Electronic Notes in Theoretical Computer Science

6th International Workshop on Rewriting Logic and its Applications

WRLA’06

Vienna, Austria

April 1–2, 2006

Guest Editors:

GRIT DENKER AND CAROLYN TALCOTT
## Contents

**Preface** .................................................. iii

**ARVIND** (Invited Speaker)  
The Semantics of a Parallel Language with Modules with Guarded Interfaces .............................................. 1

**FABRICIO CHALUB** and **CHRISTIANO BRAGA**  
Maude MSOS Tool ............................................. 3

**HORATIU CIRSTEA**, **CLÉMENT HOUTMANN**, **BENJAMIN WACK**  
Distributive Rho-Calculus ................................. 18

**FRANCISCO DURAN**, **ALBERTO VERDEJO**, **ADRIAN RIESCO**  
Implementation of Mobile Maude ........................... 35

**AZADEH FARZAN**, **JOSÉ MESEGUER**  
Making Partial Order Reduction Tools Language-Independent .... 56

**MARK HILLS**, **TRAIAN SERBANUTA**, **GRIGORE ROSU**  
A Rewrite Framework for Language Definitions and for Generation of Efficient Interpreters ............................ 76

**MIKE KATELMAN** and **JOSÉ MESEGUER**  
A Rewriting Semantics for ABEL with Applications to Hardware/Software Co-Design and Analysis ....................... 95

**PETER CSABA ÖLVECZKY**, **JOSÉ MESEGUER**  
Abstraction and Completeness for Real-Time Maude ............. 128

**ANTOINE REILLES**  
Canonical Abstract Syntax Trees ............................ 154

**DILIA RODRIGUEZ**  
On Modelling Sensor Networks in Maude ........................ 170
GUSTAVO SANTOS-GARCÍA, MIGUEL PALOMINO
Solving Sudoku Puzzles with Rewriting Rules ................. 186

RALF SASSE AND JOSÉ MESEGUER
Java+ITP: A Verification Tool Based on Hoare Logic and Algebraic
Semantics ..................................................... 202

MARTIN WIRSING, GRIT DENKER, CAROLYN TALCOTT, ANDY POGGIO, LINDA BRIESEMEISTER
A Rewriting Logic Framework for Soft Constraints .............. 223

STEVEN EKER (System demo)
Maude: What’s new in 2.2? ........................................ 240

GERMAIN FAURE AND ANTOINE REILLES (System demo)
TOM + XRHO the explicit rewriting calculus ..................... 241

ARTUR BORONAT, JOAQUÍN ORIENTE, ABEL GÓMEZ, JOSÉ Á. CARSÍ, ISIDRO RAMOS (Tool demo)
MOMENT-OCL: Algebraic Specifications of OCL 2.0 within the
Eclipse Modeling Framework ..................................... 243

MANUEL CLAVEL (Tool demo)
ITP/OCL: A Rewriting-Based Validation Tool for UML+OCL Class
Diagrams .......................................................... 245

ADRIAN RIESCO (Tool demo)
Web ITP Tool Server: A Web-Based Interface for the ITP Tool ... 246
Preface

This volume contains the preliminary proceedings of the *Sixth International Workshop on Rewriting Logic and its Applications* (WRLA 2006) held in Vienna, Austria, April 1-2, 2006, as a satellite workshop of the *European Joint Conferences on Theory and Practice of Software* (ETAPS 2006).

Rewriting logic (RL) is a natural semantic framework for representing concurrency, parallelism, communication and interaction, and an expressive (meta)logical framework for representing logics. It can then be used for specifying a wide range of systems and logical and programming languages in various application fields. In recent years, several executable specification languages based on RL (ASF+SDF, CafeOBJ, ELAN, Maude) have been designed and implemented. The aim of the WRLA workshop series is to bring together researchers with a common interest in RL and its applications, and to give them the opportunity to present their recent works, discuss future research directions, and exchange ideas.

The topics of the workshop comprise, but are not limited to,

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

Twenty papers were submitted, covering most of the topics above. Out of them the program committee selected 13 to be included in the workshop program, after a reviewing process in which every paper was evaluated by at least three referees. The workshop program also includes an invited lecture.
by Arvind, a rewriting competition, and several system and tool demonstrations.

The final proceedings of the workshop will appear in the *Electronic Notes in Theoretical Computer Science* (ENTCS) series, which is published electronically through the facilities of Elsevier Science B.V. and under its auspices. The ENTCS volumes are available online at http://www.elsevier.nl/locate/entcs.

Four of the previous WRLA workshops were held as independent events in Asilomar, California, USA (1996), Pont-à-Mousson, France (1998), Kanazawa, Japan (2000), and Pisa, Italy (2002). The fifth WRLA workshop was held as a satellite of ETAPS in Barcelona, Spain (2004). The corresponding proceedings have been published as volumes 4, 15, 36, and 71 in ENTCS. Also, selected papers from WRLA’96 have been published in a special issue of *Theoretical Computer Science* in 2002, and a special issue of *Higher Order and Symbolic Computation* consisting of selected papers from WRLA’04 is in preparation.

Even a small workshop like WRLA requires the effort of many people to make it happen. We want to thank first of all the authors who submitted papers and showed interest in the subject. The members of the program committee and the external referees appointed by them worked hard to satisfactorily complete the reviewing process on time, and we very much appreciate their efforts. Thanks to the ETAPS organizers, especially Andreas Krall for taking care of all the local arrangements, and answering many questions about organizational details. Thanks also to thank Mike Mislove, managing editor of the ENTCS series, for his assistance.

We hope that WRLA 2006, continues the tradition of stimulating discussion and providing participants with a clear view of the latest developments and research directions in rewriting logic and its different applications.

*Menlo Park, CA, February 13, 2006*  
Grit Denker and Carolyn Talcott
Program Committee

Roberto Bruni  Università di Pisa
Manuel Clavel  Universidad Complutense de Madrid
Grit Denker  SRI International, Menlo Park (co-Chair)
Francisco Durán  Universidad de Malaga
Steven Eker  SRI International, Menlo Park
Kokichi Futatsugi  JAIST, Nomi
Claude Kirchner  INRIA Lorraine & LORIA, Nancy
Salvador Lucas  Universidad Politécnica de Valencia
Narciso Martí-Oliet  Universidad Complutense de Madrid
José Meseguer  University of Illinois at Urbana-Champaign
Ugo Montanari  Università di Pisa
Pierre-Etienne Moreau  INRIA Lorraine & LORIA, Nancy
Peter Ølveczky  University of Oslo
Grigore Rosu  University of Illinois at Urbana-Champaign
Mark-Oliver Stehr  SRI International, Menlo Park
Carolyn Talcott  SRI International, Menlo Park (Chair)
Martin Wirsing  Ludwig-Maximillian University, München

External referees

Marco Bellia  Santiago Escobar  Fabio Gadducci
Mark Hills  Michael Koehler  Ivan Lanese
Kazuhiro Ogata  Antoine Reilles  Christophe Ringeissen
Traian Florin Serbanuta  Josep Silva  Emilio Tuosto
Alicia Villanueva

Organizing Committee

Grit Denker and Carolyn Talcott
Computer Science Laboratory
SRI International
Menlo Park, CA
The Semantics of a Parallel Language with Modules with Guarded Interfaces
(Invited Talk)

Arvind

Computer Science and Artificial Intelligence Laboratory
M.I.T. Cambridge, MA, USA

Abstract

Bluespec is a hardware design language based on the concept of Guarded Atomic Actions [4,1]. In TRS interpretation, a guarded atomic action is a rule which can execute if its guard (i.e., predicate) holds on the state (i.e., term) [1]. All legal behaviors of a Bluespec program can be explained in terms of an execution of rules in some sequential order. In actual implementation many rules execute simultaneously but preserve this one-rule-at-time semantics. (In general Bluespec rules are not confluent. This is desirable because a Bluespec program, when viewed as a specification, can embody nondeterminism). One-rule-at-a-time semantics provide a robust framework for verification.

