used to support the deductive verification task in the InvA tool for a mutual exclusion property of processes in the QLOCK protocol.

The mutual exclusion protocol QLOCK uses a global queue as follows:

- each process that participates in the protocol does the following:
 - if the process wants to use the critical resource and its name is not in the global queue, it places its name in the queue;

- if the process wants to use the critical resource and its name is in the global queue, if its name is at the top of the queue then the process gains access to the critical resource; otherwise it waits; and
 - if the process finishes the critical resource, it removes its name from the top of the global queue;
- the protocol should start from a state where the queue is empty; and
- it is assumed that each process can use the critical resource any number of times.

Consider the following equational theory $\mathcal{E}^{\text{QLOCK-STATE}}$, which represents the states of QLOCK with terms of sort *State*. It protects the equational theory $\mathcal{E}^{\text{MSET}}$ presented in Section 4. Processes and names of processes are modeled with natural numbers of sort *Nat* in Peano notation. A term $Pi \mid Pw \mid Pc \mid Q$ of sort *State* describes the state in which Pi is the collection of processes whose name is not in the global queue (or *idle* processes), Pw is the collection of processes whose names that are waiting to gain access to the critical resource (or *waiting* processes), Pc is the collection of processes that are using the critical resource (or *critical* processes), and Q is the global queue of the system. Sorts *MSet* and *Queue* are used to represent collections of processes and queues of processes' names, respectively.

```
fmod QLOCK-STATE is
  protecting MSET .
  sort Queue .
  op nil : -> Queue [ctor] .
  op @_ : Nat Queue -> Queue [ctor] .
  op _;_ : Queue Queue -> Queue .
  eq nil ; Q:Queue = Q:Queue .
  eq (N:Nat @ Q1:Queue) ; Q2:Queue = N:Nat @ (Q1:Queue ; Q2:Queue) .
  sort State .
  op |_|_|_ : MSet MSet MSet Queue -> State [ctor] .
endfm
```

The behavior of a concurrent system in rewriting logic is specified by rewrite rules that define how the individual transitions change the state of the system. The specification of all transitions of QLOCK is described by six rewrite rules in the rewrite theory $\mathcal{R}^{\text{QLOCK}}$ as follows.

```
mod QLOCK is
  protecting QLOCK-STATE .
  vars Pi Pw Pc : MSet .   var Q : Queue .   vars N N' N'' : Nat .
  rl [to-wait-1] : N | Pw | Pc | Q => empty | Pw N | Pc
                  | Q ; (N @ nil) .
  rl [to-wait-2] : N Pi | Pw | Pc | Q => Pi | Pw N | Pc
                  | Q ; (N @ nil) .
  rl [to-crit-1] : Pi | N | Pc | N @ Q => Pi | empty | Pc N | N @ Q .
  rl [to-crit-2] : Pi | Pw N | Pc | N @ Q => Pi | Pw | Pc N | N @ Q .
  rl [to-idle-1] : Pi | Pw | N | N' @ Q => Pi N | Pw | empty | Q .
  rl [to-idle-2] : Pi | Pw | Pc N | N' @ Q => Pi N | Pw | Pc | Q .
endm
```

Rewrite rules `to-idle-1` and `to-idle-2` specify the behavior of a process that finishes using the critical resource: it goes to state `idle` and the name on top of the global queue is removed. Similarly, rewrite rules `to-wait-1` and `to-wait-2`, and `to-crit-1` and `to-crit-2`, specify the behavior of a process that wants to use the critical resource and of a process that is granted access to the critical resource, respectively.

We want to verify that the `QLOCK` system satisfies the following safety properties. It is key that: (i) it satisfies the mutual exclusion property, namely, that at any point of execution there is at most one process using the critical resource. We also want to verify that: (ii) the name on top of the global queue coincides with the name of the process using the critical resource, if any. Finally, we want to verify that: (iii) the global queue only contains the names of all waiting and critical processes. State predicates *mutex*, *priority*, and *cqueue*, respectively, specify properties (i), (ii), and (iii) in the following equational theory $\mathcal{E}^{\text{QLOCK-PREDS}}$. State predicate *init* specifies the set of initial states of `QLOCK`, with auxiliary function *set?* that characterizes multisets having no repeated elements. State predicate *unique* is a strengthening of *mutex* and *priority*. Auxiliary function *to-soup* on input Q of sort *Queue* computes the multiset representation of Q .

```
fmod QLOCK-PREDS is
  protecting QLOCK-STATE . protecting EQ-MSET .
  vars N N'      : Nat .   var Q      : Queue .
  vars Pi Pw Pc  : MSet .  var NeS    : NeMSet .
  ops init mutex unique priority cqueue : State -> [Bool] .
  eq init( Pi | empty | empty | nil ) = set?(Pi) .
  eq mutex( Pi | Pw | empty | Q ) = true .
  eq mutex( Pi | Pw | N | Q ) = true .
  eq mutex( Pi | Pw | N NeS | Q ) = false .
  eq unique( Pi | Pw | empty | Q ) = set?(Pi Pw) .
  eq unique( Pi | Pw | N | N @ Q ) = set?(Pi Pw N) .
  eq unique( Pi | Pw | N NeS | Q ) = false .
  eq priority( Pi | Pw | empty | Q ) = true .
  eq priority( Pi | Pw | N | N' @ Q ) = N ~ N' .
  eq priority( Pi | Pw | N Pc | N' @ Q ) = (N ~ N') and (Pc ~ empty) .
  eq cqueue( Pi | Pw | Pc | Q ) = Pw Pc ~ to-soup(Q) .
  . . . .
endfm
```

Observe that $\mathcal{E}^{\text{QLOCK-PREDS}}$ protects the equality enrichment $\mathcal{E}^{\text{EQ-MSET}}$, in Section 4, for the connected component of sort *MSet* that defines the equality enrichment for sorts *Nat*, *MSet*, and *NeMSet*. The equality enrichments for these sorts are key in the specification of the state predicates. For instance, predicates *priority* and *cqueue* are directly defined in terms of the equality predicate for sorts *Nat* and *MSet*, and also use the Boolean connective for conjunction *and* that comes with the Boolean equality enrichment. Auxiliary function *set?* also makes use of the equality enrichment for sort *Nat*. Note that, in general, defining from scratch the equality enrichment for an AC-symbol such as the multiset union in $\mathcal{E}^{\text{MSET}}$, can be a daunting task. Instead, in $\mathcal{E}^{\text{QLOCK-PREDS}}$, the definition of the state

predicate *cqueue* was straightforward with the help of the equality enrichment for multisets of natural numbers.

By using the InvA tool we are able to automatically prove that predicates *mutex* and *priority* are invariants of $\mathcal{R}^{\text{QLock}}$ for any initial state that satisfies predicate *init*. For predicate *cqueue* some proof obligations cannot be automatically discharged. In general terms, 22 out of 26 proof obligations were automatically discharged. However, this is an encouraging result, given that the current version of the InvA tool does not yet have dedicated inference support for Boolean equality enrichments, which could further improve the degree of automation.

7 Related Work and Conclusion

In [8], Goguen generalized and simplified the technique given by Musser in [15] for proving induction hypothesis without induction (so-called *inductionless induction*) using enriched theories with equality. The notion of *s-taut* related to a sort *s* can be seen as a initial approximation of what we called in this paper an equality enrichment. The technique described in the paper is based in the result stated in Corollary 2.

In [14], the authors define the notion of *equality enrichment* (without axioms) as an explicit subrepresentation of an *equational equality presentation*. Our work extends this notion of equality enrichment with subsorts and axioms and also presents an automatic way to generate this equality enrichment modulo axioms. As the authors of [14] also remark, an equality enrichment can be used for inductionless induction theorem proving.

In [16], the authors propose an equality predicate for algebraic specifications. Unlike our work, the authors do not consider axioms and sufficient completeness in their theories, hence they have to manage terms with defined symbols. In the positive cases, their equality predicate is equivalent to ours, but in the negative cases, a *false* answer in [16] does not mean that both terms are distinct for any possible instantiation (as we state in our work), because the negative rules are based on a check of convergence between terms. The goal of this behavior is to avoid false positives instead of capturing negative cases.

In conclusion, this paper solves an important open problem: how to make the addition of equationally defined equality predicates effective and automatic for a very wide class of equational specifications with initial algebra semantics. That such a transformation should exist is suggested by the Bergstra-Tucker meta-theorem [2], but such a meta-result is not constructive and gives no insight as to how the transformation could be defined. We have shown that it can be defined for a very wide class of algebraic specifications with highly expressive features such as order-sorted types, conditional equations, and rewriting modulo commonly occurring axioms. We have also shown that all the expected good properties of the input theory \mathcal{E} are preserved by the transformation $\mathcal{E} \mapsto \mathcal{E}^\approx$.

Using reflection, the transformation has been implemented in Maude and has already been integrated into the Maude Church-Rosser and Coherence Checker [6] (CRC-ChC), and the Maude Invariant Analyzer tool [18]. In the near future it

should be added to other tools such as the Maude Termination Tool [4] (MTT) and the Maude Sufficient Completeness Checker [11] (SCC). One obvious advantage of these additions is the possibility of systematically transforming specifications making use of built-in equalities and inequalities, which cannot be handled by formal tools, into specifications where such built-in equalities and inequalities are systematically replaced by equationally-defined equalities, so that formal tools can be applied. But this is not the only possible application by any means. For example, the case study in Subsection 6.1 shows how the addition of equationally-defined equality predicates also makes the specification and verification of safety properties in the InvA tool considerably easier.

It is also clear that adding an equationally-defined equality to Maude's Inductive Theorem Prover [10] (ITP) would make this tool more effective in many ways, and would also greatly reduce the complexities of dealing with arbitrary universal formulas as goals, since all such formulas could be reduced to unconditional equality goals. It would also be very useful to explore the use of the $\mathcal{E} \mapsto \mathcal{E} \approx$ transformation in *inductionless induction* theorem proving. Yet another very useful field of application would be *early failure detection* in narrowing-based unification. The idea is that $E \uplus B$ -unification goals can be viewed as equality goals, which can be detected to have already *failed* if they can be rewritten to *false* with $E \approx$ modulo $B \approx$.

In general, the contribution presented in this work opens many useful applications to improve the state of the art in formal verification of algebraic specifications using, in particular, the Maude formal environment.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
2. Bergstra, J., Tucker, J.: Characterization of Computable Data Types by Means of a Finite Equational Specification Method. In: de Bakker, J.W., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 81, pp. 76–90. Springer, Heidelberg (1980)
3. 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)
4. Durán, F., Lucas, S., Marché, C., Meseguer, J., Urbain, X.: Proving Operational Termination of Membership Equational Programs. Higher Order Symbolic Computation 21(1-2), 59–88 (2008)
5. Durán, F., Lucas, S., Meseguer, J.: Termination Modulo Combinations of Equational Theories. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 246–262. Springer, Heidelberg (2009)
6. Durán, F., Meseguer, J.: On the Church-Rosser and Coherence Properties of Conditional Order-Sorted Rewrite Theories. Journal of Logic and Algebraic Programming (2011) (to appear)
7. Goguen, J., Meseguer, J.: Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. Theoretical Computer Science 105, 217–273 (1992)

8. Goguen, J.A.: How to Prove Algebraic Inductive Hypotheses Without Induction. In: Bibel, W., Kowalski, R. (eds.) CADE 1980. LNCS, vol. 87, pp. 356–373. Springer, Heidelberg (1980)
9. Gutiérrez, R., Meseguer, J., Rocha, C.: Order-Sorted Equality Enrichments Modulo Axioms (Extended Version). Tech. rep., University of Illinois at Urbana-Champaign (December 2011), <http://hdl.handle.net/2142/28597>
10. Hendrix, J.: Decision Procedures for Equationally Based Reasoning. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA (2008)
11. Hendrix, J., Clavel, M., Meseguer, J.: A Sufficient Completeness Reasoning Tool for Partial Specifications. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 165–174. Springer, Heidelberg (2005)
12. Lucas, S., Marché, C., Meseguer, J.: Operational Termination of Conditional Term Rewriting Systems. *Information Processing Letters* 95(4), 446–453 (2005)
13. Meseguer, J.: Membership Algebra as a Logical Framework for Equational Specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
14. Meseguer, J., Goguen, J.A.: Initially, Induction and Computability. *Algebraic Methods in Semantics* (1986)
15. Musser, D.R.: On Proving Inductive Properties of Abstract Data Types. In: Proc. of the 7th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1980, pp. 154–162. ACM Press (1980)
16. Masaki, N., Kokichi, F.: On Equality Predicates in Algebraic Specification Languages. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 381–395. Springer, Heidelberg (2007)
17. Rocha, C., Meseguer, J.: Theorem Proving Modulo Based on Boolean Equational Procedures. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS/AKA 2008. LNCS, vol. 4988, pp. 337–351. Springer, Heidelberg (2008)
18. Rocha, C., Meseguer, J.: Proving Safety Properties of Rewrite Theories. In: Corradini, A., Klin, B., Cirstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 314–328. Springer, Heidelberg (2011)

Timed CTL Model Checking in Real-Time Maude*

Daniela Lepri¹, Erika Ábrahám², and Peter Csaba Ölveczky^{1,3}

¹ University of Oslo, Norway

² RWTH Aachen University, Germany

³ University of Illinois at Urbana-Champaign, USA

Abstract. This paper presents a timed CTL model checker for Real-Time Maude and its semantic foundations. In particular, we give a timed CTL model checking procedure for that is sound and complete for closed-bound formulas under a *continuous* semantics for a fairly large class of systems. An important benefit of our model checker is that it also automatically provides a timed CTL model checker for subsets of modeling languages, like Ptolemy II and (Synchronous) AADL, which have Real-Time Maude model checking integrated into their tool environments.

1 Introduction

Real-Time Maude [30] extends Maude [14] to support the formal modeling and analysis of real-time systems in rewriting logic. Real-Time Maude is characterized by its expressiveness and generality, natural model of object-based distributed real-time systems, the possibility to define any computable data type, and a range of automated formal analysis such as simulation, reachability and temporal logic model checking. This has made it possible to successfully apply the tool to a wide range of real-time systems, including advanced state-of-the-art wireless sensor network algorithms [17,32], multicast protocols [31,21], scheduling algorithms requiring unbounded queues [26], and routing protocols [33].

Real-Time Maude's expressiveness and generality also make it a suitable semantic framework and analysis tool for modeling languages for real-time systems [24]. For example, the tool has been used to formalize (subsets of) the industrial avionics modeling standard AADL [25], a synchronous version of AADL [6], Ptolemy II discrete-event (DE) models [7], the web orchestration language Orc [2], different EMF-based timed model transformation frameworks [34,10], etc. Real-Time Maude formal analysis has been integrated into the tool environment of some of these languages, enabling a model engineering process that combines the convenience of an intuitive modeling language with formal analysis.

* This work was partially supported by the Research Council of Norway through the Rhytm project, by the DAADppp HySmart project, and by AFOSR Grant FA8750-11-2-0084.

In Real-Time Maude, the data types of the system are defined by an algebraic equational specification, and the system’s instantaneous transitions are modeled by (instantaneous) rewrite rules. Time advance is modeled explicitly by so-called *tick (rewrite) rules* of the form $\{t\} \Rightarrow \{t'\}$ in time u if $cond$, where $\{_ \}$ is an operator that encloses the entire global state, and the term u denotes the *duration* of the rewrite. Real-Time Maude is parametric in the time domain, which may be discrete or dense. For dense time (in particular), tick rules typically have the form $\{t\} \Rightarrow \{t'\}$ in time x if $x \leq d \wedge cond$, where x is a new variable not occurring in t , d , or $cond$. This form of the tick rules ensures that any moment in time (within time d) can be visited, also for a dense time domain.

Real-Time Maude extends Maude’s rewriting, search, and linear temporal logic model checking features to the timed case. For dense time, it is of course not possible to execute all possible rewrite sequences. The fairly restrictive timed automaton formalism [3] trades expressiveness for decidability of key properties for dense/continuous time, since the state space can be divided into a finite number of “clock regions” so that any two states in the same region satisfy the same properties. Such a quotient seems hard to achieve for the much more expressive real-time rewrite theories. Instead, the general approach taken in Real-Time Maude is to use *time sampling strategies* to instantiate the new variable x in the tick rules, and to analyze the resulting specification instead of the original one. One such strategy advances time by a fixed amount Δ in each application of any tick rule. The *maximal* time sampling strategy advances time as much as possible in each application of a tick rule. Although the fixed-increment strategy can cover all possible behaviors in the original system when the time domain is discrete, the maximal time sampling typically only analyzes a *subset* of all the possible behaviors. However, in [28], it is shown that for a fairly large set of real-time systems appearing in practice, the maximal time sampling strategy yields sound and complete analyses for untimed LTL properties. For example, systems where events are triggered by the arrival of messages or by the expiration of some “timer,” and where time elapse does not change the valuation of the atomic propositions (this requirement almost always holds, since time elapse typically only changes timers and clocks, whose values are rarely relevant for temporal logic properties) satisfy the requirements for maximal time sampling analyses to be sound and complete.

Until recently, Real-Time Maude could only analyze *untimed* temporal logic properties, but not quantitative properties such as “the airbag must deploy within 5 ms of a crash,” or “the ventilator machine cannot be turned off more than once every 10 minutes.” This paper presents a model checker for Real-Time Maude for the timed temporal logic TCTL [5], which is an extension of the branching time logic CTL in which the temporal operators are annotated with a time interval, so that, for example, the formula $E \varphi_1 U_{[2,4]} \varphi_2$ holds if there is a path in which φ_2 holds after some time $2 \leq r \leq 4$ and where φ_1 holds in all states until then.

Going from untimed temporal logic to a timed temporal logic presents at least two significant challenges for Real-Time Maude:

1. What is the intended semantics of a Real-Time Maude specification with the above tick rule w.r.t. timed temporal logic properties? For example, given a tick rule $\{f(y)\} \Rightarrow \{f(y + x)\}$ in time x if $x \leq 3 - y$, should the property $AF_{[1,2]} \mathbf{true}$ (in all paths, a state satisfying \mathbf{true} will be reached in some time between 1 and 2) hold for initial state $\{f(0)\}$? There are paths (e.g., jumping directly from $\{f(0)\}$ to $\{f(3)\}$) where no state is visited in the desired time interval. On the other hand, the above rule could be seen as a natural way to specify a *continuous* process from $\{f(0)\}$ to $\{f(3)\}$ in Real-Time Maude, so that the *intended* semantics should satisfy the above property. We address this problem by presenting two different semantics for the satisfaction of a TCTL formula in Real-Time Maude: the *pointwise* semantics takes all paths into account, including the one where we jump directly from time 0 to time 3, whereas the *continuous* semantics allows us to break up longer ticks in smaller steps.
2. The previous soundness and completeness results for maximal time sampling analyses no longer hold. In the example above, maximal time sampling would not satisfy the existential formula $E F_{[1,2]} \mathbf{true}$, although the original model satisfies it in both the continuous and the pointwise semantics. To achieve sound and complete time sampling analyses for the continuous semantics and dense time, we always advance time by a time value $\frac{\bar{r}}{2}$, where \bar{r} is the “greatest common divisor” (as axiomatized in Section 4) of all the (non-zero) time values appearing in the annotations in the TCTL formula, as well as all the time values of the maximal tick steps reached from the initial state. We have only implemented our model checker and proved its completeness for the continuous semantics; however, we conjecture that for the pointwise semantics, we should use this time increment, as well as any multiple of it.

This paper describes our model checker, its semantic foundations, and its implementation in Maude. Most importantly, we prove that our model checker provides sound and complete model checking under the continuous semantics for TCTL formulas where the intervals are *closed* intervals; i.e., have the forms $[r_1, r_2]$ and $[r_1, \infty)$.

An important benefit of our work is that a TCTL model checker for Real-Time Maude also gives us a TCTL model checker *for free* for Ptolemy II DE models, synchronous AADL models, and other modeling languages for which Real-Time Maude models can be generated. As shown in Section 6, our model checker has already been integrated into the Ptolemy II tool, allowing the user to model check TCTL properties of Ptolemy II models from within Ptolemy II.

Related Work. The tools Kronos [38], REDLIB [36], and TSMV [22] implement TCTL model checkers for, respectively, timed automata, linear hybrid automata, and timed Kripke structures. The tool UPPAAL [9] provides an efficient symbolic model checking procedure for timed automata for a subset of *non-nested* TCTL properties. The Roméo tool [16,11], based on a timed extension of Petri nets [123], has an integrated timed model checker for some *non-nested* TCTL modalities, with the addition of bounded response properties. These formalisms

are significantly less expressive than real-time rewrite theories [27], which makes their model checking problems decidable. The first approaches to model checking timed temporal properties for Real-Time Maude are described in [20,37] and analyze important *specific* classes of timed temporal logic formulas (time-bounded response, time-bounded safety, and minimum separation), but only for *flat* object-based specifications. Unlike in [20,37], our new model checker is not limited to specific classes of temporal logic properties, but offers the *full* TCTL. The new model checker is also not limited to flat object-oriented systems, but can analyze any (sensible) Real-Time Maude model.