One can think of a module in Bluespec as a hardware box with ports and wires. However, unlike popular hardware design languages like Verilog, one thinks of a port in Bluespec as a method, similar to a method in an object oriented language. Bluespec methods have another characteristic, that is, a method can be invoked only when its guard is true. Interface guards also act as implicit conditions in a rule that calls those methods. A rule can execute only if all its implicit conditions are satisfied.

Each method has a set of input and output wires and a light to indicate if the method is ready to be invoked. The light corresponds to the guard condition. The output of a read method is not valid unless the corresponding light is green. An action method, in addition, has a button which needs to be pressed to invoke it. An action method can be invoked only when the corresponding light is green.

The language also specifies the consequence if several buttons of a module are pressed simultaneously; it is either illegal to do so or it results in the state of system as if method 1 executed before method 2 or vise versa [2]. (The notion of simultaneity can be introduced by assuming that each action takes a bounded amount of time before its effect takes place; two actions or method calls are considered simultaneous if they are initiated before the effect of either takes place).
We will present an operational semantics of Bluespec in terms of a simple sequential language. We will also show how, given a module and a specification for parallel semantics for its interface, one can automatically derive another module to satisfy the new parallel semantics [3].

References


Abstract

Modular structural operational semantics (MSOS) is a new framework that allows structural operational semantics (SOS) specifications to be made modular in the sense of not imposing the redefinition of transition rules, which is the case in SOS specifications, when an extension is made. Maude MSOS tool (MMT) is an executable environment for MSOS implemented in Full Maude as a realization of a semantics-preserving mapping between MSOS and rewriting logic (RWL). The modular SOS definition formalism (MSDF) is the specification language supported by MMT. MSDF syntax is quite close to MSOS mathematical notation and user-friendly by allowing several syntactic components to be left implicit. MMT joins the support for modularity with a user-friendly syntax together with the efficient execution and analysis of the Maude engine. We have used MMT in several different examples from programming languages semantics and concurrent systems. This paper reports on the development of MMT and its application to these two classes of specifications.

Key words: Modular SOS, Modular SOS Definition Formalism, Rewriting Logic, Maude

1 Introduction

Structural operational semantics (SOS) [16] is a well known formalism for the specification of programming language semantics (e.g. [16]) and concurrent systems (e.g. [10]). However, an important aspect of specifications [12] was left open by Plotkin in his seminal lecture at Aarhus: modularity.

Plotkin’s framework requires that transition rules in a given specification, handling certain semantic information, to be modified, when extensions to the specification alter the semantic information handled by the transition rules. That is, SOS is not modular. For instance, the transition rules of a specification of

---

1 Email: fchalub@ic.uff.br
2 Email: cbraga@ic.uff.br

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
functional constructs for a programming language, that has an environment as a semantic component, have to be modified when the specification is extended with imperative constructs, that has a store as semantic component. This modification is necessary since the extended configuration has another component for the mapping from memory locations to values, that represent the store.

Plotkin himself declares in [16]: “As regards to modularity, if we get the other things in a reasonable state, then current ideas for imposing modularity on specifications will prove useful”. Quoting Mosses [12], as opposed to support to modularity, “the other things” seem to be in good shape.

Mosses’s modular structural operational semantics (MSOS) [12] solves the modularity problem in SOS specifications. The intuition is elegantly simple: given a label in a transition, a semantic component becomes a value bound to an index within the label which has a record-like structure, encapsulating the semantic information. Moreover, the record structure is extensible, therefore transition rules may refer to the label indices relevant to the language construct they specify. Transition rules in MSOS are then specified once and for all. For instance, transition rules for functional constructs only mention the environment and transition rules for imperative constructs relate the language constructs to store. Therefore, in the extension with imperative constructs, the transition rules for the functional constructs are not modified.

The aim of this paper is to report on an effort for building and applying an executable environment for MSOS specifications: the Maude MSOS Tool (MMT). MMT is a formal tool [4] implemented in Full Maude [5] as a realization of a semantics-preserving mapping between MSOS and rewriting logic (RWL) [8]. MMT supports the modular SOS definition formalism (MSDF) [11,3], a concrete syntax for MSOS, developed by the first author and Mosses, which appears to be quite suitable for such modular specifications representing closely the mathematical notation of MSOS in [12]. We have used MMT to specify and analyze several programming language semantics specifications and concurrent systems. The first example is a set of MSDF specifications for basic semantic primitives, created by Mosses, known as Constructive MSOS (see e.g. [13]). The semantics of a programming language is given by a transformation from the programming language’s syntax to Constructive MSOS primitives. We used Constructive MSOS to give executable (and analyzable) semantics for two programming languages in two different paradigms, namely functional and object-oriented. The second example is the semantics of a “lazy evaluation” functional programming language named Mini-Freja [15]. The third and fourth examples come from concurrent systems. The third is an specification for CCS and fourth is composed by several distributed algorithms specifications: Lamport’s bakery algorithm, dining philosophers, leader election on an asynchronous ring, Peterson’s solution for mutual exclusion, mutual exclusion with semaphores, and the thread game. The implementation of MMT and its many applications, constitute the contributions

3 http://maude-msos-tool.sourceforge.net/
4 MMT specifications can be downloaded from http://maude-msos-tool.sf.net/specs.
reported by this paper.  

The remainder of this paper is organized as follows. Section 2 discusses MSOS main features. Section 3 recalls the formal foundations of MMT. Section 4 discusses the implementation of MMT. Section 5 outlines the case studies we have developed using MMT. We conclude the paper in Section 6 discussing related efforts within the RWL community and the SOS community and describing our next steps.

2 Modular SOS

This Section presents key MSOS concepts and adapts material from [12].

The key to MSOS modularity lies on its transition labels which encapsulate the semantic information. MSOS labels are structured as records with each index in a record typed as read-only, read-write or write-only. A read-only index holds a value that does not change in a transition. Read-write indices are always declared in pairs: an index i represents information before the transition and index i’ represents information after the transition. A write-only index, always declared “quoted” as in a side-effected read-write index, holds a value that may only be updated in a transition. (Thus, quoted indices always refer to information after a transition.) If we consider the specification of programming languages semantics, a read-only index holds an environment-like structure, read-write indices holds a memory store-like structure, and a write-only index holds (a free monoid of) emitted information, such as output or synchronization signals.

Let us illustrate this. The following SOS transition rule specifies (part) of the semantics for a let expression, typical in functional languages,

\[
\rho_1[\rho_0] \vdash e \rightarrow e' \\
\rho_1 \vdash \text{let } \rho_0 \text{ in } e \rightarrow \text{let } \rho_0 \text{ in } e'
\]

where \(\rho_0\) and \(\rho_1\) are sets of environment bindings, the SOS expression \(\rho_1[\rho_0]\) means that the set of bindings \(\rho_1\) is overridden by \(\rho_0\), and \(e\) and \(e'\) are expressions in the object language being specified. Let us extend this specification with imperative assignments to memory locations given by the following transition rule,

\[
\rho \vdash (l := v, \sigma) \rightarrow (\text{skip}, \sigma[l \mapsto v])
\]

where \(l\) is a memory location, \(v\) is a (storable) value, \(\sigma\) is the memory location before the transition, \(\sigma[l \mapsto v]\) is the memory location after the transition which is simply \(\sigma\) updated with the pair \(l \mapsto v\), and \(\text{skip}\) is the identity of command composition. It should be clear now that the assignment extension implies the modification of the original transition rule for the let expression, since an assignment requires the addition of memory locations.