Paper Structure. Section 2 introduces real-time rewrite theories and Real-Time Maude. Section 3 describes our model checker and its semantics. Section 4 presents our soundness and completeness results. We discuss our model checker implementation in Section 5, and demonstrate its applicability on a Ptolemy II DE model in Section 6. Finally, concluding remarks are given in Section 7.

2 Real-Time Rewrite Theories and Real-Time Maude

A *rewrite theory* is a tuple (Σ, E, R) , where (Σ, E) is a *membership equational logic theory* [14] that defines the state space of a system as an algebraic data type, with Σ a *signature* declaring sorts, subsorts, and function symbols, and E a set of *conditional equations* and *membership axioms*, and where R is a set of *labeled conditional rewrite rules* of the form $[l] : t \longrightarrow t' \text{ if } \textit{cond}$, where l is a label, t, t' are Σ -terms, and \textit{cond} is a conjunction of *rewrite conditions* $u \longrightarrow u'$, *equational conditions* $v = v'$, and *membership conditions* $w : s$, where u, u', v, v', w are Σ -terms and s is a sort in Σ . A rule is implicitly universally quantified by the variables appearing in t, t' and \textit{cond} , and specifies a set of local *one-step transitions* in the system. Rules are applied modulo the equations E . The set $\mathbb{T}_{\Sigma/E,s}$ of *states* of sort s is defined by the E -equivalence classes of ground terms of sort s .

Real-time rewrite theories [27] are used to specify real-time systems in rewriting logic. Rules are divided into *tick rules*, that model time elapse in a system, and *instantaneous rules*, that model instantaneous change. Formally a *real-time rewrite theory* \mathcal{R} is a tuple $(\Sigma, E, R, \phi, \tau)$ such that

- (Σ, E, R) is a rewrite theory, with a sort `System` and a sort `GlobalSystem` with no subsorts or supersorts and with only one operator $\{_ \} : \text{System} \rightarrow \text{GlobalSystem}$ which satisfies no non-trivial equations; furthermore, for any $f : s_1 \dots s_n \rightarrow s$ in Σ , the sort `GlobalSystem` does not appear in $s_1 \dots s_n$.
- $\phi : \text{TIME} \rightarrow (\Sigma, E)$ is an equational theory morphism which interprets `TIME` in \mathcal{R} ; the theory `TIME` [27] defines time abstractly as an ordered commutative monoid $(\text{Time}, 0, +, <)$. We write $0, +, \dots$ instead of $\phi(0), \phi(+), \dots$ and use `Time` for $\phi(\text{Time})$.
- τ is an assignment of a term τ_l of sort `Time` to each rewrite rule in R of the form $[l] : \{t\} \longrightarrow \{t'\} \text{ if } \textit{cond}$. Such a rule is called a *tick rule* if $\tau_l \neq 0$; in this case τ_l denotes the duration of the step. Rules that are not tick rules are called *instantaneous rules* and are assumed to take zero time. Since the

initial state has the form $\{t\}$, the form of the tick rules ensures that time advances uniformly in the whole system.

We write $t \xrightarrow{r} t'$ when t can be rewritten into t' in time r by a *one-step rewrite*, and also write $t \xrightarrow{\text{inst}} t'$ for one-step rewrites applying an *instantaneous rule*.

A tick step $t \xrightarrow{r} t'$ is *maximal* if there is no $r' > r$ with $t \xrightarrow{r'} t''$ for some t'' . A *timed path* in \mathcal{R} is an infinite sequence $\pi = t_0 \xrightarrow{r_0} t_1 \xrightarrow{r_1} t_2 \xrightarrow{r_2} \dots$ such that

- for all $i \in \mathbb{N}$, $t_i \xrightarrow{r_i} t_{i+1}$ is a one-step rewrite in \mathcal{R} ; or
- there exists a $k \in \mathbb{N}$ such that $t_i \xrightarrow{r_i} t_{i+1}$ is a one-step rewrite in \mathcal{R} for all $0 \leq i < k$, there is no one-step rewrite from t_k in \mathcal{R} , and $t_j = t_k$ and $r_j = 0$ for each $j \geq k$.

For paths π of the above form we define $d_m^\pi = \sum_{i=0}^{m-1} r_i$, $t_m^\pi = t_m$ and $r_m^\pi = r_m$.

We call the timed path $\pi = t_0 \xrightarrow{r_0} t_1 \xrightarrow{r_1} t_2 \xrightarrow{r_2} \dots$ a *timed fair path* if

- for any ground term Δ of sort **Time**, if there is a k such that for each $j > k$ there is a one-step tick rewrite $t_j \xrightarrow{r_j} t$ with $\Delta \leq d_j^\pi + r$ then there is an l with $\Delta \leq d_l^\pi$, and
- for each k , if for each $j > k$ both a maximal tick step with duration 0 and an instantaneous rule can be applied in t_j then $t_l \xrightarrow{\text{inst}} t_{l+1}$ is a one-step rewrite applying an instantaneous rule for some $l > k$.

We denote the set of all timed fair paths of \mathcal{R} starting in t_0 by $tfPaths_{\mathcal{R}}(t_0)$. A term t is *reachable* from t_0 in \mathcal{R} in time r iff there is a path $\pi \in tfPaths_{\mathcal{R}}(t_0)$ with $t_k^\pi = t$ and $d_k^\pi = r$ for some k . A path π is *time-divergent* iff for each time value $r \in \mathbf{Time}$ there is an $i \in \mathbb{N}$ such that $d_i^\pi > r$.

The Real-Time Maude tool [29] extends the Maude system [14] to support the specification, simulation, and analysis of real-time rewrite theories. Real-Time Maude is parametric in the time domain, which may be discrete or dense, and defines a supersort **TimeInf** of **Time** which adds the infinity element **INF**. To cover all time instances in a dense time domain, tick rules often have one of the forms

```

cr1 [tick] : {t} => {t'} in time x if x <= u /\ cond [nonexec] . (†),
cr1 [tick] : {t} => {t'} in time x if cond [nonexec] . (*), or
rl [tick] : {t} => {t'} in time x [nonexec] . (§).

```

where x is a new variable of sort **Time** not occurring in $\{t\}$ and *cond*. This ensures that the tick rules can advance time by *any* amount in rules of the form (*) or (§) and *any* amount less than or equal to u in rules of the form (†). Rules of these forms are called *time-nondeterministic* and are not directly executable in general, since many choices are possible for instantiating the new variable x .

In contrast to, e.g., timed automata, where the restrictions in the formalism allow the abstraction of the dense time domain by “clock regions” containing bisimilar states [3], for the more complex systems expressible in Real-Time Maude there is not such a discrete “quotient”. Instead, Real-Time Maude executes time-nondeterministic tick rules by offering a choice of different *time sampling strategies* [29], so that only some moments in the time domain are visited. For example, the *maximal* time sampling strategy advances time by the maximum possible time elapse u in rules of the form (†) (unless u equals **INF**), and

tries to advance time by a user-given time value r in tick rules having other forms. In the *default* mode each application of a time-nondeterministic tick rule will try to advance time by a given time value r .

The paper [29] explains the semantics of Real-Time Maude in more detail. In particular, given a real-time rewrite theory \mathcal{R} and a time sampling strategy σ , there is a real-time rewrite theory \mathcal{R}^σ that has been obtained from \mathcal{R} by applying a theory transformation corresponding to using the time sampling strategy σ when executing the tick rules. In particular, the real-time rewrite theory $\mathcal{R}^{maxDef(r)}$ denotes the real-time rewrite theory \mathcal{R} where the tick rules are applied according to the maximal time sampling strategy, while $\mathcal{R}^{def(r)}$ denotes \mathcal{R} where the tick rules are applied according to the default time sampling strategy (tick steps which advance time by 0 are not applied).

A real-time rewrite theory \mathcal{R} is *time-robust* if the following hold for all ground terms t, t', t'' of sort `GlobalSystem` and all ground terms r, r' , of sort `Time`:

- $t = t'$ holds in the underlying equational theory for any 0-time tick step $t \xrightarrow{0} t'$.
- $t \xrightarrow{r+r'} t''$ if and only if there is a t' of sort `Time` such that $t \xrightarrow{r} t'$ and $t' \xrightarrow{r'} t''$.
- If $t \xrightarrow{r} t'$ is a tick step with $r > 0$, and $t' \xrightarrow{inst} t''$ is an instantaneous one-step rewrite, then $t \xrightarrow{r} t''$ is a *maximal* tick step.
- for $M = \{r \mid \exists t'. t \xrightarrow{r} t'\}$ we have that either there is a maximal element in M or M is the whole domain of `Time`.

Real-Time Maude extends Maude's *linear temporal logic model checker* to check whether each behavior, possibly up to a certain time bound, satisfies an (untimed) LTL formula. *State propositions* are terms of sort `Prop`. The labeling of states with propositions can be specified by (possibly conditional) equations of the form

$$\{statePattern\} \mid= prop = b$$

for b a term of sort `Bool`, which defines the state proposition *prop* to evaluate to b in all states matching the given pattern. We say that a set of atomic propositions is *tick-invariant* in \mathcal{R} if tick rules do not change their values.

Since the model checking commands execute time-nondeterministic tick rules according to a time sampling strategy, only a subset of all possible behaviors is analyzed. Therefore, Real-Time Maude analysis is in general *not sound and complete*. However, the reference [28] gives easily checkable sufficient conditions for soundness and completeness, which are satisfied by many large Real-Time Maude applications.

3 Timed CTL Model Checking for Real-Time Maude

In untimed temporal logics it is not possible to reason about the *duration* of/between events. There are many *timed* extensions of temporal logics [4,35,12]. In this paper we consider TCTL [5] with interval time constraints on temporal operators.

3.1 Timed CTL

In *computation tree logic (CTL)* [5], a *state formula* specifies a property over the computation tree corresponding to the system behavior rooted in a given state. State formulae are constructed by adding universal (A) and existential (E) path quantifiers in front of *path formulae* to specify whether the path formula must hold, respectively, on *each* path starting in the given state, or just on *some* path. Path formulae are built from state formulae using the temporal operators X (“next”) and U (“until”), from which F (“finally”) and G (“globally”) can be derived.

Timed CTL (TCTL) is a quantitative extension of CTL [5], where the scope of the temporal operators can be limited in time by subscripting them with time constraints. In this paper we consider an interval-bound version of TCTL where the temporal operators are subscripted with a time interval. A *time interval* I is an interval of the form $[a, b]$, $(a, b]$, $[a, b_\infty)$ or (a, b_∞) , where a and b are values of sort Time and b_∞ is a value of sort TimeInf .

Definition 1. *Given a set Π of atomic propositions, TCTL formulae are built using the following abstract syntax:*

$$\varphi ::= \text{true} \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid E\varphi U_I \varphi \mid A\varphi U_I \varphi$$

where $p \in \Pi$ and I is a time interval.

We omit the bound $[0, \infty)$ as subscript and we write $\leq b$, $< b$, $\geq a$ and $> a$ for $[0, b]$, $[0, b)$, $[a, \infty)$ and (a, ∞) , respectively. We denote by TCTL_{cb} the fragment of TCTL where all time bounds are of the form $[a, b]$ with $a < b$, or $[a, \infty)$.

3.2 Timed Kripke Structures and TCTL Semantics

The semantics of TCTL formulae is defined on Kripke structures. A *Kripke structure* is a transition system with an associated labeling function, which maps each state in the transition system to the set of atomic propositions that hold in that state.

A *timed Kripke structure* is a Kripke structure where each transition has the form $s \xrightarrow{r} s'$, where r denotes the duration of the transition step.

Definition 2. *Given a set of atomic propositions Π and a time domain \mathcal{T} , a timed Kripke structure is a triple $TK = (S, \xrightarrow{\mathcal{T}}, L)$ where S is a set of states, $\xrightarrow{\mathcal{T}} \subseteq S \times \mathcal{T} \times S$ is a transition relation with duration, and L is a labeling function $L : S \rightarrow \mathcal{P}(\Pi)$. The transition relation $\xrightarrow{\mathcal{T}}$ is **total**¹ i.e., for each $s \in S$ there exist $r \in \mathcal{T}$, $s' \in S$ such that $(s, r, s') \in \xrightarrow{\mathcal{T}}$. We write $s \xrightarrow{r} s'$ if $(s, r, s') \in \xrightarrow{\mathcal{T}}$.*

¹ A transition relation $\xrightarrow{\mathcal{T}}$ can be made *total* by defining $(\xrightarrow{\mathcal{T}})^\bullet = \xrightarrow{\mathcal{T}} \cup \{(s, 0, s) \in S \times \mathcal{T} \times S \mid \neg \exists s' \in S, r \in \mathcal{T} \text{ s.t. } (s, r, s') \in \xrightarrow{\mathcal{T}}\}$.

We use a similar notation for timed paths in a timed Kripke structure \mathcal{TK} as for real-time rewrite theories. Thus, a *timed path* is written $\pi = t_0 \xrightarrow{r_0} t_1 \xrightarrow{r_1} \dots$, we define $d_m^\pi = \sum_{i=0}^{m-1} r_i$, $t_m^\pi = t_m$ and $r_m^\pi = r_m$, and the set of all timed fair paths originating in state t is denoted by $tfPaths_{\mathcal{TK}}(t)$.

The semantics of TCTL formulae is defined as follows:

Definition 3. For timed Kripke structures $\mathcal{TK} = (S, \xrightarrow{\tau}, L)$, states $t \in S$, and TCTL formulae φ , the pointwise satisfaction relation $\mathcal{TK}, t \models_p \varphi$ is defined inductively as follows:

$$\begin{aligned}
 \mathcal{TK}, t \models_p \text{true} & \quad \text{always.} \\
 \mathcal{TK}, t \models_p p & \quad \text{iff } p \in L(t). \\
 \mathcal{TK}, t \models_p \neg\varphi_1 & \quad \text{iff } \mathcal{TK}, t \not\models_p \varphi_1. \\
 \mathcal{TK}, t \models_p \varphi_1 \wedge \varphi_2 & \quad \text{iff } \mathcal{TK}, t \models_p \varphi_1 \text{ and } \mathcal{TK}, t \models_p \varphi_2. \\
 \mathcal{TK}, t \models_p E \varphi_1 U_I \varphi_2 & \quad \text{iff there exists } \pi \in tfPaths_{\mathcal{TK}}(t) \text{ and an index } k \text{ s.t.} \\
 & \quad d_k^\pi \in I, \mathcal{TK}, t_k^\pi \models_p \varphi_2, \text{ and} \\
 & \quad \mathcal{TK}, t_l^\pi \models_p \varphi_1 \text{ for all } 0 \leq l < k. \\
 \mathcal{TK}, t \models_p A \varphi_1 U_I \varphi_2 & \quad \text{iff for each } \pi \in tfPaths_{\mathcal{TK}}(t) \text{ there is an index } k \text{ s.t.} \\
 & \quad d_k^\pi \in I, \mathcal{TK}, t_k^\pi \models_p \varphi_2, \text{ and} \\
 & \quad \mathcal{TK}, t_l^\pi \models_p \varphi_1 \text{ for all } 0 \leq l < k.
 \end{aligned}$$

For a timed Kripke structure $\mathcal{TK} = (S, \xrightarrow{\tau}, L)$, a state $t \in S$ and paths $\pi, \pi' \in tfPaths_{\mathcal{TK}}(t)$ we say that π' is a *simple time refinement* of π if either $\pi = \pi'$ or π' can be obtained from π by replacing a transition $t_k \xrightarrow{r_k} t_{k+1}$, $r_k > 0$, by a sequence $t_k \xrightarrow{r'_k} t \xrightarrow{r''_k} t_{k+1}$ of transitions for some $t \in S$ and time values $r'_k, r''_k > 0$ with $r'_k + r''_k = r_k$. A path π' is a *time refinement* of another path π if π' can be obtained from π by applying a (possibly infinite) number of time refinements. We also say that π is a *time abstraction* of π' .

Definition 4. The continuous-time satisfaction relation $\mathcal{TK}, t \models_c \varphi$ is defined as the pointwise one for the first four cases; for the last two cases we have:

$$\begin{aligned}
 \mathcal{TK}, t \models_c E \varphi_1 U_I \varphi_2 & \quad \text{iff there is a path } \pi \in tfPaths_{\mathcal{TK}}(t) \text{ such that for each} \\
 & \quad \text{time refinement } \pi' \in tfPaths_{\mathcal{TK}}(t) \text{ of } \pi \text{ there is an} \\
 & \quad \text{index } k \text{ s.t. } d_k^{\pi'} \in I, \mathcal{TK}, t_k^{\pi'} \models_c \varphi_2, \text{ and} \\
 & \quad \mathcal{TK}, t_l^{\pi'} \models_c \varphi_1 \text{ for all } 0 \leq l < k. \\
 \mathcal{TK}, t \models_c A \varphi_1 U_I \varphi_2 & \quad \text{iff for each path } \pi \in tfPaths_{\mathcal{TK}}(t) \text{ there is a time} \\
 & \quad \text{refinement } \pi' \in tfPaths_{\mathcal{TK}}(t) \text{ of } \pi \text{ and an index } k \\
 & \quad \text{s.t. } d_k^{\pi'} \in I, \mathcal{TK}, t_k^{\pi'} \models_c \varphi_2, \text{ and} \\
 & \quad \mathcal{TK}, t_l^{\pi'} \models_c \varphi_1 \text{ for all } 0 \leq l < k.
 \end{aligned}$$

3.3 Associating Timed Kripke Structures to Real-Time Rewrite Theories

To each real-time rewrite theory we associate a timed Kripke structure as follows:

Definition 5. Given a real-time rewrite theory $\mathcal{R} = (\Sigma, E, R, \phi, \tau)$, a set of atomic propositions Π and a protecting extension $(\Sigma \cup \Pi, E \cup D) \supseteq (\Sigma, E)$, we define the associated timed Kripke structure

$$\mathcal{TK}(\mathcal{R})_{\Pi} = (\mathbb{T}_{\Sigma/E, \text{GlobalSystem}}, (\xrightarrow{\mathcal{T}}_{\mathcal{R}})^{\bullet}, L_{\Pi}),$$

where $(\xrightarrow{\mathcal{T}}_{\mathcal{R}})^{\bullet} \subseteq \mathbb{T}_{\Sigma/E, \text{GlobalSystem}} \times \mathbb{T}_{\Sigma/E, \phi(\text{Time})} \times \mathbb{T}_{\Sigma/E, \text{GlobalSystem}}$ contains all transitions of the kind $t \xrightarrow{r} t'$ which are also one-step rewrites in \mathcal{R} and all transitions of the kind $t \xrightarrow{0} t$ for all those states t that cannot be further rewritten in \mathcal{R} , and for $L_{\Pi} : \mathbb{T}_{\Sigma/E, \text{GlobalSystem}} \rightarrow \mathcal{P}(\Pi)$ we have that $p \in L_{\Pi}(t)$ if and only if $E \cup D \vdash (t \models p) = \text{true}$.

We use this transformation to define $\mathcal{R}, L_{\Pi}, t_0 \models_c \varphi$ as $\mathcal{TK}(\mathcal{R})_{\Pi}, t_0 \models_c \varphi$, and similarly for the pointwise semantics. The model checking problems $\mathcal{TK}(\mathcal{R})_{\Pi}, t_0 \models_p \varphi$ and $\mathcal{TK}(\mathcal{R})_{\Pi}, t_0 \models_c \varphi$ are decidable if

- the equational specification in \mathcal{R} is *Church-Rosser* and *terminating*,
- the set of states reachable from t_0 in the rewrite theory \mathcal{R} is *finite*, and
- given a pair of reachable states t and t' , the number of one-step rewrites of the kind $t \xrightarrow{r} t'$ in \mathcal{R} is *finite*.

As mentioned above, real-time rewrite theories generally contain a time-nondeterministic tick rule, but since Real-Time Maude executes such theories by applying a *time sampling strategy* σ , our model checker does not analyze \mathcal{R} but the executable theory \mathcal{R}^{σ} in which the time sampling strategy transformation has been applied. Thus, we associate a timed Kripke structure not to \mathcal{R} , but to \mathcal{R}^{σ} , and hence the third requirement is satisfied by all but the most esoteric cases; indeed, the tick rules in all Real-Time Maude applications we have seen are deterministic, in the sense that there is at most one one-step tick rewrite $t \xrightarrow{r} t'$ from any state, when the time sampling strategy is taken into account.

We denote by $\mathcal{TK}(\mathcal{R}, t_0)_{\Pi}$ the timed Kripke structure associated to \mathcal{R} which is restricted to states reachable from t_0 , and for states t reachable from t_0 we write $\mathcal{R}, L_{\Pi}, t \models \varphi$ for $\mathcal{TK}(\mathcal{R}, t_0)_{\Pi}, t \models \varphi$.

4 Sound and Complete TCTL Model Checking for Real-Time Maude

As mentioned above, for dense time domains, Real-Time Maude only analyzes those behaviors obtained by applying the tick rules according to a selected time sampling strategy. The paper [28] specifies some conditions on a real-time rewrite theory \mathcal{R} and on the atomic propositions that ensure that model checking $\mathcal{R}^{\text{maxDef}(r)}$, i.e., using the maximal time sampling strategy, is a sound and complete model checking procedure to check whether all behaviors in the original model \mathcal{R} satisfy an *untimed* LTL formula without the next operator.

For example, if no application of a tick rule changes the valuation of the atomic propositions in a formula and instantaneous rewrite rules can only be applied after *maximal* tick steps or after applying an instantaneous rule, then model checking $\mathcal{R}^{\text{maxDef}(r)}$ gives a sound and complete model checking procedure for

\mathcal{R} .² This result yields a feasible sound and complete model checking procedure for many useful (dense-time) systems, that include many systems that cannot be modeled as, e.g., timed automata.

As explained in the introduction, this completeness result does not carry over to *timed* temporal logic properties. In the following we focus on dense time, since we can achieve sound and complete model checking for discrete time by exploring all possible tick steps in the pointwise semantics, and by advancing time by the smallest possible non-zero duration in the continuous semantics. Furthermore, as already mentioned, in this paper we restrict our treatment to TCTL_{cb} formulas under the continuous semantics.³

Our goal is therefore to find a discrete abstraction of a real-time rewrite theory \mathcal{R} , so that model checking the abstraction (under the pointwise semantics) is equivalent to model checking \mathcal{R} under the continuous semantics. One part of our solution is to make sure that time progress “stops” at any time point when a time bound in the formula could be reached. This can be achieved if we split any tick step by an amount that divides all possible maximal tick durations and all possible finite non-zero time bounds in the formula. Let \bar{r} be the greatest common divisor of the durations of all maximal tick steps in $\mathcal{R}^{maxDef(r)}$ reachable from the initial state and each finite non-zero time bound in the formula; then “stopping” at each interesting time point should be achieved if we divide each maximal tick step into smaller steps of duration \bar{r} .

However, the following example shows that it is not sufficient to always advance time by this greatest common divisor \bar{r} to obtain a sound and complete abstraction under the continuous semantics. Consider a (dense-time) theory \mathcal{R} that has only one behavior in terms of maximal tick steps, which we show here in terms of validity of the atomic proposition p in the corresponding states:

$$\pi = \neg p \xrightarrow{1} \neg p \xrightarrow{inst} p \xrightarrow{inst} \neg p \xrightarrow{1} \dots (\neg p \text{ forever})$$

That is, a p -state is reachable in exactly time 1, and ticks do not change the valuations of the atomic propositions. In this model all maximal tick steps have duration 1. Let’s consider the formula $\varphi = E \varphi_1 U_{[1,1]} true$, where φ_1 is the formula $E F_{[1,1]} p$. The formula φ says that φ_1 must hold all the way until we reach time 1. The greatest common divisor of all maximal time increments and all time values in φ is still 1, so the “greatest common divisor” abstraction is equivalent to $\mathcal{R}^{maxDef(r)}$. In particular, this abstraction (i.e., the above behavior) satisfies φ w.r.t. the initial state $\pi(0)$. However, $\mathcal{R}, L_{\{p\}}, \pi(0) \models_c \varphi$ does *not* hold, since φ does not hold in the timed refinement (where the first tick has been split into two smaller ones)

$$\pi' = \neg p \xrightarrow{1/2} \neg p \xrightarrow{1/2} \neg p \xrightarrow{inst} p \xrightarrow{inst} \neg p \xrightarrow{1} \dots (\neg p \text{ forever})$$

because φ_1 does not hold in the second state in the refinement.

² The requirements in [28] are weaker than described here.

³ We are currently working on releasing the restriction to closed bounds. However, our proof for the completeness result cannot be directly extended to TCTL formulas with open bounds.

Our approach is therefore to capture all these “intermediate” states by further splitting the “gcd” tick steps into two smaller tick steps. In essence, we advance time not by \bar{r} , but by “half” the gcd \bar{r} in each tick step.

To formalize this notion, let us first consider the time domain. Real-time rewrite theories are parametric in their time domain; the time domain must only satisfy some abstract properties given in some *functional theory* defined in [27] that defines the time domain abstractly as a commutative monoid $(0, \leq, Time)$ with some additional operators. The following theory states that there exist functions `gcd` and `half` on the non-zero time values with the expected properties.

```
fth GCD-TIME-DOMAIN is including LTIME-INF .
  sort NzTime .   subsort NzTime < Time .   cmb T:Time : NzTime if T:Time /= 0 .
  op gcd : NzTime NzTime -> NzTime [assoc comm] .
  op _divides_ : NzTime NzTime -> Bool .
  op half : NzTime -> NzTime .
  vars T1 T2 T3 : NzTime . vars T T' : Time .
  eq T1 divides T1 = true .
  ceq T1 divides T2 = false if T2 < T1 .
  eq T1 divides (T1 + T2) = T1 divides T2 .
  eq gcd(T1, T2) divides T1 = true .
  ceq gcd(T1, T2) >= T3 if T3 divides T1 /\ T3 divides T2 .
  eq half(NZT) + half(NZT) = NZT .
endfth
```

In the following we assume that all considered time domains satisfy the theory `GCD-TIME-DOMAIN`, and write *gcd* and *half* for the interpretation of `gcd` and `half`, respectively.

The real-time rewrite theory $\mathcal{R}^{gcd(t_0, r, \varphi)}$ is obtained from the tick-robust real-time rewrite theory \mathcal{R} , a state t_0 in \mathcal{R} , and a TCTL formula φ , by advancing time by “half” the greatest common divisor of all the following values:

- all tick step durations appearing in paths from $tfPaths_{\mathcal{R}^{maxDef(r)}}(t_0)$ and
- all finite non-zero lower and upper bounds of all temporal operators in φ .

Definition 6. For a real-time rewrite theory \mathcal{R} whose time domain satisfies the theory `GCD-TIME-DOMAIN`, a non-zero time value r , a TCTL formula φ and a state t_0 of \mathcal{R} we define

$$T_1(\mathcal{R}, t_0, r) = \{r' \in \text{NzTime} \mid \exists \pi \in tfPaths_{\mathcal{R}^{maxDef(r)}}(t_0). \exists i \geq 0. r' = r_i^\pi\}$$

$$T_2(\varphi) = \{r \in \text{NzTime} \mid \text{there exists a subformula } E \varphi_1 U_I \varphi_2 \text{ or } A \varphi_1 U_I \varphi_2 \text{ of } \varphi \text{ with } r \text{ a non-zero finite lower or upper bound in } I\}$$

$$GCD(\mathcal{R}, r, \varphi, t_0) = gcd(T_1(\mathcal{R}, t_0, r) \cup T_2(\varphi)).$$

If $T_1(\mathcal{R}, t_0, r)$ and $T_2(\varphi)$ are finite then the *GCD* value is well-defined and we can define the real-time rewrite theory $\mathcal{R}^{gcd(t_0, r, \varphi)}$ as follows:

Definition 7. Given a real-time rewrite theory \mathcal{R} whose time domain satisfies the theory `GCD-TIME-DOMAIN`, a non-zero time value r , a TCTL formula φ , a

state t_0 of \mathcal{R} , and assume that $\bar{r} = GCD(\mathcal{R}, t_0, r, \varphi)$ is a defined non-zero time value. Then $\mathcal{R}^{gcd(t_0, r, \varphi)}$ is defined as \mathcal{R} but where each tick rule of the forms (\dagger) , $(*)$, and (\S) is replaced by the respective tick rule:

```

cr1 [tick] : {t} => {t'} in time x if x := half( $\bar{r}$ )  $\wedge$  cond [nonexec] .
cr1 [tick] : {t} => {t'} in time x if x := half( $\bar{r}$ )  $\wedge$  cond [nonexec] .
cr1 [tick] : {t} => {t'} in time x if x := half( $\bar{r}$ ) [nonexec] .
    
```

The following lemma states that the evaluation of the formula φ and its subformulas does not change inside tick steps of $\mathcal{R}^{gcd(t_0, r, \varphi)}$.

Lemma 1. *Assume a time-robust real-time rewrite theory \mathcal{R} whose time domain satisfies the theory GCD-TIME-DOMAIN. Let Π be a set of tick-invariant atomic propositions, and assume a protecting extension of \mathcal{R} defining the atomic propositions in Π and inducing a labeling function L_Π . Let t_0 be a state of \mathcal{R} , r a non-zero time value of sort **Time**, φ a TCTL_{cb} formula over Π , and assume that $\bar{r} = GCD(\mathcal{R}, t_0, r, \varphi)$ is a defined non-zero time value.*

Then for each subformula φ' of φ , each time-divergent path $\pi \in \text{tfPaths}_{\mathcal{R}}(t_0)$ and for all tick step sequences $t_i^\pi \xrightarrow{r_i^\pi} \dots \xrightarrow{r_{j-1}^\pi} t_j^\pi$ in π satisfying $n \cdot \bar{r} < d_i^\pi < d_j^\pi < (n+1) \cdot \bar{r}$ for some n we have that

$$\mathcal{R}, L_\Pi, t_i^\pi \models_c \varphi' \quad \text{iff} \quad \mathcal{R}, L_\Pi, t_j^\pi \models_c \varphi' .$$

Proof. The proof by induction on the structure of φ' can be found in our technical report [19].

Based on the above lemma we gain our completeness result:

Theorem 1. *Let \mathcal{R} , L_Π , t_0 , r , φ and \bar{r} be as in Lemma 1. Then*

$$\mathcal{R}, L_\Pi, t \models_c \varphi \quad \iff \quad \mathcal{R}^{gcd(t_0, r, \varphi)}, L_\Pi, t \models_p \varphi$$

for all states t reachable in $\mathcal{R}^{gcd(t_0, r, \varphi)}$ from t_0 .

Proof. Again, the proof is given in [19].

5 Implementation

Our model checker makes the natural and reasonable assumption that given a real-time rewrite theory \mathcal{R} , and an initial state t_0 on which we would like to check some TCTL formula φ , all behaviors starting from t_0 are *time-diverging* w.r.t. the selected time sampling strategy σ . This assumption also implies that the transition relation $\xrightarrow{\mathcal{T}}_{\mathcal{R}^\sigma}$ in the timed Kripke structure $\mathcal{TK}(\mathcal{R}^\sigma, t_0)_\Pi$ is *total*.

The current implementation of the model checker assumes that time values are either in **NAT-TIME-DOMAIN-WITH-INF** or **POSRAT-TIME-DOMAIN-WITH-INF**, and provides the user with two possible model-checking strategies:

- (i) The *basic* strategy, which performs the model checking on the model obtained by applying the user-defined time sampling strategy on the original model.

- (ii) The *gcd* strategy, which extends the maximal time sampling strategy with the “gcd” transformation to perform the model checking for the satisfaction problem $\mathcal{R}^{gcd(t_0, r, \varphi)}, L_{II}, t_0 \models_p \varphi$.

Soundness and completeness of the *gcd* strategy might come at the cost of a larger state space due to the application of the *gcd* transformation. When the *gcd* strategy is impractical, the user can still perform model checking with the generally faster basic strategy, which does not increase the system state space and can still be very useful to discover potential bugs, as illustrated below.

Real-Time Maude, and hence our model checker, is implemented in Maude, making extensive use of Maude’s meta-programming capabilities. The model checker first constructs the timed Kripke structure, according to the selected model checking strategy, by collecting all the reachable states and transitions. When the *gcd* strategy is selected, the timed Kripke structure is refined by “splitting” the transitions into smaller ones of duration equal to half the computed greatest common divisor. Then, the satisfaction sets of each subformula are recursively computed. Since the meta-representation of the states can be fairly large⁴, performing the rest of the model checking procedure on the generated timed Kripke structure is fairly inefficient. In our current implementation, we assign a unique natural number to each (meta-represented) state in the generated timed Kripke structure, and construct a more compact timed Kripke structure, where all the occurrences of these meta-represented states are replaced by their respective identifiers. We then perform the recursive computation of the satisfaction set of φ on this compact representation. This optimization led to a large performance improvement and made it feasible to apply our model checker to a number of case studies in reasonable time, whereas working directly on meta-represented terms made model checking unfeasible even for simple case studies.

Our implementation of the TCTL model checker is based on the *explicit-state* CTL model checking approach [8] that, starting with the atomic propositions, recursively computes for each subformula of the desired TCTL formula the set of satisfying reachable states. We implemented specific procedures for a basic set of temporal modal operators and we expressed other formulas into this canonical form. The basic set consists of the CTL modal operators $E \varphi_1 U \varphi_2$, $E G \varphi$, the TCTL $_{\leq \geq}$ modal operators $E \varphi_1 U_{\sim r} \varphi_2$ with $\sim \in \{>, \geq\}$, $E \varphi_1 U_{\sim r} \varphi_2$ with $\sim \in \{<, \leq\}$, $A \varphi_1 U_{>0} \varphi_2$ and the TCTL $_{cb}$ modal operator $E \varphi_1 U_{[a,b]} \varphi_2$. The procedures for CTL modalities follow the standard explicit algorithm [8]. For TCTL $_{\leq \geq}$ modalities, our implementation adapts the TCTL $_{\leq \geq}$ model checking procedure defined in [18] for time-interval structures and to timed Kripke structures with time-diverging paths.

The ease and flexibility of the Maude meta-level allowed us to implement the model checker reasonably quickly and easily. However, the convenience of operating at the meta-level comes at a certain cost in terms of computational

⁴ For example, *each* state in the Maude representation of the Ptolemy II model in Section 6 “contains” the entire Ptolemy II model.

⁵ We denote by TCTL $_{\leq \geq}$ the restricted TCTL logic with time constraints on the temporal modalities of the form $\sim r$, where $\sim \in \{<, \leq, \geq, >\}$,

efficiency, even with our optimizations. Therefore, the current Real-Time Maude model checker should be regarded as a *working prototype* for a C++ implementation that we plan to implement in the future.

Our model checker is available at <http://folk.uio.no/leprid/TCTL-RTM/> together with the technical report [19], the specifications and analysis commands of the case studies in this paper.

5.1 Using the Model Checker

In Real-Time Maude, the user is provided with two TCTL model checking commands, corresponding respectively to the basic and the gcd strategy, with syntax

$$(\text{mc-tctl } t \mid= \varphi .) \quad \text{and} \quad (\text{mc-tctl-gcd } t \mid= \varphi .)$$

for t the initial state and φ a TCTL formula. The syntax of TCTL formulas is fairly intuitive, with syntactic sugar for (untimed) CTL formulas, common abbreviations and boolean connectors such as **AF**, **EF**, **AG**, **EG**, **iff** and **implies**, etc. For example, $E \text{ true } U_{\leq r}(\neg\varphi \wedge A G (E F_{[a,b]}\varphi'))$ is written⁶

$$E \text{ tt } U[\leq \text{ than } r](\text{not } \varphi \text{ and } AG (EF[c \ a, \ b \ c] \varphi'))$$

We do not support counter-example generation, since, in contrast to linear temporal logics, where counter-examples are just paths, it is generally more complex to generate counter-examples in branching-time temporal logics, where counterexamples are parts of computation trees (see, e.g. [13]). For example, a counter-example to the validity of the formula $E F p$, for p an atomic proposition, is the entire computation tree (where each state is a $\neg p$ -state).

6 Model Checking a Ptolemy II Discrete-Event Model

Real-Time Maude provides a formal analysis tool for a set of modeling languages for embedded systems, including Ptolemy II discrete-event (DE) models that cannot be formalized by, say, timed automata. Ptolemy II [15] is a well-established modeling and simulation tool used in industry that provides a powerful yet intuitive graphical modeling language. Our model checker has been integrated into Ptolemy II by Kyungmin Bae, so that we can now model check TCTL properties of Ptolemy II DE models *from within Ptolemy*⁷. We show the TCTL analysis of a Ptolemy II model of a hierarchical traffic light system, in which our model checker has uncovered a previously unknown flaw. Notice that Ptolemy II DE models satisfy the requirements for having a sound and complete analysis when using the gcd strategy. The analysis has been performed on a 2.4GHz Intel[®] Core 2 Duo processor with 4 GB of RAM.

⁶ The model checker syntax for TCTL formulas supports also open bounds, e.g. the user could write $[c \ 0, \ b \ \circ]$ for $[0, b)$, which would be internally reduced to $[< \text{ than } b]$.

⁷ Real-Time Maude verification commands can be entered into the dialog box that pops up when the blue button in Fig. 1 is clicked.

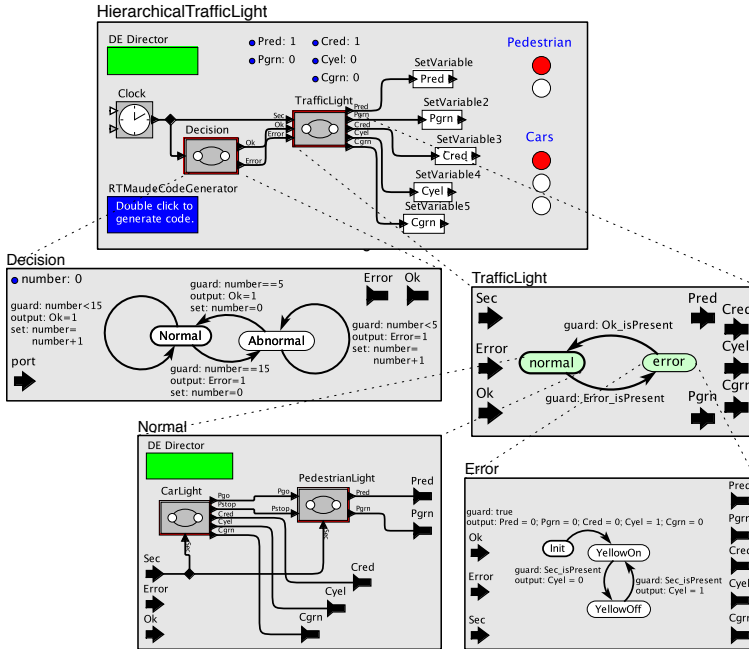


Fig. 1. A hierarchical fault-tolerant traffic light system in Ptolemy II

Figure 1 shows a hierarchical Ptolemy II model of a fault-tolerant traffic light system at a pedestrian crossing, consisting of one car light and one pedestrian light. Each light is represented by a set of *set variable* actors (Pred and Pgrn represent the pedestrian light, and Cred, Cyel and Cgrn represent the car light). A light is *on* iff the corresponding variable equals 1. The FSM actor Decision “generates” failures and repairs by alternating between staying in location Normal for 15 time units and staying in location Abnormal for 5 time units. Whenever the model operates in *error* mode, all lights are turned off, except for the yellow light of the car light, which is blinking. We refer to [7] for a thorough explanation of the model.

An important fault tolerance property is that the car light will turn yellow, *and only yellow*, within 1 time unit of a failure. We can model check this bounded response property with the command:

```
Maude> (mc-tctl {init} /=
  AG((('HierarchicalTrafficLight . 'Decision | (port 'Error is present))
  implies AF[<= than 1] ('HierarchicalTrafficLight |
    ('Cyel = # 1, 'Cgrn = # 0, 'Cred = # 0)))) .)
```

In about 15 secs, the command returns that the property is not satisfied. This model checking uncovered a previously unknown scenario, which shows that,

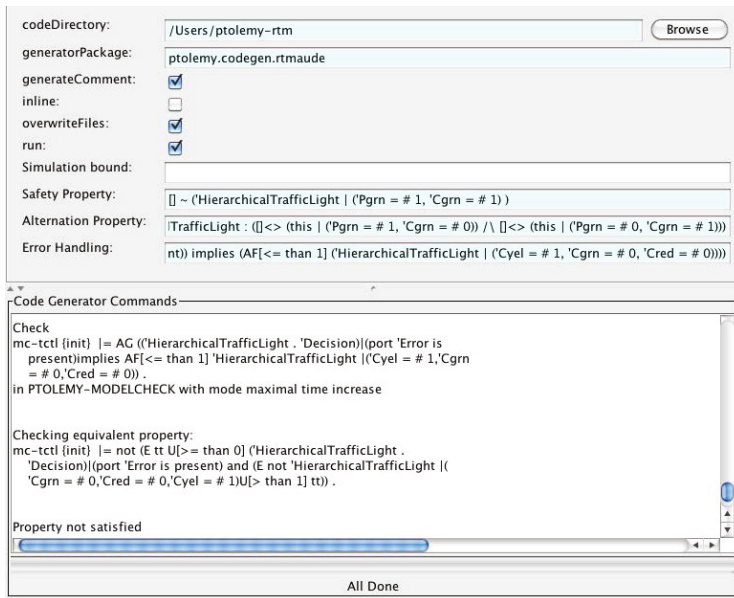


Fig. 2. Dialog window for the hierarchical traffic light code generation

after a failure, the car light may show red or green in addition to blinking yellow. Eleven of the 15 seconds used by the timed CTL model checker were used to generate the timed Kripke structure. Because of the large size of the system states in this case study, it was impossible to run the same analysis before implementing the optimization that mapped each state to an unique identifier. The same property can be model checked with the gcd strategy command `mc-tks-gcd` in about 22 secs.