Let us see now how the semantics of these two constructs could be given in MSOS. The following MSOS transition rule specifies (part) of the semantics for
a \textbf{let} expression,

\[
\begin{align*}
\text{let } \rho_0 \text{ in } e & \quad \rightarrow \quad \text{let } \rho_0 \text{ in } e' \\
\end{align*}
\]

where the metanotation '\{-i = v, \ldots\}' represents a labeled transition with a ‘\(i\)’ any given index, ‘\(v\)’ a (general) value bound to ‘\(i\)’, and the ellipsis ‘\(\ldots\)’ representing the (same) remainder of the label, whose values \textit{may} change. The remaining variables are typed as before. The index \(\rho\) is a read-only index capturing the environment. The meaning of an assignment could be given by the following transition rule,

\[
l := v - \{\sigma = \sigma_0, \sigma' = \sigma_0[l \mapsto v], -\} \rightarrow \text{skip}
\]

where \(\sigma\) is a read-write index and ‘\(-\)’ means that the remainder of the label is unobservable. MSOS uses a “prime notation” to represent the value of an index \textit{after} a transition. Therefore, the memory before the transition is given by the unprimed \(\sigma\) and the memory \textit{after} the transition is given by the primed \(\sigma\). Note that with the MSOS formulation, the transition rule for \textbf{let} need not to be modified, since the new rule for assignments simply range over the new index \(\sigma\), different from index \(\rho\), the one used by the \textbf{let} transition rule.

Computation in MSOS is given by a sequence of adjacent transitions with composable labels. Given two transitions \(\gamma_1 - \alpha_1 \rightarrow \gamma_1'\) and \(\gamma_2 - \alpha_2 \rightarrow \gamma_2'\), with configurations \(\gamma_1, \gamma_1', \gamma_2, \gamma_2'\) and labels \(\alpha_1\) and \(\alpha_2\), they are adjacent iff \(\gamma_1' = \gamma_2\). Label composability is defined by cases on each possible index type: (i) for a read-only index \(i\), \((\alpha_1; \alpha_2).i = \alpha_1.i = \alpha_2.i\); (ii) for read-write pair of indices \(i\) and \(i'\), \((\alpha_1; \alpha_2).i = \alpha_1.i\) and \((\alpha_1; \alpha_2).i' = \alpha_2.i'\), assuming \(\alpha_1.i' = \alpha_2.i\); (iii) for a write-only index \(i'\), \((\alpha_1; \alpha_2).i' = \alpha_1.i' \cdot \alpha_2.i'\), where ‘\(\cdot\)’ is the binary operation of the free monoid bound to \(i\).

To conclude our brief presentation of MSOS, let us define generalized transition systems (GTS), the models of MSOS specifications. A GTS is a tuple \(\langle \Gamma, \mathcal{A}, -\rightarrow, T \rangle\), such that \(\mathcal{A}\) is a category (named label category) with morphisms \(\mathcal{A}\), and \(\langle \Gamma, \mathcal{A}, -\rightarrow, T \rangle\) is a labeled terminal transition system, with \(\Gamma\) the set of configurations, \(-\rightarrow \subseteq \Gamma \times \mathcal{A} \times \Gamma\) the transition relation, and \(T \subseteq \Gamma\) the set of terminal states.

### 3 RWL as a Semantic Framework for MSOS

In this Section we briefly recall the formal foundations of Maude MSOS Tool (MMT) [8], that is, MMT as an implementation of a formally defined semantics-preserving mapping from MSOS to RWL. The focus is on the intuition behind the ideas and not on the actual formalizations, which can be found in the cited references. The following paragraphs outline the main aspects of the mapping to

---

\[^{5}\text{A labeled terminal transition system is a labeled transition system where the terminal configurations are distinguished.}\]
help a proper reading of Section 4, as follows: the mapping from MSOS to RWL, Maude as a formal metatool, and MMT as one such formal tool. We use Maude as a concrete syntax to define RWL theories.

The mapping from MSOS to RWL was defined by one of the present authors together with Haeusler, Meseguer, and Mosses [1] and then further developed jointly with Meseguer. The mapping produces, for each MSOS specification, a rewrite theory that includes a record theory, defined in the ‘RECORD’ module, applying a technique named modular rewriting semantics (MRS) [2]. A record is essentially a set of index-value pairs, with non-duplicated indices, that captures the notion of MSOS labels, with each record field declared by a membership axiom, such as ‘mb (rho : E) : Field.’, with ‘E’ of sort ‘Env’ a subsort of ‘Component’ and ‘rho’ a constant of sort ‘Index’.

fmod RECORD is
  sorts Index Component Field PreRecord Record .
  subsort Field < PreRecord .
  op null : -> PreRecord [ctor] .
  op _,_ : PreRecord PreRecord -> PreRecord [ctor assoc comm
                     id: null] .
  op _:_ : [Index] [Component] -> [Field] [ctor] .
  op duplicated : [PreRecord] -> [Bool] .
  var I : Index . vars C C’ : Component . var PR : PreRecord .
  eq duplicated((I : C),(I : C’), PR) = true .
  cmb {PR} : Record if duplicated(PR) /= true .
endfm

A transition rule is represented as a conditional rewrite rule in the generated rewrite theory, one of the choices proposed in [7]. Also, the generated rewrite rules should mimic operational semantics transitions. Therefore, RWL’s inference rules for reflexivity, transitivity and congruence should not apply. This is controlled by the ‘step’ rule defined in the ‘RCONF’ module.

mod RCONF is protecting RECORD .
  sorts Program Conf .
  op <_,_> : Program Record -> Conf [ctor] .
  op {_} : [Program] [Record] -> [Conf] [ctor] .
  op [_,_] : [Program] [Record] -> [Conf] [ctor] .
  vars P P’ : Program . vars R R’ : Record .
endm

The ‘step’ rule requires that all the transitions from an MSDF specification, which includes the ‘RCONF’ module, are represented as rewrites of the form \{t\} \rightarrow \{t’\}. Thus, an MSOS computation is represented as a sequence of rewrites of the form \langle t_1 \rangle \rightarrow \langle t_2 \rangle \rightarrow \cdots \rightarrow \langle t_n \rangle, where \langle t, t’, t_1, t_2, \ldots, t_n \rangle are terms constructed out of the MSDF configurations and labels, as described in the next paragraph. The reflexivity inference rule is not applicable because the premise of the ‘step’ rule forces a change in the term. The rewrite cannot be transitive because there is no transition rule with a square bracket constructor on the left-
hand side. The congruence inference rule is not applicable because subterms of $t$ are not in the form $\{t\}$, required by the rewrite rules. (Of course, we assume that the configuration constructors, declared in the signature of module ‘RCONF’, do not appear as subterms of $t$, otherwise a proper renaming of the configuration constructors would be required.)

Note that the labeled transition rules induce a (labeled transition) relation with elements structured as triples, with the label being its second projection, as opposed to rewrite rules that induce a (rewriting) relation with pairs as elements. To produce an element of the rewriting relation from an element from the labeled transition relation, the label from the transition relation is split into its pre and post projections, which are defined below. An element of the rewriting relation, built from an element from the labeled transition relation, is then given by a pair of pairs: the first pair is given by the first projection of the element from the labeled transition relation (e.g. the program syntax), and the pre-projection of the label, (e.g. the pre-projection of the environment); and the second pair is given by the second projection of the element from the labeled transition relation (e.g. the program syntax), and the post-projection of the label (e.g. the post-projection of the environment).

The pre- and post-projections are defined interpreting the MSOS label category (see Section 2) as a pre-order, with ordering relations defined for each possible index type in a label, that is, read-only, read-write and write-only. Thus, (i) both the pre- and post-projections of a read-only index are the value bound to that index in a label; (ii) the pre-projection of a read-write index is the first projection of the pair bound to that index in a label and the post-projection of the index is the second projection of the value bound to that index; (iii) we consider a trace semantics for write-only indices, thus given a write-only index in a labeled transition in a: a) conclusion of a transition rule, the pre-projection of the index is the prefix of the monoid bound to the index, and the post-projection is given by appending the value emitted by the transition, in the given index, to the value produced by the pre-projection; b) premise of a transition rule, the pre-projection is the identity element of the monoid and the post-projection is calculated in the same way as for the labeled transition in the conclusion of a transition rule. The identity element is used because we want only the information emitted at this index, in this transition, and not the complete trace. The same rules for the calculations of pre- and post-projections apply to a label resulting from the composition of two other labels.