Using the gcd strategy we can also determine a “minimal” time interval such that the above bounded response is satisfied in the system. In particular, we discovered that this interval is [5, 12] by trying different values for *a* and *b* in the interval-bounded command

```
Maude> (mc-tctl {init} |=
  AG(('HierarchicalTrafficLight . 'Decision | (port 'Error is present))
    implies AF[c a, b c] ('HierarchicalTrafficLight |
      ('Cyel = # 1, 'Cgrn = # 0, 'Cred = # 0)))) .)
```

Figure 2 shows the dialog window for the Real-Time Maude code generation of the hierarchical traffic light model: after entering the *error handling* property, a simple click on the **Generate** button will display the result of the model checking command execution in the “Code Generator Commands” box.

7 Conclusions and Future Work

We have described the semantic foundations of our TCTL model checker for Real-Time Maude. Our modeling formalism is more expressive than those of other timed model checkers, allowing us to analyze real-time systems which are beyond the scope of other verification tools. In particular, we have proved soundness and completeness of our model checker for a class of dense-time Real Time Maude specifications that contain many systems outside the scope of other real-time model checkers. Furthermore, the introduced TCTL model checker also provides for free a timed temporal logic model checker for interesting subsets of modeling languages widely used in industry, such as Ptolemy II and the avionics standard AADL.

So far, we have only proved soundness and completeness for formulas with closed intervals under the continuous semantics. We should also cover formulas with open time intervals and the pointwise semantics. The model checker should also provide counter-examples in a user-friendly way, when possible. We should also extend our model checker to *time-bounded* TCTL model checking to support the model checking of systems with infinite reachable state space. Finally, the current version of the tool is implemented at the Maude meta-level; for efficiency purposes, it should be implemented in C++ in the Maude engine.

References

1. van der Aalst, W.M.P.: Interval Timed Coloured Petri Nets and their Analysis. In: Ajmone Marsan, M. (ed.) ICATPN 1993. LNCS, vol. 691, pp. 453–472. Springer, Heidelberg (1993)
2. Alturki, M., Meseguer, J.: Real-time rewriting semantics of Orc. In: Proc. PPDP 2007. ACM (2007)
3. Alur, R., Dill, D.L.: A theory of timed automata. *Theoretical Computer Science* 126(2), 183–235 (1994)
4. Alur, R., Henzinger, T.: Logics and Models of Real Time: A Survey. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 74–106. Springer, Heidelberg (1992)
5. Alur, R., Courcoubetis, C., Dill, D.: Model-checking in dense real-time. *Inf. Comput.* 104, 2–34 (1993)
6. Bae, K., Ölveczky, P.C., Al-Nayeem, A., Meseguer, J.: Synchronous AADL and Its Formal Analysis in Real-Time Maude. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 651–667. Springer, Heidelberg (2011)
7. Bae, K., Ölveczky, P.C., Feng, T.H., Lee, E.A., Tripakis, S.: Verifying hierarchical Ptolemy II discrete-event models using Real-Time Maude. *Science of Computer Programming* (to appear, 2012), doi:10.1016/j.scico.2010.10.002
8. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
9. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
10. Boronat, A., Ölveczky, P.C.: Formal Real-Time Model Transformations in MOMENT2. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 29–43. Springer, Heidelberg (2010)

11. Boucheneb, H., Gardey, G., Roux, O.H.: TCTL Model Checking of Time Petri Nets. *J. Logic Computation* 19(6), 1509–1540 (2009)
12. Bouyer, P.: Model-checking timed temporal logics. *ENTCS* 231, 323–341 (2009)
13. Clarke, E.M., Grumberg, O., McMillan, K.L., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: *DAC 1995* (1995)
14. 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)
15. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE* 91(2), 127–144 (2003)
16. Gardey, G., Lime, D., Magnin, M., Roux, O.(H.): Romeo: A Tool for Analyzing Time Petri Nets. In: Etessami, K., Rajamani, S.K. (eds.) *CAV 2005*. LNCS, vol. 3576, pp. 418–423. Springer, Heidelberg (2005)
17. Katelman, M., Meseguer, J., Hou, J.: Redesign of the LMST Wireless Sensor Protocol through Formal Modeling and Statistical Model Checking. In: Barthe, G., de Boer, F.S. (eds.) *FMOODS 2008*. LNCS, vol. 5051, pp. 150–169. Springer, Heidelberg (2008)
18. Laroussinie, F., Markey, N., Schnoebelen, P.: Efficient timed model checking for discrete-time systems. *Theor. Comput. Sci.* 353, 249–271 (2006)
19. Lepri, D., Ölveczky, P.C., Ábrahám, E.: Timed CTL model checking in Real-Time Maude, <http://folk.uio.no/leprid/TCTL-RTM/tctl-rtm2011.pdf>
20. Lepri, D., Ölveczky, P.C., Ábrahám, E.: Model checking classes of metric LTL properties of object-oriented Real-Time Maude specifications. In: *Proc. RTRTS 2010*. *EPTCS*, vol. 36, pp. 117–136 (2010)
21. Lien, E., Ölveczky, P.C.: Formal modeling and analysis of an IETF multicast protocol. In: *Proc. SEFM 2009*. IEEE Computer Society (2009)
22. Markey, N., Schnoebelen, P.: TSMV: A Symbolic Model Checker for Quantitative Analysis of Systems. In: *QEST*. IEEE Computer Society (2004)
23. Morasca, S., Pezzè, M., Trubian, M.: Timed high-level nets. *The Journal of Real-Time Systems* 3, 165–189 (1991)
24. Ölveczky, P.C.: Semantics, Simulation, and Formal Analysis of Modeling Languages for Embedded Systems in Real-Time Maude. In: Agha, G., Danvy, O., Meseguer, J. (eds.) *Formal Modeling: Actors, Open Systems, Biological Systems*. LNCS, vol. 7000, pp. 368–402. Springer, Heidelberg (2011)
25. Ölveczky, P.C., Boronat, A., Meseguer, J.: Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude. In: Hatcliff, J., Zucca, E. (eds.) *FMOODS/FORTE 2010, Part II*. LNCS, vol. 6117, pp. 47–62. Springer, Heidelberg (2010)
26. Ölveczky, P.C., Caccamo, M.: Formal Simulation and Analysis of the CASH Scheduling Algorithm in Real-Time Maude. In: Baresi, L., Heckel, R. (eds.) *FASE 2006*. LNCS, vol. 3922, pp. 357–372. Springer, Heidelberg (2006)
27. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science* 285, 359–405 (2002)
28. Ölveczky, P.C., Meseguer, J.: Abstraction and completeness for Real-Time Maude. *Electronic Notes in Theoretical Computer Science* 176(4), 5–27 (2007)
29. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation* 20(1-2), 161–196 (2007)
30. Ölveczky, P.C., Meseguer, J.: The Real-Time Maude Tool. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008*. LNCS, vol. 4963, pp. 332–336. Springer, Heidelberg (2008)

31. Ölveczky, P.C., Meseguer, J., Talcott, C.L.: Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design* 29(3), 253–293 (2006)
32. Ölveczky, P.C., Thorvaldsen, S.: Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. *Theoretical Computer Science* 410(2-3), 254–280 (2009)
33. Riesco, A., Verdejo, A.: Implementing and analyzing in Maude the enhanced interior gateway routing protocol. *Electr. Notes Theor. Comput. Sci.* 238(3), 249–266 (2009)
34. Rivera, J.E., Durán, F., Vallecillo, A.: On the Behavioral Semantics of Real-Time Domain Specific Visual Languages. In: Ölveczky, P.C. (ed.) *WRLA 2010. LNCS*, vol. 6381, pp. 174–190. Springer, Heidelberg (2010), see also the e-Motions web page http://atenea.lcc.uma.es/index.php/Main_Page/Resources/E-motions
35. Wang, F.: Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE* 92(8), 1283–1307 (2004)
36. Wang, F.: REDLIB for the formal verification of embedded systems. In: *Proc. ISoLA 2006*. IEEE (2006)
37. Wirsing, M., Bauer, S.S., Schroeder, A.: Modeling and analyzing adaptive user-centric systems in Real-Time Maude. In: *Proc. RTRTS 2010. EPTCS*, vol. 36, pp. 1–25 (2010)
38. Yovine, S.: Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer* 1(1-2), 123–133 (1997)

Using Narrowing to Test Maude Specifications^{*}

Adrián Riesco

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

Abstract. Testing is one of the most important and most time-consuming tasks in the software developing process and thus techniques and systems to automatically generate and check test cases have become crucial. In previous work we have presented techniques to test membership equational logic specifications; these techniques consist of two steps: first several ground terms are generated by using all the available constructor symbols in a breadth-first search, and then these terms are processed to check whether they fulfill some properties. This approach presents the drawback of separating two related processes, thus examining several terms that are indistinguishable from the point of view of testing. We present here a narrowing-based test-case generator that improves the performance of the tool and extends its use to rewriting logic specifications. First, we present two mechanisms to improve the narrowing commands currently available in Maude to use conditional statements and equational modules. Then, we show how to use these new narrowing commands to perform three different approaches to testing for any Maude specification: code coverage, property-based testing, and conformance testing. Finally, we present trusting mechanisms to improve the performance of the tool. We illustrate the tool by means of an example.

Keywords: testing, Maude, narrowing, coverage, property, conformance.

1 Introduction

Testing is a technique for checking the correctness of programs by means of executing several inputs and studying the obtained results. Testing is one of the most important stages of the software-development process, but it also is a very time-consuming and tedious task, and for this reason several efforts have been devoted to automate it [21]. Basically, we can distinguish two different approaches to testing: glass-box testing [13,24], that uses the specific statements of the system to generate the most appropriate test cases, and black-box testing [32,14,5], that considers the system as a black box with an unknown structure and where a specification of the system is used to generate the test cases and check their correctness. We can also distinguish different kinds of testing depending on how the test cases are obtained: they can either be ground terms

^{*} Research supported by MEC Spanish project *DESAFIOS10* (TIN2009-14599-C03-01) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC1465).

that are later executed to check the obtained results or terms with variables that are symbolically executed [20] to find the most appropriate values to test the program. While the former generates in general more test cases (because it just combines constructors to build terms) they can be illegal (input that can never be used in real executions) and equivalent (different test cases check the same statements), the latter generates less but more accurate test cases.

Maude [8] is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Maude modules correspond to specifications in rewriting logic [22]. This logic is an extension of equational logic; in particular, Maude functional modules correspond to specifications in membership equational logic [3], which, in addition to equations, allows the statement of membership axioms characterizing the elements of a sort. Rewriting logic extends membership equational logic by adding rewrite rules, that represent transitions in a concurrent system. Maude system modules are used to define specifications in this logic. The current version of Maude supports a limited version of narrowing [31], a generalization of term rewriting that allows to execute terms with variables by replacing pattern matching by unification, for some *unconditional rewriting logic theories without memberships*. This limitation is dropped in this work by using a program transformation and by checking separately the conditions.

As part of an ongoing project to test and debug Maude specifications, we have implemented a declarative debugger for Maude specifications [28], that allows the user to debug both wrong (incorrect results obtained from a valid input) and missing (incomplete results obtained from a valid input) answers in any Maude specification, and a test case generator for functional modules [27]. The testing approach used in that paper consists of different phases: first, the module is preprocessed to obtain the statements used by the functions being tested; then, terms are generated by using a breadth-first search that takes into account the constructor information provided by the user, and then each of these terms is executed step-by-step to check the used statements. However, this approach uses ground terms and, as explained above, presents an important drawback: since the test cases are not generated following the structure of the program but just the available constructors, most of them apply the same statements, hence consuming most of the time and preventing more complex terms from being checked due to the time and space constraints. This problem is solved here by symbolically executing terms with variables with narrowing.

We present in this paper a program transformation to test Maude functional modules by using narrowing, a strategy to use membership axioms and conditional statements in the narrowing process,¹ and the adaptation of three testing techniques to Maude: two white-box approaches (one selects a set of test cases whose correctness must be checked by the user, while the other one checks

¹ This strategy allows all kinds of conditions: rewrite and equational conditions, solved by narrowing (the latter, which includes equational and matching conditions, requires a previous transformation), and membership conditions, solved by using unification.

whether a property holds in the specification) and one black-box mechanism (conformance testing). In the first case, in addition to other criteria described in [27], we have adapted a new criterion to select the set of test cases to be checked by the user in system modules, which is based on modified condition decision coverage [19] and checks the negative information (the rules that are not applied). Finally, we enhance the performance of the tool by providing trusting techniques that prevent the system from taking into account some statements. The transformation, the extension of the narrowing process, and the testing strategies have been implemented in a Maude prototype by using its meta-level capabilities, that allow to manipulate Maude modules and statements as usual data. Moreover, it also provides support for some predefined modules and attributes, such as `owise`, that indicates that the current equation is only used when the rest of equations cannot be applied.

The rest of the paper is organized as follows: Section 2 presents some related work and its relation with our system. Section 3 introduces Maude and narrowing, Section 4 describes a module transformation that allows us to use narrowing on Maude functional modules, while Section 5 presents how to use conditional rules in the narrowing process. Section 6 illustrates how the techniques described in the previous sections are used to generate test cases, while Section 7 presents some trusting techniques to improve the performance of the system. Finally, Section 8 concludes and outlines some future work. The source code of the tool, examples, related papers, and much more information is available at <http://maude.sip.ucm.es/testing/>.

2 Related Work

Different approaches to testing for declarative languages have been proposed in the literature. As explained in the introduction, test cases can be checked in different ways: executing ground test cases or symbolically executing terms with variables.

The first approach is followed by Smallcheck [30], a property-driven Haskell tool that considers that most of the errors can be found by using only a few constructors, and thus it generates all the possible combinations of constructors given a (usually small) bound on the size of the test cases. Another tool following this ground approach is Quickcheck [7], a test-case generator developed for Haskell specifications where the programmer writes assertions about logical properties that a function should fulfill; test cases are randomly generated by using the constructors of the data type (in contrast to the complete search performed by Smallcheck) to test and attempt to falsify these assertions. The project, started in 2000, has been extended to generate test cases for several languages such as Java, C++, Erlang, and several others. Finally, Easycheck [6] is a test-case generator for Curry that takes advantage of the non-determinism of functional-logic programs to generate a tree of potential test cases, that is later traversed to list only the most interesting ones.

The second approach has been applied by Lazy Smallcheck [30] (an improvement of a previous system called SparseCheck), a library for Haskell to test

partially-defined values that uses a mechanism similar to narrowing to test whether the system fulfills some requirements. Another way of achieving symbolic execution is by considering that the statements in the program under test introduce constraints on the variables, an approach followed by PET [15], that uses Constraint Logic Programming to generate test cases satisfying some coverages on object-oriented languages. Finally, narrowing has been used to verify security protocols [21,18], symbolically exploring the state space trying to find a flow in the protocol.

The previous version of our approach is quite similar to Smallcheck: we generate the complete search space given the constructors and a bound, but we use them for both white-box and black-box testing, while Smallcheck only tries to disprove some properties. Note that, on the one hand, the strategies in our previous system could possibly be improved by following an approach similar to Easycheck, while on the other hand we can consider that the current narrowing approach is another way of pruning the tree of possible terms, making our approach similar to it. Regarding Quickcheck, it is an industrial tool with several heuristics and a lot of experience in testing, and hence it presents a better performance than our tool, that we try to improve by providing trusting mechanisms to the user. On the other hand, an advantage of our tool is the computation of test cases fulfilling different coverage criteria, which allows the user to test the specification by checking test cases “by hand” even when no properties over the specification are stated, and the usage of Maude as both a specification and implementation language, which allows to perform conformance testing using a previously tested Maude module as specification. Moreover, both Quickcheck and our tool implement the *shrinking* mechanism, that consists of returning the simplest form of a term that detects a bug in the program; in our case it is implemented by performing a breadth-first search using narrowing steps, that will find the simplest term (w.r.t. the number of steps) reproducing the buggy behavior. The more similar approach to ours is Lazy Smallcheck; both are narrowing-based experimental tools that focus on research rather than in efficiency, and thus they present a similar performance; however, Smallcheck is only applied to property-based testing. PET provides a coverage of the statements in Java-like programs, but it does not allow the user to state properties or check the correctness of the system against a specification. Finally, the verification of security protocols focus on a specific problem and cannot be compared with the rest of tools.

Note that, in general, each system only focus in one testing approach: coverage, properties, or conformance. Maude features allow us to implement a wide range of testing techniques: we can manipulate its modules to perform white-box testing by using its meta-level capabilities; its analysis tools (such as the `search` command) ease the testing of properties; and Maude programs can be used as specification of others.

3 Preliminaries

This section introduces Maude and its narrowing mechanisms [9].

3.1 Maude

Maude functional modules [8, Chap. 4], with syntax `fmod ... endfm`, are executable membership equational specifications that allow the definition of sorts (by means of keyword `sort(s)`); subsort relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`). Maude system modules [8, Chap. 6], introduced with syntax `mod ... endm`, are executable rewrite theories [22]. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`).

An important characteristic of Maude functional modules is that sorts are grouped into equivalence classes called *kinds*; that is, all the sorts related by a subsort relation belong to the same kind [8]. Intuitively, terms with a kind but without a sort represent undefined or error elements. We will make extensive use of kinds to indicate that variables may have any sort when performing unification; the proper sorts of the variables will be later checked by means of membership axioms.

We introduce Maude modules with an example; variable declarations are not shown because of space constraints, but assume they are defined at the sort level. We specify ordered lists of natural numbers in the following module:

```
(fmod SORTED-NAT-LIST is
  pr NAT .
```

We use the sort `NatList`, with constructors `nil` and `_. _`, for generic lists and `SortedList` for sorted lists, which are a subsort of `NatList`:

```
  sorts SortedList NatList .      subsorts SortedList < NatList .
  op nil : -> SortedList [ctor] . op _ _ : Nat NatList -> NatList [ctor] .
```

We use membership axioms to characterize nonempty sorted lists. They indicate that the singleton list is ordered (`o11`) and that a larger list is ordered if the first element is equal to or smaller than the second one and the rest of the list is also ordered (`o12`):

```
  mb [o11] : N . nil : SortedList .
  cmb [o12] : N . N' . L : SortedList if N <= N' /\ N' . L : SortedList .
```

We also specify a function `ins-sort` that sorts a list by inserting the elements in an ordered fashion by using the auxiliary function `ins-list`:

```
  op ins-sort : NatList -> SortedList .
  eq [is1] : ins-sort(nil) = nil .
  eq [is2] : ins-sort(N . L) = ins-list(ins-sort(L), N) .
```

This function returns the singleton list when inserting an element into the empty list, and otherwise it distinguishes whether the first element in the list is smaller or not:

```

op ins-list : SortedList Nat -> SortedList .
eq [il1] : ins-list(nil, N) = N . nil .
ceq [il2] : ins-list(N . SL, N') = N' . (N . SL) if N' <= N .
ceq [il3] : ins-list(N . SL, N') = N . ins-list(SL, N') if N < N' .

```

Since we are also interested on testing system modules, we use this module to specify how processes enter into a critical section in the following system module `CS`. We consider that processes are represented by their priority (the smaller the number the higher the priority), and hence lists of natural numbers stand for lists of processes:

```

(mod CS is
  pr SORTED-NAT-LIST .

```

We define the sort `NatSoup` for an associative and commutative multiset built with the operators `mtSoup` and `_,_;` the sort `NatWithEmpty` for a supersort of the natural numbers with an extra element `empty`; and `System` for the system, that receives as arguments a multiset of natural numbers (the idle processes), a sorted list of numbers (the processes waiting to enter the critical section), a value of sort `NatWithEmpty` (the process in the critical section), and another multiset of numbers (the processes that have already entered the critical section):

```

sort System NatSoup NatWithEmpty .      subsort Nat < NatSoup NatWithEmpty .
op empty : -> NatWithEmpty [ctor] .      op mtSoup : -> NatSoup [ctor] .
op _,_ : NatSoup NatSoup -> NatSoup [ctor assoc comm id: mtSoup] .
op _[_][_]: NatSoup NatList NatWithEmpty NatSoup -> System [ctor] .

```

We use the rule `ticket` to introduce a new process into the list of waiting processes:

```

r1 [ticket] : (N, NS) [NL] [NWE] NS' => NS [ins-list(NL, N)] [NWE] NS' .

```

If at least one process is waiting to enter the critical section and it contains the value `empty`, then the first process in the list is introduced into the critical section:

```

r1 [cs-in] : NS [N . NL] [empty] NS' => NS [NL] [N] NS' .

```

The rule `cs-out` moves the process from the critical section to the finished section:

```

r1 [cs-out] : NS [NL] [N] NS' => NS [NL] [empty] (N, NS') .

```

Finally, the rule `reset` moves the elements in the fourth component of the system to the first one to start the process again:

```

r1 [reset] : mtSoup [nil] [empty] NS => NS [nil] [empty] mtSoup .
endm)

```

3.2 Narrowing

Narrowing [31,12,23] is a generalization of term rewriting that allows free variables in terms and replaces pattern matching by unification in order to reduce

these terms. It was first used for solving equational unification problems [29] and then generalized to deal with problems of symbolic reachability. Similarly to rewriting, where at each rewriting step one must choose which subterm of the subject term and which rule of the specification are going to be considered, at each narrowing step one must choose which subterm of the subject term, which rule of the specification, and which instantiation on the variables of the subject term and the rule's lefthand 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 $l \Rightarrow r$ to rewrite t at a position p , but narrowing unifies the lefthand side l and the chosen subject term t before actually performing the rewriting step, while in rewriting this term must be an instance of l (i.e., only matching is required). Using this narrowing approach, we can obtain a substitution that, applied to an initial term that only contains variables (except for the function symbol at the top), generates the most general term that can apply the traversed rules.

We denote by $t \rightsquigarrow^\sigma t'$, with $\sigma = q_1; \dots; q_n$ a sequence of labels, the succession of narrowing steps applying (in the given order) the statements $q_1; \dots; q_n$ that leads from the initial term t (possibly with variables) to the term t' , and by θ_σ the substitution used by this sequence, which results from the composition of the substitutions obtained in each narrowing step. We will overload the notation $t \rightsquigarrow^q t'$ by using conditions in q to illustrate narrowing steps due to conditions.

In the example above, we could start from the term `NS1 [NL] [NWE] NS2`, with `NS1` and `NS2` variables of sort `NatSoup`, `NL` a variable of sort `NatList`, and `NWE` a variable of sort `NatWithEmpty`, and apply one step of narrowing to obtain a set of four terms, each of them corresponding to the application of one of the rules for `System`. For example,

$$\text{NS1 [NL] [NWE] NS2} \rightsquigarrow^{\text{ticket}} \text{NS3 [ins-list(NL, N1)] [NWE] NS2}$$

where `NS1` has been replaced by `N1`, `NS3` (with `N1` and `NS3` fresh variables of sorts `Nat` and `NatSoup`, respectively) and then the rule `ticket` has been applied.

The latest version of Maude includes an implementation of narrowing for free, C, AC, or ACU theories in Full Maude [9]. More specifically, we are interested in the `metaNarrowSearchPath` function that, given a term and a bound on the number of narrowing steps, returns all the possible paths starting from this term, the used substitutions, and the applied rules. We use this command to perform a breadth-first search of the state space. Note that the current implementation of narrowing only works for non-conditional rules and specifications without membership axioms; we will show in Section 5 how to check separately the conditions, including membership conditions.

4 A Module Transformation for Narrowing

We present in this section a simple module transformation that will be applied to the modules in order to use narrowing with the equational part of Maude.

This transformation has two objectives: on the one hand it transforms equations into rules (and thus it requires to transform equational conditions into rewrite conditions), which allows us to use narrowing with the equational part of Maude system modules. On the other hand, and since the current implementation of narrowing in Maude does not support memberships, we transform all the terms where membership inferences may be needed into equivalent terms with the variables declared at the kind level, while extra membership conditions stating the correct sort, that will be separately checked with the mechanisms in the next section, are added for each variable whose type has changed. More specifically, the transformation takes an equation of the form

$$l = r \text{ if } \bigwedge_{i=1}^n t_i = t'_i \wedge \bigwedge_{j=1}^m p_j := u_j \wedge \bigwedge_{k=1}^l v_k : s_k$$

and returns a rule

$$\begin{aligned} \text{kind}(l) \Rightarrow \text{kind}(r) \text{ if } & \text{mbs}(l) \wedge \\ & \bigwedge_{i=1}^n (\text{kind}(t_i) \Rightarrow w_i \wedge \text{kind}(t'_i) \Rightarrow w_i) \wedge \\ & \bigwedge_{j=1}^m (\text{kind}(u_j) \Rightarrow \text{kind}(p_j) \wedge \text{mbs}(p_j)) \wedge \\ & \bigwedge_{k=1}^l \text{kind}(v_k) : s_k \end{aligned}$$

where

- The terms w_i are fresh variables of the same kind as the corresponding term.
- The function kind replaces the sort of all the variables in the term given as argument by the corresponding kind (we follow here the Maude approach that represents each variable as a pair of an identifier and a type, that can be either a sort or a kind; thus, we can modify these pairs when the second component is a sort by the appropriate kind).
- The function mbs generates a conjunction of conditions stating that the variables, whose type has been changed by its kind, have in fact the sort previously required, that is:

$$\begin{aligned} \text{mbs}(f(t_1, \dots, t_n)) &= \text{mbs}(t_1) \wedge \dots \wedge \text{mbs}(t_n) \\ \text{mbs}(c) &= \text{nil} \\ \text{mbs}(v) &= \text{kind}(v) : \text{sort}(v) \end{aligned}$$

where f is a function symbol, the t_i are terms, c is a constant, v is a variable, and $\text{sort}(v)$ returns the sort of v .

We have to transform similarly all the membership axioms and rules in the module in order to apply them. In the membership case we obtain another membership axiom with the lefthand side and the condition transformed as shown above² while rules are transformed into rules, being the equational part transformed as in the previous cases while the rewriting conditions remain unchanged.

² Note that this transformation may generate invalid membership axioms, because they may contain rewrite conditions. However, in practice all the equations and rules in the module are unconditional and the membership axioms have been removed in order to use narrowing; these conditions and membership axioms are kept apart and checked separately by using the techniques described in Section 5.

Note that, since Maude equational modules are assumed to be confluent and terminating, the equations may be understood as oriented from left to right, which is what we are explicitly doing when transforming them into rules. Moreover, the kind transformation only postpones (but not prevents from) the checking of the specific sorts of the variables to the condition of the rule. For these reasons, it is straightforward to see that this transformation is correct, even though it can only be executed by using narrowing as explained in the next section.

If we transform the critical section example above, the membership axiom `o12` is modified as follows (assume that the variables are now declared at the kind level):

```
cmb [o12] : N . N' . L : SortedList
  if N : Nat /\ N' : Nat /\ L : NatList /\
     N <= N' => B /\ true => B /\
     N' . L : SortedList .
```

The first three conditions indicate that the variables, that are now declared at the kind level, have in fact the appropriate sort. The next two conditions deal with the first condition of the original axiom, `N <= N'`, which is an abbreviation for `N <= N' = true`; in this case both sides of the equality must be rewritten to the same variable `B`, defined in the kind of `Bool`. Finally, the membership condition remains unchanged.

In a similar way, the equation `i12` is transformed into the following rule:

```
crl [i12] : ins-list(N . SL, N') => N' . N . SL
  if N : Nat /\ SL : SortedList /\ N':Nat /\
     N' <= N => B /\ true => B .
```

where the first three conditions indicate that the variables have the appropriate sort. The next two conditions deal with the condition of the original axiom, `N' <= N''`, which is an abbreviation for `N' <= N'' = true`; in this case both sides of the equality must be rewritten to the same variable `B`, defined in the kind of `Bool`.

5 Narrowing of Conditional Rules

We present in this section a methodology to take into account the conditions in the narrowing process because, as explained in the introduction, they are not supported by the current implementation of the Maude system. Note that other systems deal with rewrite conditions (see e.g. [23,16]) with a similar approach to ours: they must be solved before applying the body of the rule. The novelty of our technique, beyond describing and implementing this narrowing of conditional rules in Maude, lies on the resolution of membership conditions by means of unification.

Basically, when a rule is applied the conditions must be evaluated separately by using narrowing (remember that equational conditions become rewrite conditions) to find a substitution (that must be the composition of the substitutions

obtained for each single condition) that fulfills them. If the set of substitutions fulfilling the conditions is nonempty, all of them extend the set of substitutions obtained for the unconditional rule; otherwise, the rule cannot be applied.

However, in addition to rewrite conditions we must also take into account membership conditions. The current implementation of narrowing does not support membership axioms, and thus we must independently check whether a membership condition holds. The first step to achieve it was presented in the previous section: we transform the lefthand of the statements to deal with kinds instead of sorts in order to move the membership information to the conditions. The next step consists of checking the memberships (those introduced by the transformation, as well as those stated by the user); if the sort is defined by using membership axioms (and possibly by operators), then we unify the current term with the lefthand side of each membership axiom inferring this sort or any of its subsorts and then we proceed to prove the conditions in the corresponding axioms as explained before, applying the substitution obtained in the unification (moreover, it also updates the type of the variables, if they are at the kind level, to the required sorts in order to use the operator definitions, see the example below for details). Otherwise (the sort is not defined by using memberships) we update the type of the variables and the rest of the condition is processed.

In our example, we can apply conditional narrowing to `ins-list(NL, N1)`. The narrowing process would start by unifying this term with the lefthand side of `i12`³ whose transformed version was presented at the end of the previous section:

$$\text{ins-list}(\text{NL}, \text{N1}) \rightsquigarrow^{\text{unif-lhs}(i12)} \text{ins-list}(\text{N2} . \text{SL1}, \text{N1})$$

This first step requires the initial list of natural numbers `NL` to be of the form `N2 . SL1`, being `N2` and `SL1` fresh variables at the kind level. Thus, the unification generates the substitution $\text{NL} \mapsto \text{N2} . \text{SL1}$. However, it must be extended by using the conditions of the applied rule. The first condition, `N : Nat`, is a membership condition for a sort that is not defined with membership axioms, and thus it forces the variable `N2`⁴ to have sort `Nat`; we change the sort of the variable and proceed with the next condition. The second condition, `SL : SortedList`, is trickier because this sort is defined by means of membership axioms. We must use a transformed version of the membership axiom `o12` to obtain:

$$\text{ins-list}(\text{N2} . \text{SL1}, \text{N1}) \rightsquigarrow^{\text{unif-lhs}(o12)} \text{ins-list}(\text{N2} . \text{N3} . \text{NL2}, \text{N1})$$

where the unification of the term with the lefthand side of the membership axiom gives the substitution $\text{SL1} \mapsto \text{N3} . \text{NL2}$. Note that the transformation of `o12` generates three initial conditions (`N : Nat` \wedge `N' : Nat` \wedge `L : NatList`) that

³ Note that other rules, such as `i11` or `i13`, could be also used. In the same way, some other steps in this example could apply different membership axioms and rules.

⁴ Note that, after the unification, the rule is being symbolically applied by using the substitution $\text{N} \mapsto \text{N2}$; $\text{SL} \mapsto \text{SL1}$; $\text{N}' \mapsto \text{N1}$. In the following, we will not show the substitution required to apply each rule.

just update the sorts of the variables, two rewrite conditions, $N \leq N' \Rightarrow B$ and $\text{true} \Rightarrow B$, which require narrowing again to be solved, and keeps the membership condition unmodified. As we will explain below, since our implementation supports some predefined operators such as \leq , that returns `true` when its first argument is 0, we can use narrowing to solve the rewrite conditions:

$$\text{ins-list}(N2 . N3 . NL2, N1) \rightsquigarrow^{N \leq N' \Rightarrow B} \text{ins-list}(0 . N3 . NL2, N1)$$

and the current substitution is extended with $N2 \mapsto 0 ; B \mapsto \text{true}$. With this substitution the next condition of `ol2` ($B \Rightarrow \text{true}$) trivially holds and only the membership condition, $N' . L : \text{SortedList}$, remains. It can be satisfied by using the membership axiom `ol1`, which extends the substitution with $NL2 \mapsto \text{nil}$. Summarizing the narrowing process thus far, starting from the term `ins-list(NL, N1)` and applying the rule `il2` and its two first conditions (which includes applying the membership axiom `ol2` and all its conditions, the rule for \leq , and the membership axiom `ol1`), we have reached `ins-list(0 . N3 . nil, N1)` with the substitution $NL \mapsto 0 . N3 . \text{nil}$. We proceed now with the third condition of `il2`, $N' : \text{Nat}$, that simply updates the sort of $N1$. The next condition, $N' \leq N \Rightarrow B$, is solved as explained above by using the substitution $N1 \mapsto 0 ; B \mapsto \text{true}$:

$$\text{ins-list}(0 . N3 . \text{nil}, N1) \rightsquigarrow^{N1 \leq 0 \Rightarrow B} \text{ins-list}(0 . N3 . \text{nil}, 0)$$

Finally, the last condition for `il2` holds because `true` is rewritten to itself, and the rule is applied by using the obtained substitution in the righthand side, obtaining the following complete narrowing step with the substitution $NL \mapsto 0 . N3 . \text{nil} ; N1 \mapsto 0$:

$$\text{ins-list}(NL, N1) \rightsquigarrow^{\text{il2}} 0 . 0 . N3 . \text{nil}$$

5.1 A Brief Note on Predefined Functions

As we have shown in the example above, we use some predefined functions on the narrowing process. We basically add some rules to deal with the most used functions for boolean values and natural numbers. For example, we add the rules

```
r1 [let1] : 0 <= s(N) => true .
r1 [let2] : s(N1) <= s(N2) => N1 <= N2 .
r1 [let3] : s(N) <= 0 => false .
```

to deal with the `_<=_` function. In this way we introduce rules are easily managed by the narrowing mechanisms and greatly increase the number of Maude specifications that can be tested with the tool.

6 Using Narrowing to Generate Test Cases

Different testing techniques can be used to test Maude specifications, and for each of these techniques a different narrowing strategy will be used. We show

in this section how to compute a coverage, how to check whether a specification fulfills an invariant, and how to examine if, given a correct specification, another module performs the required actions, which is called *conformance testing*.

6.1 Coverage Criteria

Code coverage techniques [19,25] consist of selecting a set of test cases that, when executed, apply all the statements required by the coverage criterion. In our case we use *global branch coverage* [13], a strategy that tries to find test cases that use all the statements potentially used by the function under test (which, of course, includes the functions and membership axioms in the conditions) and has been already described for functional modules in [27], and *system coverage*, an adaptation of modified condition decision coverage [19] that tries to obtain information by making the conditions to fail.

Narrowing can be naturally used to compute global branch coverage by starting with a term with variables and performing a breadth-first search, where after each narrowing step, that computes the set of reachable terms by applying one rule, we check that the conditions of each rule are fulfilled by using the mechanism presented in the previous section, thus removing some of the obtained terms and extending the substitutions when required (e.g., in the example of the previous section, the substitution was extended to $NL \mapsto 0 . N3 . \text{nil} ; N1 \mapsto 0$). This search finishes when all the statements required by the coverage have been used, a bound in the number of steps has been reached, or all the possible states have been reached (this last point is checked by trying to unify the terms obtained in each step with any of the previous terms; that is, we build a graph instead of a tree). Moreover, our system provides two different options to select the set of test cases: the smallest one, composed of the minimum number of terms whose execution leads to the execution of all the statements in the coverage and thus may contain complex test cases; and the simplest one, in the sense that it may present more but simpler test cases. The user can switch between these two modes to decide which one is more appropriate for each specification.

More formally, we look for a set of sequences σ_i and terms t_i , $0 \leq i \leq l$, such that, given the set of labels Q defining the coverage and a term $f(v_1, \dots, v_n)$, with f the function under test and v_i variables of the appropriate sorts, $\forall q \in Q \exists \exists_{i=0}^l . f(v_1, \dots, v_n) \rightsquigarrow^{\sigma_i} t_i \wedge q \in \sigma_i$. The test cases will be $\bigcup_{i=0}^l \theta_{\sigma_i}(f(v_1, \dots, v_n))$. Since there are several different possibilities to select the σ_i , the different strategies to display the set of test cases will choose between a small number of large sequences, that will generate less test cases applying more rules, and a big number of short sequences, that will generate simpler test cases. Note that the extension to testing of system modules is straightforward by starting from $c(v_1, \dots, v_n)$, with c any constructor for the sort being tested.

In our lists example, we may be interested in testing the function `ins-sort` using the global branch coverage criterion. This function is defined by two equations (`is1` and `is2`); one of these equations uses the function `ins-list`, and thus its three equations (`i11`, `i12`, and `i13`) must also be added to the needed coverage; finally, this function uses the functions `_<_` and `_<=_`, imported from

```

ins-sort(L)  $\rightsquigarrow^{is2}$ 
ins-list(ins-sort(L1), N1)  $\rightsquigarrow^{is2}$ 
ins-list(ins-list(ins-sort(L2), N2), N1)  $\rightsquigarrow^{is2}$ 
ins-list(ins-list(ins-list(ins-sort(L3), N3), N2), N1)  $\rightsquigarrow^{is2}$ 
ins-list(ins-list(ins-list(ins-list(ins-sort(L4), N4), N3), N2), N1)  $\rightsquigarrow^{is1}$ 
ins-list(ins-list(ins-list(ins-list(nil, N4), N3), N2), N1)  $\rightsquigarrow^{i11}$ 
ins-list(ins-list(ins-list(N4 . nil, N3), N2), N1)  $\rightsquigarrow^{i13}$ 
ins-list(ins-list(0 . ins-list(nil, s(N5)), N2), N1)  $\rightsquigarrow^{i11}$ 
ins-list(ins-list(0 . s(N5) . nil, N2), N1)  $\rightsquigarrow^{i12, o11}$ 
ins-list(0 . 0 . s(N5) . nil, N1)  $\rightsquigarrow^{i12, o12, o11}$ 
0 . 0 . 0 . s(N5) . nil

```

Fig. 1. Narrowing path for global branch coverage

NAT and a variable of sort `SortedList`, which is defined with two membership axioms (`o11` and `o12`). All these statements must be executed at least once by the test cases to fulfill global branch coverage. We can use our tool to automatically generate the test cases, following the default strategy that selects the smaller set of test cases:

```
Maude> (test ins-sort .)
```

```
1. ins-sort(1 . 0 . 0 . 0 . nil) has been reduced to 0 . 0 . 0 . 1 . nil
All the statements were covered.
```

Note that the tool shows the initial term, the result of reducing it in the module, and whether some reachable statements could not be used. The term shown by the tool may be obtained as shown in Figure 1, where `s` stands for the successor function over natural number (note that this is one branch of a search tree of depth 10).

Since all the possible instantiations of this term generate test cases traversing all the required statements, the tool generates the simplest one by replacing the variables with constants of the given sort (or the simplest built term if the sort does not have constants). If we find that any reduction is wrong, we could debug it with:

```
Maude> (invoke debugger with user test case 1 .)
Declarative debugging of wrong answers started.
```

This command starts the declarative debugging process [28] that, by asking questions to the user about the computations that took place will find the specific statement that generated the wrong behavior. This command is available for all the testing options.

Note that the extension to testing of system modules is straightforward; in this case we want to test the transitions of the terms with a given sort instead of a specific function, and thus the narrowing process starts with a term with variables of the given sort, and tries to apply all the reachable rules, equations, and memberships, proceeding in the same way as the testing for functional modules. That is, we start the narrowing process from $c(v_1, \dots, v_n)$, with c any constructor

for the sort and v_i variables of the appropriate sorts, and continue as indicated above.

Moreover, we propose another coverage criterion, related to modified condition decision coverage (MCDC) [19,10]. Basically, MCDC requires that all the conditions in a program are evaluated with the given set of test cases to both true and false. In a non-deterministic framework as the one of system modules it is important to know, as explained in [28], the applied statements that make the program reach certain states, the *positive information*, but also the statements that were not applied and thus prevented the program from reaching some other values, the *negative information*. While we obtain the positive information with the global branch coverage shown above, it does not provide any of the negative information. For this reason, we have implemented the so-called system coverage criterion, which requires a set of test cases to apply all the rules in the transformed module (which corresponds to global branch coverage) but also to fail for at least one condition for each rule in the *original* module.