The mapping from MSOS to RWL is semantics-preserving in the precise sense of a bisimulation [8] between the models of MSOS specifications (GTS from Section 2) and the rewrite theories generated by the mapping from MSOS to RWL, that is, a rewrite theory with the rewriting relation representing small-step transitions as explained above.

Maude has been shown to be a formal metatool [4], which means that an executable environment can be built in Maude for a given concept (e.g., logic, specification language or model of computation) once a proper mapping is defined.
between this concept and rewriting logic. Such a mapping then is implemented as a (transformation) metafunction in Maude that, given a term in the algebra that formalizes the signature of the given concept, produces a term in the signature of Maude modules. Moreover, Full Maude may be used to create such an environment. Full Maude endows the Maude system with an extensible module algebra. In order to create the executable environment, Full Maude should be extended with the above mentioned metafunction. (Due to space limitations we do not detail the process of extending Full Maude.)

MMT is a formal tool implemented using the mapping from MSOS to RWL recalled from [8] in this Section as the transformation metafunction, with modular SOS definition formalism (MSDF) as concrete syntax for MSOS and Maude for RWL. Section 4 continues this paper exemplifying MSDF syntax and how MSDF specifications are transformed into Full Maude system modules.

4 Maude MSOS Tool

4.1 MSDF

MSDF is a concrete syntax for MSOS developed by the first author and Mosses. This Section describes the main elements of MSDF: modules, module inclusions, syntax definitions, label definitions, and transitions, using as example an specifications for let expressions, where values are integers.

MSDF modules begin with the string ‘msos’ and end with ‘sosm’. Module inclusion can be implicit or explicit. An MSDF module \( M \) implicitly includes other modules that declare sets used in the declaration of datatypes and labels of \( M \). Explicit inclusion of other modules is done using the Prolog-like ‘see’ syntax. In our example, we declare a module ‘LET’ that implicitly includes the specification for set ‘Int’, since our let expressions only handle integers as values.

Syntax declarations are made using sets and functions on sets. It is also possible to use parameterized sets, such as tuples, lists, and maps. For instance, the module ‘LET’ defines the let construct using mixfix syntax, and a Backus-Naur Form (BNF) style, as a function from Dec \( \times \) Exp to Exp, declared as follows:

\[
\text{Exp ::= let Dec in Exp | Int | Id .}
\]

For each BNF declaration, two derived sets are implicitly declared for each set \( s \) in the declaration: ‘\( s^* \)’ and ‘\( s^+ \)’ for tuples and non-empty tuples of elements from \( s \), respectively. (The ‘LET’ module, however, does not use any of these derived sets.) Other parameterized sets are defined with the following syntax: ‘\((s)\text{List}\)’ defines finite lists, ‘\((s)\text{Set}\)’ for finite sets, and ‘\((s,k)\text{Map}\)’ for finite maps from \( s \) to \( k \). The ‘LET’ module declares a map from identifiers to integers to define the set ‘Env’, representing the environment, declared as follows:

\[
\text{Env = (Id, Int)Map .}
\]

Regarding label declaration, the indices of the components define a field to be read-only, read-write, or write-only using a “prime notation”, as explained in
Section 2: if there is a single, unprimed index, then the field defines a read-only component; an index that appears both unprimed and primed defines a read-write component with both components holding the same type of values; and a single primed index defines a write-only component. The ellipsis syntax ‘...’ means that the label declaration may be further extended with new components. In module ‘LET’, the declaration for the label with an environment component is as follows:

\[ \text{Label} = \{ \text{env} : \text{Env}, \ldots \} \]

The transitions in MSDF should operate on typed value-added syntactic trees. Therefore the type of the term to be matched against should also be specified using the operator ‘:’. The type of the right-hand side of a transition is assumed to be the same as the type of the left-hand side. This is useful, for example, in languages where separate environments are provided for values and closures (e.g., Common Lisp). By using typed syntactic trees, the rule for the lookup of identifiers might use its type (a value-bound identifier or a closure-bound identifier) to choose the correct environment to use.

The syntax for transition rules in MSDF closely represents mathematical notation for inference rules. The premise is a (comma separated) conjunction of labeled transitions, predicates or Maude-like matching equations. As an example of a MSDF transition, the following transition rule specifies Rule 1, on Section 2.

\[
\text{Exp} -\{ \text{env} = (\text{Env1} / \text{Env2}), \ldots \} \rightarrow \text{Exp}'
\]

\[\text{-----------------------------------------------}\]

\[ (\text{let Env1 in Exp}) : \text{Exp} -\{ \text{env} = \text{Env2}, \ldots \} \rightarrow \text{let Env in Exp}' \]

In MSDF there is no explicit variable declaration: every set name defines the prefix of a variable that may be “primed” (end with ‘’’) or may end with a number, such as ‘Exp’ and ‘Env1’, respectively. The ‘Exp’ after the colon on the left-hand side of the conclusion indicates the exact type of the term that will be matched by this transition.

\[\text{Exp}' \]

\[\text{let Env in Exp}' \]

\[\text{-----------------------------------------------}\]

\[ (\text{let Env1 in Exp}) : \text{Exp} -\{ \text{env} = \text{Env2}, \ldots \} \rightarrow \text{let Env in Exp}' \]

4.2 Compiling MSDF into Full Maude

As exemplified in Section 4.1, an MSDF module has four components: a module inclusion section, syntax and datatype definition section, label declaration section, and transition rules declaration section. A high level view of the compilation into Full Maude system modules is outlined as follows: ‘see’ declarations are transformed into module inclusions and implicit module inclusions are solved; syntax and datatype declarations are transformed into an equational theory; label declaration is transformed into an extension of the ‘RECORD’ theory, and the transition rules are transformed into conditional rewrite rules, with the label expressions transformed into record expressions as explained in Section 3. The remainder of this Section outlines the transformation for each component, and explains how the MSDF module LET in Section 4.1 is transformed into a Maude system module. The complete transformation details can be found in [3].
Compilation of module inclusions. The module inclusions in MSDF are converted directly into Full Maude inclusions. Implicit module inclusions are solved by searching Full Maude’s database for the module name that declares each referenced set in the syntax declarations and label declarations of an MSDF module.

The Maude system module generated from the LET MSDF specification imports several modules, including views for automatically generated sequences of the MSDF sets, a view for the finite map representing the environment, and a module for built-in integers.

Compilation of syntax definitions and datatypes. The syntax declarations in the MSOS specification are used to generate the signature of the Full Maude module, by converting each set declaration into a sort declaration, each subset inclusion into a subsort inclusion, and each function declaration into an operator.

Parameterized sets in MSDF—lists, sets, and maps—are converted into parameterized sorts in Full Maude. For each parameterized type, there is a built-in parameterized Full Maude functional module that implements the functionality of the relevant datatype. For example, tuples are transformed into an instance of module ‘\texttt{SEQUENCE(X::TRIV)}’, which defines parameterized sorts ‘\texttt{Seq(X)}’ and ‘\texttt{NeSeq(X)}’.

Each function declaration ‘\(f : s_1 \times \cdots \times s_n \to s\)’ in the set of functional symbols in the syntax declaration of a MSDF specification, compiles into an operator ‘\(f : s'_1 \times \cdots \times s'_n \to s'\)’, where each set \(s_i\) is converted into a sort \(s'_i\) according to the compilation rules for the implicit sets into parameterized sorts.

Any attribute in \(f\), such as \texttt{assoc} or \texttt{comm}, is moved verbatim to the generated operator. The function \(f\) may be in mixfix format. In this case, the generated operator name is constructed by keeping all lowercase identifiers from \(f\) and substituting all uppercase identifiers by underscores. The sorts in the domain of the generated operator are named after the uppercase identifiers in \(f\).

To avoid preregularity problems, we chose not to subsort ‘\texttt{Program}’ (from the ‘\texttt{RECORD}’ module—see Section 3) with sorts generated from the syntax of an MSDF module. (The same for the ‘\texttt{Component}’ sort. See the paragraph Compilation of label declarations below.) This implies a special treatment of the \texttt{step} rule, as follows. To generate the configuration constructors for the step rule, the transformation function creates a set \(S_{\text{max}}\) with the top sorts from each connected component, induced by the subsort relations, generated from the syntax declaration section of a MSDF specification. For each sort \(s \in S_{\text{max}}\) the following operators are declared in the signature of the generated Full Maude module: ‘\(<\_,\_\> : s \times \text{Record} \to \text{Conf}\)’ for the \texttt{step} constructor, ‘\(\{\_,\_\} : s \times \text{Record} \rightarrow \text{Conf}\)’ for the left-hand side configuration constructor and ‘\(\[\_,\_\] : s \times \text{Record} \rightarrow \text{Conf}\)’ for the right-hand side configuration constructor, both declared as partial functions. Subsort overloading allows the declaration of these operators only for the top sorts.

A typed syntactic tree used in transitions is represented by a pair ‘\texttt{t:::q}\)’, where \(t\) is the term representing the syntactic tree and \(q\) is a quoted-identifier that
represents the sort. This quoted-identifier is the name of the sort prefixed with a single quote, such as ‘‘\text{\texttt{Exp}}’’. For each sort \(s \in S_{\text{max}}\) an operator \(\text{\texttt{\_\_\_\_\_\_}} : s \times \text{\texttt{Qid}} \rightarrow s\) is declared in the signature of the generated Full Maude system module.

The step rules are then generated for each sort \(s \in S_{\text{max}}\):

\[
crl \langle X : \cdots : qid(s), R \rangle \rightarrow \langle X' : \cdots : qid(s), R' \rangle
\]

\[
\text{if } \{X : \cdots : qid(s), R\} \rightarrow [X' : \cdots : qid(s), R'] \quad [\text{step}]
\]

where \(X, X'\) are variables of the sort \(s\), and \(R, R'\) are variables of the sort ‘\text{\texttt{Record}}’. The function \(\text{\texttt{qid}}(s)\) converts a sort name into a quoted-identifier.

The signature of Maude system module generated from the ‘\text{\texttt{LET}}’ MSDF specification contains sort declarations for every non-terminal on the right-hand side of the BNF declaration, such as ‘\text{\texttt{Dec}}’ and ‘\text{\texttt{Exp}}’, subset declarations representing the alternatives in the BNF declarations, such as the one between ‘\text{\texttt{Exp}}’ and ‘\text{\texttt{Int}}’. The same is done for the automatically generated sorts, such as ‘\text{\texttt{Seq(Env)}}’, sequence for ‘\text{\texttt{Env}}’, and ‘\text{\texttt{Seq(Dec)}}’, the sequence of ‘\text{\texttt{Dec}}’. MSDF function declarations are transformed into operations, such as ‘\text{\texttt{\_\_\_\_\_\_}}’ and ‘\text{\texttt{\_\_\_\_\_\_}}’. Operators for MRS configurations (e.g. \(\langle \text{\texttt{\_\_\_\_\_\_}} : \text{\texttt{Dec Record}} \rightarrow \text{\texttt{Conf}} \rangle\)), typed syntax-trees (e.g. \(\langle \text{\texttt{\_\_\_\_\_\_}} : \text{\texttt{Dec Qid}} \rightarrow \text{\texttt{Dec}} \rangle\)), field operators (e.g. \(\langle \text{\texttt{\_\_\_\_\_\_}} \langle \text{\texttt{\_\_\_\_\_\_}} : \text{\texttt{Dec Record}} \rightarrow \text{\texttt{Conf}} \rangle\)) and ‘\text{\texttt{step}}’ constructors (e.g. \(\langle \text{\texttt{\_\_\_\_\_\_}} \langle \text{\texttt{\_\_\_\_\_\_}} : \text{\texttt{Dec Record}} \rightarrow \text{\texttt{Conf}} \rangle\)) are also declared. An example of a ‘\text{\texttt{step}}’ rule is the following one, for ‘\text{\texttt{Dec}}’.

\[
crl \langle P@:\text{\texttt{Dec}} : \cdots : \text{\texttt{Dec, R:Record}} \rangle \Rightarrow \langle P'@:\text{\texttt{Dec}} : \cdots : \text{\texttt{Dec, R':Record}} \rangle \quad [\text{label step}]
\]

\[
\text{if } \{P@:\text{\texttt{Dec}} : \cdots : \text{\texttt{Dec, R:Record}}\} \Rightarrow [P'@:\text{\texttt{Dec}} : \cdots : \text{\texttt{Dec, R':Record}}] \quad [\text{label step}]
\]

\textbf{Compilation of label declarations.} MSDF label declarations are transformed into an equational theory that extends the \text{\texttt{RECORD}} theory. To avoid preregularity problems, ‘\text{\texttt{Component}}’ is not subsorted and for each index-semantic component pair in a MSDF label declaration, a ‘\text{\texttt{Field}}’ operator (‘\text{\texttt{\_\_\_\_\_\_}}’) and ‘\text{\texttt{duplicated}}’ equations are declared on \text{\texttt{RECORD}} configurations.

A word about how MSOS labels and records are represented algebraically is needed: MSOS labels are defined as purely abstract sorts ‘\text{\texttt{Label}}’, which represents an entire label; ‘\text{\texttt{ILabel}}’, which represents an entire, unobservable label; ‘\text{\texttt{FieldSet}}’, which represents a subset of the fields of a label, and ‘\text{\texttt{IFieldSet}}’, that represents an unobservable subset of the fields of a label. The equivalent MRS sorts are, respectively, ‘\text{\texttt{Record}}’, ‘\text{\texttt{PreRecord}}’, ‘\text{\texttt{IRecord}}’, and ‘\text{\texttt{IPreRecord}}’.

Additional subsorts of ‘\text{\texttt{Field}}’ and ‘\text{\texttt{Index}}’ are also defined as they are necessary for the compilation of the transition rules. For read-only fields, the sort ‘\text{\texttt{ROField}}’ is used; for read-write fields, the sort ‘\text{\texttt{RWField}}’ is used; and for write-only fields, the sort ‘\text{\texttt{WOField}}’ is used. For the indices, the following sorts are defined: for read-only indices, the sort ‘\text{\texttt{RO-Index}}’ is used; for read-write indices, both sorts ‘\text{\texttt{Pre-RW-Index}}’ and ‘\text{\texttt{Post-RW-Index}}’ are defined related to the unprimed and primed indices, respectively; and finally for write-only indices, the sort ‘\text{\texttt{WO-Index}}’ is used.

For the LET specification the read-only index ‘\text{\texttt{env}}’ is declared, a membership equation allowing elements of sort ‘\text{\texttt{Env}}’ to be part of the record structure is
generated, and an equation that checks for duplicated indices is declared.

\[
\begin{align*}
\text{op} & \text{ env : } \rightarrow \text{RO-Index} \quad \text{[ctor]} . \\
\text{mb} & \text{ env = V0:Env : ROField} . \\
\text{eq} & \ (I:\text{RO-Index} = C:\text{Env}, PR1: \text{PreRecord}) ; (I:\text{RO-Index} = C:\text{Env}, PR2: \text{PreRecord}) \\
& = I:\text{RO-Index} = C:\text{Env}, PR1: \text{PreRecord} ; PR2: \text{PreRecord} . \\
\text{eq} & \ \text{duplicated}(I: \text{Index} = C@:\text{Env}, I: \text{Index} = C'@:\text{Env}, PR: \text{PreRecord}) \\
& = \text{true} .
\end{align*}
\]

**Compilation of transitions.** Transition rules are compiled into conditional
rewrite rules. Essentially, the compilation process deals with the fact that a relation
between three elements—the two syntactic trees and the label—is converted into
a relation between MRS configurations, that is, tuples containing the syntactic
tree and the MRS record.