6.2 Checking Invariants

Checking of invariants has already been studied for Maude specifications in [8, Chapter 12]. It takes advantage of the command `search`, that performs a breadth-first search from an initial term, given a bound in the number of steps and a condition to be fulfilled; by searching for the *negation* of the invariant we can check that no illegal states are reached. We apply a similar idea in our testing framework by using symbolic search; this search will traverse all the possible states and, each time a rule is applied, it tries to find a path to fulfill the negation of the invariant. If such a path is found, then the specification does not fulfill the invariant. Note that the invariant is usually specified by using equations, and thus it is important to use equations in the narrowing process, since it allows the tool to fix the values of the initial state required to fulfill the condition.

More formally, we consider a new rule $inv(pat) \Rightarrow pat$ if $Cond$, where inv is a new operator defined over the sort of states, pat is the pattern given for the invariant, and $Cond$ a condition (we assume the invariant is composed of a pattern and a condition, see below for details). Thus, for every narrowing sequence $t \rightsquigarrow^{q_1} t_1 \rightsquigarrow^{q_2} \dots \rightsquigarrow^{q_n} t_n$, the invariant is fulfilled if, for every t_i obtained by using a narrowing step with the rule q_i we cannot find a term t' such that $t_i \rightsquigarrow^{inv} t'$ (we look for the negation of the invariant). If such a term exists, then the term $\theta_{inv}(\theta_{q_1, \dots, q_n}(t))$ can be used as initial term for debugging; otherwise, the invariant holds.

The transformation presented in Section 4 allows us to check invariants in both functional and system modules. We could e.g. set an invariant on our critical section example stating some correct property over lists or systems, but it is worth examining how an initial term proving the specification wrong is obtained. In our critical section example we can specify a function `empty?`, that checks whether a `NatSoup` is empty, defined as follows:

```

op empty? : NatSoup -> Bool .
eq [mt1] : empty?(mtSoup) = true .
eq [mt2] : empty?(NS) = false [owise] .

```

and then search for a system that never has its first argument `empty` (remember that we look for the negation of the invariant) with:

```

Maude> (test [10] System =>+ NS1 [NL] [NWE] NS2 s.t. empty?(NS1) .)
The term mtSoup [nil] [0] 0 reaches the state
  mtSoup [nil] [empty] (0,0), which does not fulfill the invariant.

```

This command looks for terms of sort `System` that, in at least one step (indicated by the search arrow `=>+`; the tool also provides searches in zero or more steps with `=>*` and searches for final forms with `=>!`) and at most 10, match the pattern and fulfill the condition (the negation of the invariant). In this case, the tool has found (as expected) an initial state that, after applying one rule (in this specific case it is `cs-out` although it would be possible to apply other rules), reaches a state that does not fulfill the invariant. In this case the narrowing process has fixed the value of the first `NatSoup` to `mtSoup` to fulfill the condition and has forced an element to be in the critical section to apply the rule, while the `nil` list and the singleton soup are just possible instances of the variables left by the narrowing step $NS1 [NL] [NWE] NS2 \rightsquigarrow^{cs-out} NS1 [NL] [empty] (N1, NS2)$ and then checking the property by instantiating `NS1` with `mtSoup` when applying the equations for `empty?`.

6.3 Conformance Testing

Conformance testing [32][14][5] involves testing a system with respect to its specification. The goal of this approach is to check that the system presents the same behavior as the specification, that has already been tested. To check whether an implementation conforms to a specification we must formalize the conformance notion by means of an *implementation relation* that relates the two systems. In our case, and taking into account that a rewrite system can be understood as a labeled transition system, where terms stand for states and rewrites for transitions, we apply to Maude specifications the conformance testing strategies for such systems [32]. In particular, we use the relation `conf` [4], that requires the implementation to perform the same actions as the specification, although it allows the implementation to execute some other actions not included in the specification, that is, `conf` requires that an implementation does what it should do, not that it does not do what it is not allowed to do.

In our framework we consider that only the rules in the original specification must be executed in the implementation, and thus narrowing steps using equations are considered auxiliary and it is not required to reproduce them in the implementation. In this way, we compute all the possible paths by using narrowing in the specification and then that all these paths are also possible in the implementation. More formally, if we denote by $\sigma|_R$ the restriction of σ to the rules in R , that is, remove from σ all those statements that are not in the set, then we require that for every narrowing sequence $t \rightsquigarrow^{\sigma_s} t_s$ in the specification

there exists a narrowing sequence $t \rightsquigarrow^{\sigma_i} t_i$ in the implementation such that $\sigma_s \upharpoonright_R$ is a prefix of $\sigma_i \upharpoonright_R$. Note that, although the reached states may be different in the specification and the implementation (only the applied rules matter), we consider that both the correct specification and the system being tested share the same signature for the initial terms and the same rule labels; this can be achieved by means of a renaming.

For the sake of example, we could create a new module RED-CS that has the same rules as CS except for the rule cs-out. We can state CS as the specification with:

```
Maude> (correct test module CS .)
CS selected as correct module for testing.
```

Now, we can check the behavior of RED-CS with respect to this module with:

```
Maude> (test in RED-CS : System .)
Starting from the term 0 [nil] [0] 0 the rule
cs-out could not be applied to the implementation.
```

That is, the tool shows the simplest term (in fact, only the 0 in the critical section is instantiated during the process) that is required to find the disconformity between the specification and the implementation due to the cs-out rule.

7 Trusting

Our tool provides some trusting techniques to enhance its performance. Basically, it only takes into account labeled statements when computing coverages and checking the implementation relation. Moreover, the user can also select a subset of these statements by using the different commands available in the tool (trusting of all the statements of a given module, trusting of a complete kind of statements—e.g. all the equations, memberships, or rules—and trusting of single statements). Using these commands we can use different trusting strategies: assuming that our specifications are structured, we can test first easier specifications, and then trust them when testing larger specifications including them; and we can trust all the equations (except for the ones defining the property when checking invariants) and memberships when testing system modules. Of course, *trusting mechanisms are correct assuming the user points out as trusted only rules that are not relevant for the testing process.*

Trusting works in a different way depending on the testing strategy: if we are computing a coverage then the trusted statements are removed from the needed coverage, and thus we may reduce both the number and the complexity of the test cases. When using conformance testing, the trusted statements are related to the specification and indicate that the behavior specified by the rule is not required to be performed in the system being tested (e.g. because it is an auxiliary rule). That is, the sequences of statements σ required for coverage are not required to contain the trusted statements,⁵ while the restriction to rules

⁵ Note that using trusting when using system coverage will remove the statements from both the positive and negative information.


```

ins-sort(L)  $\rightsquigarrow$ is2
ins-list(ins-sort(L1), N1)  $\rightsquigarrow$ is2
ins-list(ins-list(ins-sort(L2), N2), N1)  $\rightsquigarrow$ is2
ins-list(ins-list(ins-list(ins-sort(L3), N3), N2), N1)  $\rightsquigarrow$ is1
ins-list(ins-list(ins-list(nil, N3), N2), N1)  $\rightsquigarrow$ is1
ins-list(ins-list(N3 . nil, N2), N1)  $\rightsquigarrow$ is2
ins-list(0 . N3 . nil, N1)  $\rightsquigarrow$ is2, is1
0 . 0 . N3 . nil

```

Fig. 2. Narrowing path for global branch coverage with trusting

in the specification given in conformance testing is now applied to non-trusted rules in the specification.

For example, we can trust the statements `is3` and `o12` (which required the longest computations in Figure 1) if we are sure of its correctness to improve the performance of the computation of the global branch coverage in Section 6.1 by using the commands:

```

Maude> (test include SORTED-NAT-LIST .)
Labels hd is1 is2 is3 is4 is5 is6 is7 is8 is9 is10 is11 is12 is13
is14 is15 is16 is17 is18 is19 is20 is21 is22 is23 is24 is25 is26
is27 is28 is29 is30 is31 is32 is33 is34 is35 is36 is37 is38 is39
is40 is41 is42 is43 is44 is45 is46 is47 is48 is49 is50 is51 is52
is53 is54 is55 is56 is57 is58 is59 is60 is61 is62 is63 is64 is65
is66 is67 is68 is69 is70 is71 is72 is73 is74 is75 is76 is77
is78 is79 is80 is81 is82 is83 is84 is85 is86 is87 is88 is89
is90 is91 is92 is93 is94 is95 is96 is97 is98 is99 is100
Labels hd is1 is2 is3 is4 is5 is6 is7 is8 is9 is10 is11 is12 is13
is14 is15 is16 is17 is18 is19 is20 is21 is22 is23 is24 is25 is26
is27 is28 is29 is30 is31 is32 is33 is34 is35 is36 is37 is38 is39
is40 is41 is42 is43 is44 is45 is46 is47 is48 is49 is50 is51 is52
is53 is54 is55 is56 is57 is58 is59 is60 is61 is62 is63 is64 is65
is66 is67 is68 is69 is70 is71 is72 is73 is74 is75 is76 is77
is78 is79 is80 is81 is82 is83 is84 is85 is86 is87 is88 is89
is90 is91 is92 is93 is94 is95 is96 is97 is98 is99 is100
Maude> (test deselect is3 o12 .)
Labels is3 o12 have been excluded from the coverage.
Maude> (test in SORTED-NAT-LIST : ins-sort .)
1. ins-sort(0 . 0 . 0 . nil) has been reduced to 0 . 0 . 0 . nil
All the statements were covered.

```

Obtaining in this case a simpler test case that covers all the statements. It is interesting to see that trusting a rule when using conformance provides more flexibility, because it allows to perform some analyses by removing auxiliary rules that are not supposed to be applied in the final implementation. However, if the user trusts a statement that should not be trusted he may obtain an incorrect answer, hence the assumption presented above about the correctness of trusting, that may produce incorrect results. Similarly, we can trust the rule `cs-out` when using conformance testing and check that in this case the specification and the implementation perform the same actions:

```

Maude> (test deselect cs-out .)
Labels cs-out have been excluded from the coverage.
Maude> (test in RED-CS : System .)
The implementation conforms to the specification.

```

The improvement in the performance when using trusting is highly dependent on the selected set of statements: while in some cases trusting may reduce the number of steps more than a 50%, in other cases they are not reduced at all. For example, the global branch coverage obtained in Section 6.1 was highly reduced by trusting the statements shown above, reducing the depth of the search tree from 10 to 7, as illustrated in Figure 2. However, selecting other statements

⁶ Remember that this is one branch of the search tree, that is, trusting has reduced the depth of the search tree from 10 to 7, which results in a huge improvement of the performance.

such as `is1` or `is2`, that must be always executed in order to reach a state where other statements can be used, would not reduce the size of the search at all. All the examples in this paper, and much more information is available at <http://maude.sip.ucm.es/testing/>.

8 Concluding Remarks and Ongoing Work

We have presented in this paper how to use narrowing to generate test cases for Maude specifications. To achieve this we use a module transformation that allows us to use the equational part of Maude modules in the narrowing process and a method to check whether the conditions of the applied statements are fulfilled, including those conditions that require membership axioms. Using these techniques we have implemented a tool that is able to compute a set of test cases fulfilling two different coverage criteria, to check whether an invariant is fulfilled by the specification, and to examine whether an implementation of the system fulfills the behavior indicated by its specification. Moreover, two different sets of test cases can be computed: a smaller set that contains more complex terms or a larger set that contains less complex terms; the user is in charge of selecting the most appropriate depending on the complexity of the specification and his knowledge of it. Trusting mechanisms are also provided to improve the performance of both coverage criteria and conformance testing. Finally, some predefined modules can be also used to generate the test cases. We are currently working on a comparison between our current approach using narrowing and (i) the previous one using ground terms, and (ii) similar approaches in other languages, either using narrowing, like Lazy Smallcheck, or random testing, like QuickCheck.

As future work, we plan to extend the tool by introducing symbolic model checking [11], that would allow the user to check linear temporal logic formulas over the specification starting from a term with variables, thus proving the formula on, potentially, all the possible inputs of the system. Moreover, we are studying new coverage criteria and implementation relations to allow the user to choose the most appropriate technique for each application. Finally, we also intend to develop a distributed implementation of the tool to deal with narrowing; in this way, we can start the symbolic search of the system in one Maude instance and then send the different paths to different Maude processes, that must share some information (the coverage and the reached states) to finish the search as soon as possible.

Acknowledgements. I thank Santiago Escobar for his kind help with narrowing.

References

1. Beizer, B.: Software testing techniques. Dreamtech (2002)
2. Borba, P., Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.): PSSE 2007. LNCS, vol. 6153. Springer, Heidelberg (2010)

3. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. *Theoretical Computer Science* 236, 35–132 (2000)
4. Brinksma, E., Scollo, G., Steenbergen, C.: LOTOS specifications, their implementations and their tests. In: *Protocol Specification, Testing, and Verification VI*, pp. 349–360 (1987)
5. Cartaxo, E.G., Neto, F.G.O., Machado, P.D.L.: Test case generation by means of UML sequence diagrams and labeled transition systems. In: *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, SMC 2007*, pp. 1292–1297. IEEE (2007)
6. 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)
7. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 268–279 (2000)
8. 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)
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: *Maude Manual (Version 2.6)* (January 2011), <http://maude.cs.uiuc.edu/maude2-manual>
10. Dupuy, A., Leveson, N.: An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In: *Proceedings of the 19th Digital Avionics Systems Conference, DASC 2000*, vol. 1, pp. 1B6.1–1B6.7 (2000)
11. 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)
12. Fay, M.: First-order unification in an equational theory. In: Joyner, W.H. (ed.) *Proceedings of the 4th Workshop on Automated Deduction*, pp. 161–167. Academic Press (1979)
13. Fischer, S., Kuchen, H.: Systematic generation of glass-box test cases for functional logic programs. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP 2007*, pp. 63–74. ACM Press (2007)
14. Gaudel, M.-C.: Software Testing Based on Formal Specification. In: Borba, P., Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) *PSSE 2007*. LNCS, vol. 6153, pp. 215–242. Springer, Heidelberg (2010)
15. Gomez-Zamalloa, M., Albert, E., Puebla, G.: Test case generation for object-oriented imperative languages in CLP. *Theory and Practice of Logic Programming* 10, 659–674 (2010)
16. Hussmann, H.: Unification in Conditional Equational Theories. In: Caviness, B.F. (ed.) *EUROCAL 1985*. LNCS, vol. 204, pp. 543–553. Springer, Heidelberg (1985)
17. Hierons, R.M., Bogdanov, K., Bowen, J.P., Rance Cleaveland, J.D., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., Zedan, H.: Using formal specifications to support testing. *ACM Computing Surveys* 41(2), 1–76 (2009)
18. Jacquemard, F., Rusinowitch, M., Vigneron, L.: Compiling and Verifying Security Protocols. In: Parigot, M., Voronkov, A. (eds.) *LPAR 2000*. LNCS (LNAI), vol. 1955, pp. 131–160. Springer, Heidelberg (2000)

19. Jasper, R., Brennan, M., Williamson, K., Currier, B., Zimmerman, D.: Test data generation and feasible path analysis. In: Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 1994, pp. 95–107. ACM (1994)
20. King, J.C.: Symbolic execution and program testing. *Communications of the ACM* 19, 385–394 (1976)
21. Meadows, C.: Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security* 1 (1992)
22. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science* 96(1), 73–155 (1992)
23. Middeldorp, A., Hamoen, E.: Counterexamples to Completeness Results for Basic Narrowing. In: Kirchner, H., Levi, G. (eds.) ALP 1992. LNCS, vol. 632, pp. 244–258. Springer, Heidelberg (1992)
24. Müller, R.A., Lembeck, C., Kuchen, H.: A symbolic Java virtual machine for test case generation. In: IASTED Conf. on Software Engineering, pp. 365–371 (2004)
25. Ntafos, S.C.: A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering* 14, 868–874 (1988)
26. Pacheco, C.: Directed Random Testing. PhD thesis, Massachusetts Institute of Technology (June 2009)
27. Riesco, A.: Test-Case Generation for Maude Functional Modules. In: Mossakowski, T., Kreowski, H.-J. (eds.) WADT 2010. LNCS, vol. 7137, pp. 287–301. Springer, Heidelberg (2012)
28. Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R.: Declarative debugging of rewriting logic specifications. *Journal of Logic and Algebraic Programming* (2011) (to appear)
29. Robinson, J.A.: A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12(1), 23–41 (1965)
30. Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In: Gill, A. (ed.) Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, September 25, pp. 37–48. ACM (2008)
31. Slagle, J.R.: Automated theorem-proving for theories with simplifiers, commutativity and associativity. *Journal of the ACM* 21(4), 622–642 (1974)
32. Tretmans, J.: Model Based Testing with Labelled Transition Systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008)

A Rule-Based Framework for Building Superposition-Based Decision Procedures

Elena Tushkanova^{1,2}, Alain Giorgetti^{1,2},
Christophe Ringeissen¹, and Olga Kouchnarenko^{1,2}

¹ Inria, Villers-les-Nancy, F-54600, France

² CNRS FEMTO-ST and University of Franche-Comté, Besançon, F-25030, France

Abstract. This paper deals with decision procedures specified as inference systems. Among them we focus on superposition-based decision procedures. The superposition calculus is a refutation-complete inference system at the core of all equational theorem provers. In general this calculus provides a semi-decision procedure that halts on unsatisfiable inputs but may diverge on satisfiable ones. Fortunately, it may also terminate for some theories of interest in verification, and thus it becomes a decision procedure. To reason on the superposition calculus, a schematic superposition calculus has been studied, for instance to automatically prove termination. This paper presents an implementation in Maude of these two inference systems. Thanks to this implementation we automatically derive termination of superposition for a couple of theories of interest in verification.

1 Introduction

Satisfiability procedures modulo background theories such as classical data structures (e.g., lists, records, arrays, ...) are at the core of many state-of-the-art verification tools. Designing and implementing satisfiability procedures is a very complex task, where one of the main difficulties consists in proving their soundness.

To overcome this problem, the rewriting approach [2] allows us to build satisfiability procedures in a flexible way, by using a superposition calculus [14] (also called *Paramodulation Calculus* in [10]). In general, a fair and exhaustive application of the rules of this calculus leads to a semi-decision procedure that halts on unsatisfiable inputs (the empty clause is generated) but may diverge on satisfiable ones. Therefore, the superposition calculus provides a decision procedure for the theory of interest if one can show that it terminates on every input made of the (finitely many) axioms and any set of ground literals. The needed termination proof can be done by hand, by analysing the (finitely many) forms of clauses generated by saturation, but the process is tedious and error-prone. To simplify this process, a schematic superposition calculus has been developed [10] to build the schematic form of the saturations. This schematic superposition calculus is very useful to analyse the behavior of the superposition calculus on a

given input theory, as shown in [9] to prove automatically the termination and the combinability of the related decision procedure.

This paper explains how to prototype the schematic superposition calculus to provide a toolkit for further experiments. The main idea is to implement the calculus so that the user can easily modify the code corresponding to an executable specification. Implementing this schematic calculus in an off-the-shelf equational theorem prover like the E prover [15] or SPASS [16] would be a difficult and less interesting task, since the developer and the user would have to understand a complex piece of code which is the result of years of engineering and debugging. To make the task easier another quite natural solution would be to use a logical framework since this calculus is defined by an inference system. This is why we propose to prototype the schematic superposition calculus by using a rule-based logical framework. Our goal is to get a rule-based program which is as close as possible to the formal specification. To achieve this goal, we propose to use Maude because Maude includes support for unification and narrowing, which are key operations of the calculus of interest, and the Maude meta-level provides a flexible way to control the application of rules and powerful search mechanisms.

Our implementation of schematic superposition is very useful to get an automatic validation of saturations described in previous papers. Hence, our experiments allow us to find a flaw in an example of [9].

The paper is structured as follows. After introducing preliminary notions and presenting superposition calculi in Section 2, Section 3 explains how we implement these calculi using the Maude system. Then Section 4 reports our experiments with our implementation to prove the termination of superposition for theories corresponding to classical data structures such as lists and records. Section 5 concludes and presents future work.

2 Background

2.1 First-Order Logic

We assume the usual first-order syntactic notions of signature, term, position, and substitution, as defined, e.g., in [5]. We use the following notations: l, r, u, t are terms, v, w, x, y, z are variables, all other lower case letters are constant or function symbols. Given a function symbol f , a *f-rooted* term is a term whose top-symbol is f . A *compound* term is a f -rooted term for a function symbol f of arity different from 0. Given a term t and a position p , $t|_p$ denotes the subterm of t at position p , and $t[l]_p$ denotes the term t in which l appears as the subterm at position p . When the position p is clear from the context, we may simply write $t[l]$. The depth of a term is defined inductively as follows: $depth(t) = 0$, if t is a constant or a variable, and $depth(f(t_1, \dots, t_n)) = 1 + \max\{depth(t_i) \mid 1 \leq i \leq n\}$. A term is *flat* if its depth is 0 or 1. Application of a substitution σ to a term t (resp. a formula ψ) is written $\sigma(t)$ (resp. $\sigma(\psi)$).