Dealing with the syntactic trees is straightforward: the left-hand side of MSDF
transitions become the first projection on the left-hand side configuration, the
program part, and the same idea applies to the right-hand side. Recall that,
in MSDF, the syntactic trees have an associated type; this typed syntactic tree
is converted to tuples of syntactic trees and types constructed by the \( _:::_ \)
operator, as explained in Section 4.1. MSOS label expressions in transition rules
are compiled following the pre and post projections as explained in Section 3.

The transition rule for evolving `let` expressions, from Section 4.1, is trans-
morphed into the following conditional rewrite rule. Note that the ellipsis `...`
are represented directly as variables, and that a fresh variable `envVAR0:Env` is declared
for the environment in right-hand side of the rewrite in the condition. Such
variable declaration allows for expressions not in normal-form, such as `Env1:Env
/ Env2:Env`, to be used in the left-hand side.

\[
\begin{align*}
\text{crl} \ \{ \text{let Env1:Env in Exp:Exp :} & \text{ :: Exp,\{env = Env2:Env, ...:PreRecord\}} \\
& \Rightarrow [\text{let Env1:Env in Exp':Exp :} \text{ :: Exp,\{env = Env2:Env, ...:PreRecord\}}] \\
& \quad \text{if} \ \{ \text{Exp:Exp :} & \text{ :: Exp,\{env = (Env1:Env / Env2:Env), ...:PreRecord\}} \Rightarrow \\
& \quad \quad [\text{Exp':Exp :} & \text{ :: Exp,\{env = envVAR0:Env, ...:PreRecord\}}] \ [\text{label none}] .
\end{align*}
\]

5 MMT in Practice

This Section outlines examples created to assess the capabilities of MMT. The
complete descriptions can be found in [3]. The first example is the MSDF speci-
fication of Mosses’s Constructive MSOS where the semantics of a programming
language is expressed in terms of basic, abstract, constructs. The MSDF specifi-
cation of Constructive MSOS was further used to define the semantics of a subset
of Reppy’s Concurrent ML (CML) and Appel’s MiniJava. The second example is
the MSDF specification of Mini-Freja, a normal-order language [15]. The third
element is the specification and verification of CCS. The fourth example is the
specification and verification of many distributed algorithms.

Constructive MSOS is an abstract syntax for usual programming languages
constructs. Our MSDF specifications for Constructive MSOS consist of 800 lines,
divided into 74 modules that define abstract constructs and basic data types
commonly found in programming languages. (For instance, there is a module Cons/Abs/closure with the BNF and transition rules for closure values.) The fine-grained modularization of the specification is only possible due to encapsulation of the semantic components provided by MSOS, a modularization that greatly improves reusability.

We wrote two specifications using Constructive MSOS to exemplify its reusability. The semantics of two languages, MiniJava and a subset of CML were developed based on a translation from their concrete syntax into Constructive MSOS abstract constructs. The semantics of a complete programming language, such as MiniJava, is given by a MSDF module that gathers all necessary constructs and defines an initial environment for the beginning set of bindings. The conversion from MiniJava to the abstract constructs is performed, in our implementation, by the SableCC parser generator. (An external parser was used for a fine-grained lexical control during the parsing process.)

The semantics for MiniFreja was created as an example of an MSDF specification of a lazy evaluation functional language, specified in big-step operational semantics. We did not use external parsers, or libraries, followed [15] straightforwardly, and implemented the complete pattern matching algorithm specified in [15]. To test the specification we implemented and executed the sieve of Eratosthenes using “lazy lists,” as in the original MiniFreja specification.

It is well known that SOS is a formalism not only used in the specification of programming languages, but also of concurrent systems [10]. The specification and analyses of CCS\textsuperscript{6} was straight-forward, since it was originally specified in operational semantics. We also implemented and analyzed several distributed algorithms: Lamport’s bakery algorithm, some variations of dining philosophers, leader election on an asynchronous ring, Peterson’s solution for mutual exclusion, mutual exclusion with semaphores, and a thread game, where two threads compete for a shared resource. Using MSDF and MMT, the algorithms can be specified independently from any particular scheduling strategy. Such a scheduling strategy may be modularly added later.\textsuperscript{7}

These experiments gave us confidence that MSDF can express quite well MSOS, either in small-step or big-step styles. We managed to write and analyze straight-forwardly non-trivial operational semantics specifications already written in MSOS mathematical style, operational semantics specifications not written in MSOS style, and specify, quite succinctly, several concurrent systems that were not originally specified in operational semantics. Moreover, we analyzed all specifications using Maude’s built-in state search and model-checker within reasonable time frames, according to related literature. However, the tool needs improvement regarding usability. At the moment, it requires some understanding of Maude and how MSDF specifications are represented in Maude in order to execute and analyze MSDF specifications and understand some of the errors reported by MMT.

\textsuperscript{6} The MSDF specification of CCS was developed together with Alberto Verdejo.

\textsuperscript{7} \url{http://www.ic.uff.br/~cbraga/losd/publications/modular-da.pdf
Improving the tool usability is our main goal.

6 Related and Future Work

The relationship between SOS and RWL was first established by Martí Oliet and Meseguer in [7], and further developed by Verdejo in his PhD thesis [17]. In [14] the metarepresentation of GSOS in rewriting logic is used to implement methods for checking the premises of some SOS meta-theorems (e.g., GSOS congruence) in the GSOS framework. By restricting themselves to structural operational semantics, these approaches lack support for modular specifications. Modularity can be achieved in RWL, and not only while representing SOS specifications, using MRS theories, as mentioned in Section 3. Rewriting logic semantics [9] (RLS), developed by Meseguer and Roşu, allows for modular specifications with true concurrency, following a stack machine model and continuations. Both MRS and RLS, however, have readability problems, which appear to be more serious in RLS. The specification of true concurrency and continuations remains an open problem in MSOS and, therefore, in MSDF.

Mosses’s own MSOS Tool [11], implemented using Prolog, is the only alternative to MMT, at the moment, for writing MSOS specifications. The efficiency of the tool needs some improvement and, while it has support for tracing through execution paths via Prolog, it currently does not have the ability to model check specifications or any mechanism to combine it with other tools.

Other significant operational semantics tools include Hartel’s LETOS [6] and Pettersson’s RML [15]. LETOS (Lightweight Execution Tool for Operational Semantics) supports SOS (and denotational semantics) by generating Miranda code, has interesting pretty-printing facilities and support for non-determinism by means of lists. RML (Relational Meta-Language) supports natural semantics specifications generating efficient C code. Neither support any form of modular specifications, search, or model-checking.

Regarding future work, we plan to explore the combination of MMT with user-defined Maude tools, such as Verdejo’s strategy language interpreter (SLI), and Clavel’s inductive theorem prover (ITP). A prototype for a combined tool with SLI, developed together with Verdejo, has been developed, allowing the execution and search on MSDF specifications to be guided by strategies.

Acknowledgement

The authors would like to thank Peter Mosses for his helpful comments on a draft of this paper, and the anonymous referees for their careful review. This work was partially sponsored by CNPq.
References


Abstract
The rewriting calculus has been introduced as a general formalism that uniformly integrates rewriting and λ-calculus. In this calculus all the basic ingredients of rewriting such as rewrite rules, rule applications and results are first-class objects. The rewriting calculus has been originally designed and used for expressing the semantics of rule based as well as object oriented paradigms. We have previously shown that convergent term rewriting systems and classic strategies can be encoded naturally in the calculus.