A *literal* is either an equality $l = r$ or a disequality $l \neq r$. A *positive literal* is an equality and a *negative literal* is a disequality. We use the symbol \bowtie to denote

either = or \neq . The depth of a literal $l \bowtie r$ is defined as follows: $depth(l \bowtie r) = depth(l) + depth(r)$. A positive literal is *flat* if its depth is 0 or 1. A negative literal is *flat* if its depth is 0.

A first-order *formula* is built in the usual way over the universal and existential quantifiers, Boolean connectives, and symbols in a given first-order signature. We call a formula *ground* if it has no variables. A *clause* is a disjunction of literals. A *unit clause* is a clause with only one disjunct, equivalently a literal. The *empty clause*, denoted \perp , is the clause with no disjunct, corresponding to an unsatisfiable formula.

We also consider the usual first-order notions of model, satisfiability, validity, logical consequence. A *first-order theory* (over a finite signature) is a set of first-order formulae with no free variables. When T is a finitely axiomatized theory, $Ax(T)$ denotes the set of axioms of T . We consider first-order theories *with equality*, for which the equality symbol = is always interpreted as the equality relation. A formula is *satisfiable in a theory T* if it is satisfiable in a model of T . The *satisfiability problem* modulo a theory T amounts to establishing whether any given finite conjunction of literals (or equivalently, any given finite set of literals) is T -satisfiable or not. In this paper, we study decision procedures for the satisfiability problem modulo T , where $Ax(T)$ is a finite set of literals.

We consider inference systems using well-founded orderings on terms/literals that are total on ground terms/literals. An ordering $<$ on terms is a *simplification ordering* [5] if it is stable ($l < r$ implies $l\sigma < r\sigma$ for every substitution σ), monotonic ($l < r$ implies $t[l]_p < t[r]_p$ for every term t and position p), and has the subterm property (i.e., it contains the subterm ordering: if l is a strict subterm of r , then $l < r$). Simplification orderings are well-founded. A term t is *maximal* in a multiset S of terms if there is no $u \in S$ such that $t < u$, equivalently $t \not< u$ for every $u \in S$. Hence, if $t \not< u$, then t and u are different terms and t is maximal in $\{t, u\}$. An ordering on terms is extended to literals by using its multiset extension on literals viewed as multisets of terms. Any positive literal $l = r$ (resp. negative literal $l \neq r$) is viewed as the multiset $\{l, r\}$ (resp. $\{l, l, r, r\}$). Also, a term is *maximal* in a literal whenever it is maximal in the corresponding multiset.

2.2 Paramodulation Calculus

In this paper we consider unit clauses, i.e. clauses composed of at most one literal. We present the restriction UPC (for *Unit Paramodulation Calculus*) of the inference system \mathcal{PC} .

Our presentation of this calculus takes the best (to our sense) from the presentations in [2], [10] and [9]. The inference system UPC consists of the rules in Figs. 1 and 2. Expansion rules (Fig. 1) aim at generating new (deduced) clauses. For brevity left and right paramodulation rules are grouped into a single rule, called *Superposition*, that uses an equality to perform a replacement of equal by equal into a literal. *Reflection* rule generates the empty clause when the two sides of a disequality are unifiable. Contraction rules (Fig. 2) aim at simplifying the set of literals. Using *Subsumption*, a literal is removed when it is an instance

of another one. *Simplification* rewrites a literal into a simpler one by using an equality that can be considered as a rewrite rule. Trivial equalities are removed by *Deletion*. A fundamental feature of \mathcal{PC} and \mathcal{UPC} is the usage of a simplification ordering $<$ to control the application of *Superposition* and *Simplification* rules by orienting equalities. Hence, the *Superposition* rule is applied by using terms that are maximal in their literals with respect to $<$. This ordering is total on ground terms. We use a lexicographic path ordering [5] such that terms of positive depth are greater than constants.

Let us recall the usual definitions of redundancy, saturation, derivation and fairness. A clause C is *redundant* with respect to a set S of clauses if either $C \in S$ or S can be obtained from $S \cup \{C\}$ by a sequence of applications of contraction rules (cf. Fig. 2). An inference is *redundant* with respect to a set S of clauses if its conclusion is redundant with respect to S . A set S of clauses is *saturated* if every inference with a premise in S is redundant with respect to S . A *derivation* is a sequence $S_0, S_1, \dots, S_i, \dots$ of sets of clauses where each S_{i+1} is obtained from S_i by applying an inference to add a clause (by expansion rules in Fig. 1) or to delete a clause (by contraction rules in Fig. 2). For the *Simplification* rule, one can remark that its application corresponds to two steps in the derivation: the first step adds a new literal, whilst the second one deletes a literal. A derivation is characterized by its *limit*, defined as the set of persistent clauses $\bigcup_{j \geq 0} \bigcap_{i > j} S_i$, that is, the union for each $j \geq 0$ of the set of clauses occurring in all future steps starting from S_j . A derivation $S_0, S_1, \dots, S_i, \dots$ is *fair* if for every inference with premises in the limit, there is some $j \geq 0$ such that the inference is redundant with respect to S_j . The set of persistent literals obtained by a fair derivation is called the *saturation* of the derivation.

$$\text{Superposition} \quad \frac{l[u'] \bowtie r \quad u = t}{\sigma(l[t] \bowtie r)}$$

if i) $\sigma(u) \not\leq \sigma(t)$, ii) $\sigma(l[u']) \not\leq \sigma(r)$, and iii) u' is not a variable.

$$\text{Reflection} \quad \frac{u' \neq u}{\perp}$$

Above, u and u' are unifiable and σ is the most general unifier of u and u' .

Fig. 1. Expansion inference rules of \mathcal{UPC}

2.3 Schematic Paramodulation Calculus

The *Schematic Unit Paramodulation Calculus SUPC* is an abstraction of \mathcal{UPC} . Indeed, any concrete saturation computed by \mathcal{UPC} can be viewed as an instance of an abstract saturation computed by \mathcal{SUPC} , as shown by Theorem 2 in [9].

$$\begin{array}{l}
 \text{Subsumption} \quad \frac{S \cup \{L, L'\}}{S \cup \{L\}} \text{ if } L' = \sigma(L). \\
 \\
 \text{Simplification} \quad \frac{S \cup \{C[l'], l = r\}}{S \cup \{C[\sigma(r)], l = r\}} \\
 \text{if i) } l' = \sigma(l), \text{ ii) } \sigma(l) > \sigma(r), \text{ and iii) } C[l'] > (\sigma(l) = \sigma(r)). \\
 \\
 \text{Deletion} \quad \frac{S \cup \{u = u\}}{S}
 \end{array}$$

Fig. 2. Contraction inference rules of *UPC*

Hence, if *SUPC* halts on one given abstract input, then *UPC* halts for all the corresponding concrete inputs. More generally, *SUPC* is an automated tool to check properties of *UPC* such as termination, stable infiniteness and deduction completeness [9]. This paper focuses on termination.

SUPC is almost identical to *UPC*, except that literals are constrained by conjunctions of atomic constraints of the form *const(x)* where *x* is a variable. For sake of brevity, *const(x₁, . . . , x_n)* denotes the conjunction *const(x₁)* ∧ . . . ∧ *const(x_n)*. *SUPC* consists of the rules in Figs. 3 and 4.

With respect to [9], we have slightly adapted the subsumption rule so that the instantiation is not only a renaming but also a substitution instantiating constrained variables by constrained variables. This allows us to have a more compact form of saturations even for simple cases, as shown in Sect. 4. For a given theory *T* with signature Σ , *SUPC* is executed with the input $Ax(T) \cup G_0^T$ where G_0^T is defined by

$$\begin{aligned}
 G_0^T = \{ & \perp, x = y \parallel \text{const}(x, y), x \neq y \parallel \text{const}(x, y) \} \\
 & \cup \bigcup_{f \in \Sigma} \{ f(x_1, \dots, x_n) = x_0 \parallel \text{const}(x_0, x_1, \dots, x_n) \}
 \end{aligned}$$

and schematizes any set of ground flat equalities and disequalities built over Σ , along with the empty clause.

2.4 Maude Language

Maude [4] is a rule-based language well-suited to implement the above inference systems. Maude’s basic programming statements are equations and rules. Its semantics is based on rewriting logic where terms are reduced by applying rewrite rules. Maude has many important features such as reflection, pattern-matching, unification and narrowing. Reflection is a very desirable property of a computational system, because a reflective system can access its own meta-level and this way can be much more powerful, flexible and adaptable than a nonreflective one. Maude’s language design and implementation make systematic use of the fact that rewriting logic is reflective. Narrowing [3] is a generalization of term rewriting that allows free variables in terms (as in logic programming) and replaces

$$\text{Superposition} \quad \frac{l[u'] \bowtie r \parallel \varphi \quad u = t \parallel \psi}{\sigma(l[t] \bowtie r \parallel \varphi \wedge \psi)}$$

if i) $\sigma(u) \not\leq \sigma(t)$, ii) $\sigma(l[u']) \not\leq \sigma(r)$, and iii) u' is not an unconstrained variable.

$$\text{Reflection} \quad \frac{u' \neq u \parallel \psi}{\perp} \quad \text{if } \sigma(\psi) \text{ is satisfiable.}$$

Above, u and u' are unifiable and σ is the most general unifier of u and u' .

Fig. 3. Constrained expansion inference rules of *SUPC*

$$\text{Subsumption} \quad \frac{S \cup \{L \parallel \psi, L' \parallel \psi'\}}{S \cup \{L \parallel \psi\}}$$

if either a) $L \in Ax(T)$, ψ is empty and for some substitution σ , $L' = \sigma(L)$; or b) $L' = \sigma(L)$ and $\psi' = \sigma(\psi)$, where σ is a renaming or a mapping from constrained variables to constrained variables.

$$\text{Simplification} \quad \frac{S \cup \{C[l'] \parallel \varphi, l = r\}}{S \cup \{C[\sigma(r)] \parallel \varphi, l = r\}}$$

if i) $l = r \in Ax(T)$, ii) $l' = \sigma(l)$, iii) $\sigma(l) > \sigma(r)$, and iv) $C[l'] > (C[\sigma(r)] = \sigma(r))$.

$$\text{Tautology} \quad \frac{S \cup \{u = u \parallel \varphi\}}{S}$$

$$\text{Deletion} \quad \frac{S \cup \{L \parallel \varphi\}}{S} \quad \text{if } \varphi \text{ is unsatisfiable.}$$

Fig. 4. Contraction inference rules of *SUPC*

pattern-matching by unification in order to (non-deterministically) instantiate and reduce a term. The narrowing feature is provided in an extension of Maude named Full Maude. It is clearly of great interest to implement the superposition rules of our calculi.

3 Implementation

This section describes the main ideas and principles of our implementation of *UPC* and *SUPC* in Maude.

3.1 Data Representation

Let us consider how we represent terms and literals. Maude symbols are reflected in Maude as elements of the sort `Qid` (quoted identifier). Maude terms are reflected as elements of the sorts `Constant`, `Variable` and `Term`. We exploit the Maude reflection feature by using the sort `Term` to define the new sort `Literal` for literals, as follows:

```
fmod LITERAL is
  pr META-TERM .

  sort Literal .
  op _equals_ : Term Term -> Literal [comm] .
  op _!=_      : Term Term -> Literal [comm] .
endfm
```

The attribute `[comm]` declares that the infix binary symbols `equals` and `!=` for equality and disequality are commutative. For sets of literals we define the sort `SetLit` by instantiating the polymorphic sort `Set{X}` defined in the parameterized module `SET{X} :: TRIV` of the prelude of Maude, as follows:

```
view Literal from TRIV to LITERAL is
  sort Elt to Literal .
endv
```

```
fmod SETLIT is
  pr LITERAL .
  pr SET{Literal} * (sort Set{Literal} to SetLit) .
endfm
```

The first three lines declare that the sort `Literal` can be viewed as the sort of elements provided by the theory `TRIV`. This Maude view is named `Literal`. It is used in the module `SETLIT` to instantiate `Set{X}` as `Set{Literal}`. Finally, the sort `SetLit` is a renaming of the sort `Set{Literal}`. Consequently, the sets in this sort can be built by using the constant `empty`, and by using an associative, commutative, and idempotent union operator, written `_.`. A singleton set is identified with its element (`Literal` is a subsort of `Set{Literal}`).

A schematic literal is the empty clause, an axiom, or a constrained literal. The sort `AConstr` of atomic constraints is defined by the operator

```
op const : Term -> AConstr .
```

and the sort `Constr` of constraints is a renaming of the sort `Set{AConstr}` of sets of atomic constraints. Then, the sort `SLiteral` of schematic literals is declared by

```
fmod SLITERAL is
  sort SLiteral .
  op emptyClause : -> SLiteral .
  op ax : Literal -> SLiteral .
```

```

op _ || _ : Literal Constr -> SLiteral .
endfm

```

where the infix operator `||` constructs a constrained literal from a literal and a constraint. Similarly, for sets of schematic literals a sort `SetSLit` is defined in a module `SETSLIT`.

3.2 Inference Rules

This section presents the encoding of *SUPC*, the encoding of *UPC* being similar. Let us emphasize two main ideas of this encoding: 1) inference rules are translated into rewrite rules, and 2) rule application is controlled thanks to specially designed states. More precisely, the encoding description starts with the translation of some contraction rules into rewrite rules (the simplification rule is omitted). Afterwards, it continues with the expansion rules, whose fair application strategy is encoded by using a notion of state together with rules to specify the transitions between states.

Contraction Rules. The following Maude conditional rewrite rule encodes the first case of *Subsumption* inference rule in *SUPC*:

```

cr1 [subsum1] : (ax(L1), (L2 || Phi2)) => ax(L1)
  if LiteralMatch(L1, L2) /= noMatch .

```

The function call `LiteralMatch(L1, L2)` checks if the second literal `L2` is matched by the first one (`L1`), by calling the Maude function `metaMatch`.

The following two Maude conditional rewrite rules encode the second case of *Subsumption* inference rule, decomposed into two cases:

```

cr1 [subsum2] : L1 || Phi1, L2 || Phi2 => L1 || Phi1
  if isRename(L1 || Phi1, L2 || Phi2) .

```

```

cr1 [subsum3] : L1 || Phi1, L2 || Phi2 => L1 || Phi1
  if filter(L1 || Phi1, L2 || Phi2) .

```

The function `isRename` checks if one constrained literal is the renaming of another one by checking the existence of a substitution mapping the first literal into the second one, and the constraint of the first literal into the constraint of the second one. Moreover, this substitution should replace variables by variables and the correspondence between the replaced variables and the replacing ones should be one to one. The function call `filter(L1 || Phi1, L2 || Phi2)` checks if the constrained literal `L1 || Phi1` is more general than the constrained literal `L2 || Phi2` by determining the existence of a substitution mapping the first literal into the second one, and the constraint of the first literal into the constraint of the second one.

The *Simplification* inference rule rewrites a literal into a simpler one by using an axiom as a rewrite rule. This is performed by the Maude function `metaFrewrite` that rewrites the metarepresentation of a term with the rules

defined in the metarepresentation of a module. In our implementation, a function `addRl` adds an axiom to the metarepresentation of a module `INITIAL-MODULE` where all the functional symbols are defined.

```
op addRl : Term Term -> Module .
eq addRl(L, R) = addRules(
  (rl (L) => (R) [none] .), upModule('INITIAL-MODULE, false)) .
```

This function uses the Full Maude function `addRules` that takes a set of rules and a module as parameters. The axiom `ax(L equals R)` is added by the function call `addRl(L,R)`.

The inference rule *Tautology* is simply encoded by the rewrite rule

```
rl [tautology] : U equals U || Phi => empty .
```

The inference rule *Deletion* is encoded by the conditional rewrite rule

```
cr1 [del] : L || Phi => empty if isSatisfiable(Phi) == false .
```

where the function `isSatisfiable` checks if a given constraint holds, i.e. none of the terms it constraints is compound.

Expansion Rules. The order of rule applications has to be controlled. In particular, contraction rules should be given a higher priority than expansion ones. An expected solution could be to control rule applications with the strategy language described in [11, 8], but unfortunately it appeared not to be compatible with the Full Maude version 2.5b required for narrowing (see details below).

To circumvent this technical problem we propose to control rules with states. We consider three distinct states, for the sets of literals derived by *SUPC*. These states and the sort of states are defined as follows:

```
mod STATE is
  pr SETSLIT .
  sort State .
  op state : SetSLit -> State .
  op _selectOneLitFromGenSet_ : SetSLit SetSLit -> State .
  op _redundancy_ : SetSLit SLiteral -> State .
endm
```

The input state of the expansion rules of *SUPC* is expected to be of the form `state(S)` where *S* is a set of schematic literals.

The *Reflection* rule checks whether a given set of schematic literals contains a constrained disequality whose two sides are unifiable by a substitution that also satisfies the constraint. In this case the empty clause is added to the set of literals. The *Reflection* rule is encoded by the following conditional rewrite rule:

```
cr1 [reflection] :
  state((S, U' != U || Phi)) =>
    state((S, U' != U || Phi, emptyClause))
    if isSatisf(U' != U || Phi) .
```

where the function `isSatisf` performs the above mentioned checking.

The *Superposition* rule

$$\frac{l[u'] \bowtie r \parallel \varphi \quad u = t \parallel \psi}{\sigma(l[t] \bowtie r \parallel \varphi \wedge \psi)}$$

produces a new literal of the form $\sigma(l[t] \bowtie r \parallel \varphi \wedge \psi)$ from any set containing two schematic literals (axioms or constrained literals) of the form $l[u'] \bowtie r \parallel \varphi$ and $u = t \parallel \psi$, if the side conditions given in Fig. 3 are satisfied with the most general unifier σ of u and u' . This notion of superposition is close to the notion of narrowing. The idea is to use the second literal $u = t$ as a rewriting rule $u \rightarrow t$, to narrow the left-hand side $l[u']$ of the first literal. If the narrowing succeeds it produces a term $\sigma(l[t])$ where σ is a most general unifier of u and u' . It remains to apply σ to the right-hand side r of the first literal and to the conjunction of the two constraints φ and ψ .

To narrow we use a function `metaENarrowShowAll` already implemented in Full Maude version 2.5. In this version the narrowing was restricted to non-variable positions, along its standard definition. But the *Superposition* rule of *SUPC* requires the unusual feature: narrowing should also be applied at the positions of the variables schematizing constants. Therefore we have asked Santiago Escobar, the developer of narrowing in Full Maude, to implement this feature. As an answer to this request, he has introduced a flag `alsoAtVarPosition` to the narrowing function for disabling the standard restriction.

A second difficulty is that the `metaENarrowShowAll` function called on the term $l(u')$ and the rule $u \rightarrow t$ generates all the possible narrowings at all the positions, whereas one application of the *Superposition* rule produces only one literal. To solve this problem two additional states `S selectOneLitFromGenSet S'` and `S redundancy L` have been introduced, where S is a given set of schematic literals, S' is the set of schematic literals produced by the narrowing function applied to two schematic literals from S , and L is one schematic literal. Then *Superposition* is encoded by four Maude rewriting rules named `sup`, `select`, `no-sup` and `pick`. The `sup` rule is defined by

```
rl [sup] : state((S, L1, L2)) =>
  (S, L1, L2) selectOneLitFromGenSet applySup(L1, L2) .
```

where the function `applySup` generates from `L1` and `L2` a set of new schematic literals by calling the narrowing function and checking the ordering conditions of the *Superposition* rule. The ordering conditions invoke a function implementing the orderings detailed in Section 3.4. When the set of new schematic literals is empty, the rule

```
rl [no-sup] : S selectOneLitFromGenSet empty => state(S) .
```

returns the input set in a state ready for another expansion. Otherwise, the rule

```
rl [select] : S selectOneLitFromGenSet (L, S') =>
  if checkConstr(L) then S redundancy L
  else S selectOneLitFromGenSet S' fi .
```

considers one by one the schematic literals in the new set until the set is empty or a schematic literal L with a satisfiable constraint is found. Satisfiability is checked by invoking the function `checkConstr`. If the constraint of L is satisfiable then a state S `redundancy L` is constructed. It is an input state for the rule

```

r1 [pick] : S redundancy L =>
  if L isRedundant S == false
    then state((S, L))
    else state(S)
  fi .

```

which checks if a generated schematic literal L is redundant with respect to a given set S of schematic literals. The redundancy is checked by the function `isRedundant` that uses the Maude function `metaSearch`. This function tries to reach the set S from the union (S, L) of S and $\{L\}$ by applying contraction rules. If the new schematic literal is not redundant then it is added to the state, otherwise, the state is unchanged.

3.3 Saturation

A forward search for generated sets of schematic literals is performed by a function `searchState` defined by

```

op searchState : State Nat -> State .
eq searchState(S', N) = downTerm(getTerm(metaSearch(
  upModule('SP, false), upTerm(S'), 'state['S:SetSLit],
  nil, '*', unbounded, N)), error1) .

```

where `SP` is a module where all the expansion rules are defined. The function call `searchState(S,N)` tries to reach the N th state from an initial state S by applying the expansion rules. It uses a breadth-first exploration of the reachable state space, which is a fundamental graph traversal strategy implemented by the Maude `metaSearch` function with the `'*` parameter. When the Maude function `downTerm` fails in moving down the meta-represented term given as its first argument, it returns its second argument, namely `error1`, which is declared as a constant of sort `State` (`op error1 : -> [State] .`).

Then the principle of saturation is implemented by the function `saturate` defined by

```

op saturate : State -> State .
eq saturate(St) =
  if searchState(St, 1) == error1 then St else
    if searchState(St, 1) /= St then saturate(searchState(St, 1))
    else St fi fi .

```

which implements a fixpoint algorithm in order to get the state of a saturated set of schematic literals. If the initial state is already saturated, then the function returns it unchanged.

A saturated set of schematic literals could alternatively be computed from an initial state by the Maude `metaSearch` function with a `'!` parameter (searching for a state that cannot be further rewritten), but the function `searchState` computing intermediary states is also interesting for debugging purposes. Note that the Maude `metaSearch` function is already used with the parameter `'!` to apply contraction rules.

3.4 Orderings

A fundamental feature of our superposition calculi is the usage of a simplification ordering which is total on ground terms. This section presents all the orderings used in the side conditions of the inference rules and describes their implementation. In our calculi, we assume that compound terms are greater than constants. To satisfy this assumption, it is sufficient to use an LPO ordering with a precedence on function symbols such that non-constant function symbols are greater than constants.

Definition 1. *Given a precedence $>_F$ on function symbols, the **lexicographic path ordering (LPO)** $>_{lpo}$ [5] is defined as follows:*

$$LPO1 \quad \frac{(s_1, \dots, s_n) >_{lpo}^{lex} (t_1, \dots, t_m) \quad f(s_1, \dots, s_n) >_{lpo} t_1, \dots, t_m}{f(s_1, \dots, s_n) >_{lpo} f(t_1, \dots, t_m)}$$

$$LPO2 \quad \frac{f >_F g \quad f(s_1, \dots, s_n) >_{lpo} t_1, \dots, t_m}{f(s_1, \dots, s_n) >_{lpo} g(t_1, \dots, t_m)}$$

$$LPO3 \quad \frac{u_k >_{lpo} t}{f(u_1, \dots, u_k, \dots, u_p) >_{lpo} t}$$

$$LPO4 \quad \frac{}{f(u_1, \dots, u_k, \dots, u_p) >_{lpo} u_k}$$

where f and g are two functional symbols, $n \geq 0$ and $m \geq 0$ are two non-negative integers, $p \geq 1$ is a positive integer, and $s_1, \dots, s_n, t_1, \dots, t_m, u_1, \dots, u_p, t$ are terms. We write $s >_{lpo} t_1, \dots, t_m$ when $s >_{lpo} t_k$ for any positive integer $k \in [1, m]$. The ordering $>_{lpo}^{lex}$ denotes the lexicographic extension of $>_{lpo}$. The lexicographic extension can be specified as an inference system that can be directly encoded in Maude.

The LPO ordering is implemented as a Boolean function `gtLPO()` such that `gtLPO(s, SC, t) = true` if and only if $s >_{lpo} t$. One can remark the additional parameter `SC`. It collects the constrained variables that are viewed as constants in the precedence ordering: constrained variables are smaller than non-constant function symbols.

Let us briefly present the four main rules implementing $\text{gtLPO}(s, \text{SC}, t)$.

1. When $n \geq 1$ and $m \geq 1$, the rule *LPO1* is encoded by

```
ceq gtLPO(F[NeSL], SC, F[NeTL]) = true
  if gtLexLPO(NeSL, SC, NeTL) == true
    and termGtList(F[NeSL], SC, NeTL) == true .
```

where NeSL and NeTL are non-empty lists of terms. Here the head symbols of s and t are equal. Then the list of subterms NeSL of $s = \text{F}[\text{NeSL}]$ should be greater than the list of subterms NeTL of t and the term s should be greater than all the elements in the list of subterms of t .

2. When $n \geq 1$ and $m \geq 1$, the rule *LPO2* is encoded by

```
ceq gtLPO(F[NeSL], SC, G[NeTL]) = true
  if (gtSymb(F, SC, G) == true) and
    termGtList(F[NeSL], SC, NeTL) == true .
```

Here the heads of s and t are not equal. Then the head of s should be greater than the head of t and s should be greater than all the direct subterms of t .

3. The rule *LPO3* is encoded by

```
ceq gtLPO(F[UL1, Uk, UL2], SC, t) = true
  if gtLPO(Uk, SC, t) == true .
```

whose condition checks whether a direct subterm of $s = \text{F}[\text{UL1}, \text{Uk}, \text{UL2}]$ is greater than t .

4. The rule

```
eq gtLPO(F[UL1, Uk, UL2], SC, Uk) = true .
```

encodes *LPO4*, when a direct subterm of $s = \text{F}[\text{UL1}, \text{Uk}, \text{UL2}]$ is equal to t .

The ordering $>_{lpo}$ on terms is extended to literals thanks to the multiset extension of $>_{lpo}$. An equality $l = r$ is represented as a multiset $\{l, r\}$ while a disequality $l \neq r$ is represented as a multiset $\{l, l, r, r\}$. As for the lexicographic extension, the multiset extension can be specified as an inference system that can be directly encoded in Maude.

4 Experimentations

We have done some experiments to compare the (schematic) saturations computed by our tool with corresponding results we can find in the literature. For the theory of lists without extensionality, our tool generates the same saturation as the one given in [10]. More surprisingly, for the theory of lists with extensionality, our implementation reveals that the description given in [9] for the saturation is incomplete. We also consider the case of records of length 3 for which superposition is known to terminate on ground literals [1].

4.1 Theories of Lists

We experiment with two theories of lists *à la Shostak*, either without or with extensionality.

Let $\Sigma_{List} = \{cons, car, cdr\}$ be the signature of the theory of lists. The set G_0^{List} consists of the empty clause \perp and the following schemas of ground flat literals over the signature Σ_{List} :

$$x = y \parallel const(x, y) \quad (1)$$

$$x \neq y \parallel const(x, y) \quad (2)$$

$$car(x) = y \parallel const(x, y) \quad (3)$$

$$cdr(x) = y \parallel const(x, y) \quad (4)$$

$$cons(x, y) = z \parallel const(x, y, z) \quad (5)$$

where x , y and z are constrained variables.

Theory of Lists without Extensionality. The theory of lists without extensionality is axiomatized by the following two axioms:

$$car(cons(X, Y)) = X \quad (6)$$

$$cdr(cons(X, Y)) = Y \quad (7)$$

where X and Y are universally quantified variables.

Lemma 1. *The set $G_0^{List} \cup \{(6), (7)\}$ is saturated by SUPC.*

This result is given in [10]. The interested reader can find our proof in Appendix A.

From an encoding of $G_0^{List} \cup \{(6), (7)\}$ our tool generates no new schematic literal. Notice that on this example the abstraction by schematization is exact, in the following sense: the saturated set computed by SUPC is the schematization of any saturated set computed by UPC.

Theory of Lists with Extensionality. This theory is axiomatized by the two axioms (6) and (7), plus the axiom (called the extensionality axiom)

$$cons(car(X), cdr(X)) = X \quad (8)$$

where X is a universally quantified variable.

Lemma 2. *The saturation of $G_0^{List} \cup \{(6), (7), (8)\}$ by SUPC consists of G_0^{List} , (6), (7), (8) and the following constrained literals:*

$$cons(x, cdr(y)) = z \parallel const(x, y, z) \quad (9)$$

$$cons(car(x), y) = z \parallel const(x, y, z) \quad (10)$$

$$car(x) = car(y) \parallel const(x, y) \quad (11)$$

$$cdr(x) = cdr(y) \parallel const(x, y) \quad (12)$$

$$cons(car(x), cdr(y)) = z \parallel const(x, y, z) \quad (13)$$

Proof. The set of axioms $\{(6), (7), (8)\}$ is saturated. The set G_0^{List} is also saturated. It remains to show the same property for the union of both.

Superposition between (6) and (5) and between (7) and (5) respectively yields renamings of (3) and (4), which are immediately removed by the subsumption rule. *Superposition* between (8) and (3) yields the new constrained literal

$$cons(x, cdr(y)) = y \parallel const(x, y). \tag{14}$$

Then, *Superposition* between (14) and (1) gives the constrained literal (9), which subsumes (14). Similarly, *Superposition* between (8) and (4) yields the new constrained literal

$$cons(car(x), y) = x \parallel const(x, y) \tag{15}$$

and *Superposition* between (15) and (1) gives the constrained literal (10), which subsumes (15). *Superposition* between (6) and (10) and between (7) and (9) respectively gives the constrained literals (11) and (12). *Superposition* between (8) and (11) gives the new constrained literal

$$cons(car(x), cdr(y)) = y \parallel const(x, y) \tag{16}$$

and *Superposition* between (16) and (12) gives the constrained literal (13), which subsumes (16). *Superposition* between any axiom and (1) yields constrained literals that are immediately removed by the subsumption rule. Any other application of *Superposition* rule between an axiom and a constrained literal yields a constrained literal that is already in the set $G_0^{List} \cup \{(6), (7), (8)\} \cup \{(9), (10), (11), (12), (13)\}$. Since no other rule can be applied to this set of schematic literals, we conclude that it is saturated. \square

The example given in [9] is not complete. In that paper, it is said that the saturation by *SUPC* of $G_0^{List} \cup \{(6), (7), (8)\}$, consists of the constrained literals (9) and (10), while it also contains (11), (12) and (13). From an encoding of $G_0^{List} \cup \{(6), (7), (8)\}$ our tool generates these five new constrained literals. On this example we can see that the abstraction by schematization is a over-approximation: the abstract saturation computed by *SUPC* is larger than any concrete saturation computed by *UPC*.

4.2 Theory of Records

A record can be considered as a special form of array where the number of elements is fixed. Contrary to the theory of arrays, the theory of records can be specified by unit clauses. The termination of superposition for the theories of records with and without extensionality is shown in [1]. We consider here the theory of records of length 3 without extensionality given by the signature $\Sigma_{Rec} = \bigcup_{i=1}^3 \{rstore_i, rselect_i\}$ and axiomatized by the following set of axioms $Ax(Rec)$:

$$rselect_i(rstore_i(X, Y)) = Y \text{ for all } i \in \{1, 2, 3\}$$

and

$$rselect_i(rstore_j(X, Y)) = rselect_i(X, Y) \text{ for all } i, j \in \{1, 2, 3\}, i \neq j,$$

where X and Y are universally quantified variables. Let G_0^{Rec} be defined as in Section 2.3

Lemma 3. *The saturation of $G_0^{Rec} \cup Ax(Rec)$ by SUPC consists of G_0^{Rec} , $Ax(Rec)$ and the constrained literals*

$$rselect_i(x) = rselect_i(y) \quad || \quad const(x, y)$$

for $i = 1, 2, 3$.

A proof of this lemma can be found in Appendix B. From an encoding of $G_0^{Rec} \cup Ax(Rec)$ our tool generates the schematic saturation given in Lemma 3 which corresponds to the form of saturations described in 1.

5 Conclusion

This paper reported on a prototyping environment for designing and verifying decision procedures. This environment, based on the theoretical studies in [10, 9], is the first implementation including both superposition and schematic superposition calculi. It has been implemented from scratch on the firm basis provided by Maude. Some automated deduction tools are already implemented in Maude, for instance a Church-Rosser checker [6], a coherence checker [7], etc. Our tool is a new contribution to this collection of tools. This environment will help testing new saturation strategies and experimenting new extensions of the original (schematic) superposition calculus. A short term future work is to consider non-unit clauses. Since schematic superposition is interesting beyond the property of termination, we also want to extend the implementation so that we can check deduction completeness and stably infiniteness [9] which are key properties for the combination of decision procedures. We are also interested in developing new schematic calculi for superposition modulo fragments of arithmetic such as Integer Offsets [13] and Abelian Groups [12]. The reported implementation is a firm basis for all these future developments.

Acknowledgments. We are deeply grateful to Santiago Escobar for his help on the use of narrowing in Maude and to Alberto Verdejo for his answers about the strategy language for Maude.

References

- [1] Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. *ACM Trans. Comput. Log.* 10(1) (2009)
- [2] Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. *Inf. Comput.* 183(2), 140–164 (2003)

- [3] Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Unification and Narrowing in Maude 2.4. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 380–390. Springer, Heidelberg (2009)
- [4] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: Specification and programming in rewriting logic. *Theoretical Computer Science* (2001)
- [5] Dershowitz, N., Jouannaud, J.P.: Rewrite systems. In: van Leeuwen, J. (ed.) *Handbook of Theoretical Computer Science, Formal Models and Semantics*, vol. B, pp. 243–320. MIT Press (1990)
- [6] Durán, F., Meseguer, J.: A Church-Rosser Checker Tool for Conditional Order-Sorted Equational Maude Specifications. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 69–85. Springer, Heidelberg (2010)
- [7] Durán, F., Meseguer, J.: A Maude Coherence Checker Tool for Conditional Order-Sorted Rewrite Theories. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 86–103. Springer, Heidelberg (2010)
- [8] Eker, S., Martí-Oliet, N., Meseguer, J., Verdejo, A.: Deduction, strategies, and rewriting. *Electr. Notes Theor. Comput. Sci.* 174(11), 3–25 (2007)
- [9] Lynch, C., Ranise, S., Ringeissen, C., Tran, D.K.: Automatic decidability and combinability. *Inf. Comput.* 209(7), 1026–1047 (2011)
- [10] Lynch, C., Morawska, B.: Automatic decidability. In: *Proc. of 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, Copenhagen, Denmark, pp. 7–16. IEEE Computer Society Press (2002)
- [11] Martí-Oliet, N., Meseguer, J., Verdejo, A.: Towards a strategy language for Maude. *Electr. Notes Theor. Comput. Sci.* 117, 417–441 (2005)
- [12] Nicolini, E., Ringeissen, C., Rusinowitch, M.: Combinable Extensions of Abelian Groups. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 51–66. Springer, Heidelberg (2009)
- [13] Nicolini, E., Ringeissen, C., Rusinowitch, M.: Combining satisfiability procedures for unions of theories with a shared counting operator. *Fundamenta Informaticae* 105(1-2), 163–187 (2010)
- [14] 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)
- [15] Schulz, S.: E - a brainiac theorem prover. *AI Commun.* 15(2-3), 111–126 (2002)
- [16] Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischniewski, P.: SPASS Version 3.5. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 140–145. Springer, Heidelberg (2009)

A Schematic Saturation of Lists

Lemma 1. *Let*

$$G_0^{List} = \{\perp\} \cup \begin{cases} x = y & \parallel \text{const}(x, y) & \text{(7)} \\ x \neq y & \parallel \text{const}(x, y) & \text{(2)} \\ \text{car}(x) = y & \parallel \text{const}(x, y) & \text{(3)} \\ \text{cdr}(x) = y & \parallel \text{const}(x, y) & \text{(4)} \\ \text{cons}(x, y) = z & \parallel \text{const}(x, y, z) & \text{(5)} \end{cases}$$

and let

$$Ax(List) = \begin{cases} \text{car}(\text{cons}(X, Y)) = X & \text{(6)} \\ \text{cdr}(\text{cons}(X, Y)) = Y & \text{(7)} \end{cases}$$

The set $G_0^{List} \cup Ax(List)$ is saturated by SUPC.

Proof. The set of axioms $\{\text{(6)}, \text{(7)}\}$ is saturated. The set G_0^{List} is also saturated. It remains to show the same property for the union of both.

Superposition between (6) and (5) yields a renaming of (3) , which is immediately removed by the subsumption rule. Similarly, *Superposition* between (7) and (5) yields a renaming of (4) , which is removed by the subsumption rule as well. *Superposition* between any axiom and (1) yields a schematic literals that are immediately removed by the subsumption rule. Since no other rule can be applied between an axiom and a schematic literal, we conclude that the set $G_0^{List} \cup \{\text{(6)}, \text{(7)}\}$ is saturated. \square

B Schematic Saturation of Records

Lemma 3. *Let G_0^{Rec} be the set that consists of the empty clause \perp and the constrained literals*

$$x = y \parallel \text{const}(x, y) \quad (17)$$

$$x \neq y \parallel \text{const}(x, y) \quad (18)$$

$$\text{rstore}_1(x, y) = z \parallel \text{const}(x, y, z) \quad (19)$$

$$\text{rstore}_2(x, y) = z \parallel \text{const}(x, y, z) \quad (20)$$

$$\text{rstore}_3(x, y) = z \parallel \text{const}(x, y, z) \quad (21)$$

$$\text{rselect}_1(x) = y \parallel \text{const}(x, y) \quad (22)$$

$$\text{rselect}_2(x) = y \parallel \text{const}(x, y) \quad (23)$$

$$\text{rselect}_3(x) = y \parallel \text{const}(x, y) \quad (24)$$

Let $Ax(Rec)$ be the set of axioms

$$\text{rselect}_1(\text{rstore}_1(X, Y)) = Y \quad (25)$$

$$\text{rselect}_2(\text{rstore}_2(X, Y)) = Y \quad (26)$$

$$rselect_3(rstore_3(X, Y)) = Y \quad (27)$$

$$rselect_1(rstore_2(X, Y)) = rselect_1(X) \quad (28)$$

$$rselect_1(rstore_3(X, Y)) = rselect_1(X) \quad (29)$$

$$rselect_2(rstore_1(X, Y)) = rselect_2(X) \quad (30)$$

$$rselect_2(rstore_3(X, Y)) = rselect_2(X) \quad (31)$$

$$rselect_3(rstore_1(X, Y)) = rselect_3(X) \quad (32)$$

$$rselect_3(rstore_2(X, Y)) = rselect_3(X) \quad (33)$$

The saturation of $G_0^{Rec} \cup Ax(Rec)$ by *SUPC* consists of G_0^{Rec} , $Ax(Rec)$ and the following constrained literals:

$$rselect_1(x) = rselect_1(y) \parallel const(x, y) \quad (34)$$

$$rselect_2(x) = rselect_2(y) \parallel const(x, y) \quad (35)$$

$$rselect_3(x) = rselect_3(y) \parallel const(x, y) \quad (36)$$

Proof. The set of axioms $Ax(Rec)$ is saturated. The set of schematic literals G_0^{Rec} is also saturated. It remains to show the same property for the union of both.

Superposition between (25) and (19) yields a renaming of (22), which is immediately removed by the subsumption rule. It is similar for the indices 2 and 3, between (26) and (20) and between (27) and (21).

Superposition between (28) and (20) yields the constrained literal (34). Afterwards, *Superposition* between (29) and (21) yields a renaming of (34), which is immediately removed by the subsumption rule. It is similar for the indices 2 and 3, between (30) and (19) and between (32) and (19).

Superposition between any axiom and (17) yields schematic literals that are immediately removed by the subsumption rule. Since no other rule can be applied between an axiom and a schematic literal, we conclude that the set $G_0^{Rec} \cup Ax(Rec) \cup \{(34), (35), (36)\}$ is saturated for *SUPC*. \square

Author Index

- Ábrahám, Erika 139, 182
Arusoaie, Andrei 83
- Bae, Kyungmin 99
Basu, Ananda 1
Bensalem, Saddek 1
Bozga, Marius 1
Bruni, Roberto 118
- Corradini, Andrea 118
- Eckhardt, Jonas 54
Ellison, Chucky 83
- Fadlisyah, Muhammad 139
- Gadducci, Fabio 118
Giorgetti, Alain 221
Gutiérrez, Raúl 162
- Hills, Mark 10
- Klint, Paul 10
Kouchnarenko, Olga 221
- Lepri, Daniela 182
Lluch Lafuente, Alberto 118
Lucanu, Dorel 31
- Meseguer, José 54, 99, 162
Mühlbauer, Tobias 54
- Ölveczky, Peter Csaba 139, 182
- Riesco, Adrián 201
Ringeissen, Christophe 221
Rocha, Camilo 162
Roşu, Grigore 31, 83
- Şerbănuţă, Traian Florin 31, 83
Sifakis, Joseph 1
- Tushkanova, Elena 221
- Vandin, Andrea 118
Vinju, Jurgen J. 10
- Wirsing, Martin 54