In this paper, we go a step further and we propose an extended version of the calculus that allows one to encode unrestricted term rewriting systems. This version of the calculus features a new evaluation rule describing the behavior of the result structures and a call-by-value evaluation strategy. We prove the confluence of the obtained calculus and the correctness and completeness of the proposed encoding.

Keywords: rewriting calculus, lambda calculus, term rewriting systems, fixpoints.

1 Introduction
The ability to discriminate patterns is one of the main basic mechanisms the human reasoning is based on. Indeed, the ability to recognize patterns, i.e. pattern matching, is present since the beginning of information processing modeling. Instances of it can be traced back to pattern recognition and it has been extensively studied when dealing with strings [11], trees [9] or feature objects [1].

Pattern matching has also been widely used in functional programming (e.g. ML, Haskell, Scheme), logic programming (e.g. Prolog), rewrite based programming (e.g. ASF+SDF [14], ELAN [2], Maude [13], Obj* [8]), script programming (e.g. sed, awk). It has been generally considered as a convenient mechanism for expressing complex requirements about the argument of a function, more than a real computation paradigm.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
The rewriting calculus [5,7] by unifying λ-calculus and rewriting, makes all the basic ingredients of rewriting explicit objects, in particular the notions of rule application and result. Its basic idea is to abstract on patterns instead of simple variables as in the λ-calculus, and then to produce terms such as \( f(x) \rightarrow x \), that could be represented in a λ-style as \( \lambda f(x).x \).

The rewriting calculus has been originally designed and used for expressing the semantics of rule based as well as object oriented paradigms [6]. Indeed, in rewriting calculus the term rewriting system (TRS) consisting of the rules \( a \rightarrow b \) and \( b \rightarrow c \) can be represented by the structure \( a \rightarrow b \leftrightarrow b \rightarrow c \) and its application to the constant \( a \) is encoded by the term \( (a \rightarrow b \leftrightarrow b \rightarrow c) \) \( a \), i.e. the application of the structure to the argument. This latter term reduces in the rewriting calculus to \( b \). If we consider the structure \( a \rightarrow b \leftrightarrow a \rightarrow c \) consisting of two rules with overlapping left-hand sides, the application \( (a \rightarrow b \leftrightarrow a \rightarrow c) \) \( a \) evaluates to the structure \( b \leftrightarrow c \) that can be seen as the non-deterministic choice between the two terms \( b \) and \( c \).

General term rewriting systems and classical guiding strategies have been encoded in the original rewriting calculus [5] by adding an additional operator that intuitively selects one of the elements from a set of results. We have shown that an equivalent operator can be encoded in the current version of the calculus but the encoding is limited in this case to convergent term rewriting systems [7].

We show in this paper that the previously proposed encoding can be extended to the general case, i.e. to arbitrary term rewrite systems. For this, a new evaluation rule that enriches the semantics of the structure operator is added and an evaluation strategy is enforced by imposing a certain discipline on the application of the evaluation rules. This strategy is defined syntactically using an appropriate notion of value and is used in order to recover the confluence of the calculus that is lost in the general case.

Roadmap In Section 2, we give the syntax and the evaluation semantics of the proposed calculus and we prove its confluence. Then in Section 3, we discuss the expressive power of the calculus. More precisely we propose an encoding of (non-convergent) term rewriting systems in the calculus. Finally in Section 4, we conclude and give some perspectives of this work.

2 The distributive \( \rho \)-calculus: \( \rho_d \)-calculus

We present here the syntax and the semantics of the proposed calculus as well as its main properties.

2.1 Syntax

We consider in what follows the meta-symbols “\( \_ \rightarrow \_ \)” (abstraction operator), and “\( \_ \leftrightarrow \_ \)” (structure operator), and the (hidden) application operator. We assume that the application operator associates to the left, while the other
operators associate to the right. The priority of the application is higher than that of "\( \_ \rightarrow \_ \)" which is, in turn, of higher priority than the "\( \_ \rightarrow \_ \)". The symbols \( A, B, C, \ldots \) range over the set \( \mathcal{T} \) of terms, the symbols \( x, y, z, \ldots \) range over the set \( \mathcal{X} \) of variables \( (\mathcal{X} \subseteq \mathcal{T}) \), the symbols \( a, b, c, \ldots, f, g, h \) and string built from them range over a set \( \mathcal{K} \) of term constants \( (\mathcal{K} \subseteq \mathcal{T}) \). Finally, the symbols \( P, Q \) range over the set \( \mathcal{P} \) of patterns, \( (\mathcal{X} \subseteq \mathcal{P} \subseteq \mathcal{T}) \). All symbols can be indexed. The symbol \( \text{stk} \) is a special constant denoting matching failures and whose semantics will be given in the next section. To denote a tuple of terms \( A_1 \ldots A_n \), we will use the vector notation \( \overrightarrow{A} \). This notation will be used in combination with the application operator : \( \overrightarrow{AB} \) means \(((\overrightarrow{AB_1})\ldots)B_n\).

The syntax of the basic rewriting calculus is inductively defined as follows:

\[
\begin{align*}
\mathcal{P} & ::= \mathcal{X} | \mathcal{K} | \mathcal{K} \mathcal{P} | \text{stk} & \text{Patterns} \\
\mathcal{T} & ::= \mathcal{X} | \mathcal{K} | \mathcal{P} \rightarrow \mathcal{T} | \mathcal{T} \mathcal{T} | \mathcal{T} \triangleleft \mathcal{T} | \text{stk} & \text{Terms}
\end{align*}
\]

We call \textit{algebraic} the patterns used in this version of the calculus and we usually denote a term of the form \( \ldots ((f \ A_1) \ A_2) \ldots) \ A_n \) with \( f \in \mathcal{K} \) by \( f(A_1, A_2, \ldots, A_n) \). A \textit{linear} pattern is a pattern where every variable occurs at most once.

The \textit{values} represent intuitively the terms that we do not need to evaluate and are inductively defined by:

\[
\mathcal{V} ::= \mathcal{X} | \mathcal{K} | \mathcal{K} \mathcal{V} | \mathcal{V} \rightarrow \mathcal{T} & \text{Values}
\]

These values can be extended to the so-called \textit{structure values} and \textit{stuck values}, which will restrict the applications of the evaluation rules \((\gamma), (\rho)\) and \((\delta)\):

\[
\begin{align*}
\mathcal{V}_\gamma & ::= \mathcal{V} | \mathcal{V}_\gamma \triangleleft \mathcal{V}_\gamma & \text{Structure Values} \\
\mathcal{V}_{\rho\delta} & ::= \mathcal{V} | \text{stk} & \text{Stuck Values}
\end{align*}
\]

One can notice that the only potential redexes (\textit{i.e.} applications of variables, abstractions or structures) in values are inside abstractions. In what follows the symbol \( V \) ranges over the set \( \mathcal{V} \) of values, the symbol \( V^\gamma \) ranges over the set \( \mathcal{V}_\gamma \) of structure values, the symbol \( V^{\rho\delta} \) ranges over the set \( \mathcal{V}_{\rho\delta} \) of stuck values. All these symbols can be indexed.

\begin{definition}[Free and bound variables] Given a term \( A \), the sets of its free variables denoted \( \mathcal{FV}(A) \) and bound variables denoted \( \mathcal{BV}(A) \) are defined as follows:

\[
\begin{align*}
\mathcal{FV}(x) & \triangleq \{x\} & \mathcal{BV}(x) = \emptyset \\
\mathcal{FV}(f) & \triangleq \emptyset & \mathcal{BV}(f) = \emptyset \\
\mathcal{FV}(P \rightarrow A) & \triangleq \mathcal{FV}(A) \setminus \mathcal{FV}(P) & \mathcal{BV}(P \rightarrow A) = \mathcal{BV}(A) \cup \mathcal{FV}(P)
\end{align*}
\]
\end{definition}
\[ \mathcal{FV}(A B) \triangleq \mathcal{FV}(A) \cup \mathcal{FV}(B) \]
\[ \mathcal{FV}(A \upharpoonright B) \triangleq \mathcal{FV}(A) \cup \mathcal{FV}(B) \]
\[ \mathcal{FV}(\text{stk}) \triangleq \emptyset \]
\[ \mathcal{BV}(A B) = \mathcal{BV}(A) \cup \mathcal{BV}(B) \]
\[ \mathcal{BV}(A \upharpoonright B) = \mathcal{BV}(A) \cup \mathcal{BV}(B) \]
\[ \mathcal{BV}(\text{stk}) = \emptyset \]

As usual, we work modulo “\(\alpha\)-conversion” and adopt Barendregt’s “hygiene-convention”, i.e. free and bound variables have different names.

**Definition 2.2 (Substitutions)**

A substitution \(\theta\) is a mapping from the set of variables to the set of terms. A finite substitution \(\theta\) has the form \(\{A_1/x_1 \ldots A_m/x_m\}\), and its domain \(\{x_1, \ldots, x_m\}\) is denoted by \(\text{Dom}(\theta)\). The application of a substitution \(\theta\) to a term \(A\) such that \(\text{Dom}(\theta) \cap \mathcal{BV}(A) = \emptyset\), denoted by \(A\theta\), is defined as follows:

\[
x_i\theta \triangleq \begin{cases} 
A_i & \text{if } x_i \in \text{Dom}(\theta) \\
 x_i & \text{otherwise}
\end{cases}
\]

\[
(P \rightarrow A)\theta \triangleq P \rightarrow A\theta
\]

\[
(A B)\theta \triangleq A\theta \; B\theta
\]

\[
(A \upharpoonright B)\theta \triangleq A\theta \; \upharpoonright \; B\theta
\]

\[
\text{stk}\theta \triangleq \text{stk}
\]

We should point out that since we consider classes of terms modulo the \(\alpha\)-conversion, any term \(A\) has a proper representative \(A'\) such that \(\mathcal{BV}(A') \cap \text{Dom}(\theta) = \emptyset\), which avoids potential variable captures.

### 2.2 Operational semantics

The evaluation mechanism of the rewriting calculus relies on the fundamental operation of matching that allows us to bind variables to their current values. In the general rewriting calculus we allow the matching to be performed modulo a congruence on terms. This congruence used at matching time is a fundamental parameter of the calculus and different instances are obtained when instantiating this parameter by a congruence defined, for example, syntactically, or equationally or in a more elaborated way [6].

For the purpose of this paper we restrict to syntactic matching, in which case the matching substitution, when it exists, is unique and can be computed by a simple recursive algorithm given for example by G. Huet [10].

The operational semantics of the \(\rho_d\)-calculus is defined by the following rules:

\[
(P \rightarrow A) \; V^{\rho_d} \rightarrow_{\rho} A\theta \quad \text{if} \; \; P\theta \equiv V^{\rho_d}
\]

\[
(A \upharpoonright B) \; V^{\rho_d} \rightarrow_{\delta} A \; V^{\rho_d} \upharpoonright B \; V^{\rho_d}
\]

\[
A \; (V_1^{\gamma} \upharpoonright V_2^{\gamma}) \rightarrow_{\gamma} A \; V_1^{\gamma} \upharpoonright A \; V_2^{\gamma}
\]
The rule \((\rho)\) can be applied if (and only if) such a substitution \(\theta\) exists and in this case it is applied to the term \(A\). If such a substitution does not exist then this rule can not be fired and the term is left as it is, representing a failure. Nevertheless, further reductions or instantiations are likely to modify \(V^{\rho\delta}\) so that the appropriate substitution can be found and the rule can be fired. The rule \((\delta)\) right-distributes the application over the structures. This gives the possibility, for example, to apply in parallel two distinct pattern abstractions to a given term. The rule \((\gamma)\) is the counterpart of the rule \((\delta)\) and left-distributes the application of a term over a structure. The implicit conditions imposing that the arguments of an application are values are essentially related to the confluence of the calculus and are discussed in Section 2.3.

**Definition 2.3 (One-step relation)**

The one-step relation induced by a set of rewrite rules \(\mathcal{R}\) is noted \(\rightarrow_{\mathcal{R}}\) and is the compatible closure of the relation induced by the set of rules \(\mathcal{R}\):

- if \(t \rightarrow_{\mathcal{R}} u\) then \(t \rightarrow_{\mathcal{R}} u\);
- if \(t \rightarrow_{\mathcal{R}} u\) then \(f(t_1, \ldots, t, \ldots, t_n) \rightarrow_{\mathcal{R}} f(t_1, \ldots, u, \ldots, t_n)\).

The multi-step relation, denoted \(\rightarrow_{\mathcal{R}}\), is the reflexive and transitive closure of \(\rightarrow_{\mathcal{R}}\). Similarly, the multi-step relation induced by the rules of the \(\rho\delta\)-calculus is denoted \(\rightarrow_{\rho\delta}\), with the compatible closure defined as follows.

**Definition 2.4 (Compatible closure of \(\rightarrow_{\rho\delta}\))**

In the distributive \(\rho\)-calculus, a context is a special term defined by the following grammar:

\[
C[\ ] ::= [\ ] | \mathcal{P} \rightarrow C[\ ] \mid T \ C[\ ] \mid C[\ ] \ T \mid C[\ ] \ \triangleright \ T \mid T \ \triangleright \ C[\ ]
\]

The compatible closure of \(\rightarrow_{\rho\delta}\) is the (finest) relation \(\rightarrow_{\rho\delta}\) such that if \(t \rightarrow_{\rho\delta} u\), then for any context \(C[\ ]\), we have \(C[t] \rightarrow_{\rho\delta} C[u]\).

**Example 2.5 (Simple example + failures)**

If we consider the terms \((f(x) \rightarrow (3 \rightarrow 3) x) \ f(3)\) and \((f(x) \rightarrow (3 \rightarrow 3) x) \ f(4)\) then the following reductions are obtained:

\[(f(x) \rightarrow (3 \rightarrow 3) x) \ f(3) \rightarrow_{\rho} (3 \rightarrow 3) 3 \rightarrow_{\rho} 3\]

\[(f(x) \rightarrow (3 \rightarrow 3) x) \ f(4) \rightarrow_{\rho} (3 \rightarrow 3) 4\]

The term \((a \rightarrow b \triangleright a \rightarrow c)\) does not reduce to \(b \triangleright c\):

\[(a \rightarrow b \triangleright a \rightarrow c) \ a \rightarrow_{\delta} (a \rightarrow b) a \triangleright (a \rightarrow c) a \rightarrow_{\rho} b \triangleright c\]

The term \((a \rightarrow b \triangleright b \rightarrow c)\) reduces similarly to \(b \triangleright (b \rightarrow c)\).

Notice that the term \((a \rightarrow b \triangleright b \rightarrow c)\) does not reduce to \(b\) as one might expect. Instead, the fact that the rule \(b \rightarrow c\) fails to apply to \(a\) (in classical rewriting) is also recorded in the final result as a (failure) term in normal form. This approach is very interesting when we want to handle explicitly the failures by allowing rules that can handle such particular terms (e.g., for an exception handling mechanism). However, if the user is not interested in
the explicit manipulation of matching failures and just wants to ignore such a behavior, we need to handle uniformly matching failures and eliminate them when not significant for the computation.

For this, we first want to represent all the definitive failures by the constant \( 	ext{stk} \) whose exact semantics should be the following: if for any reduction of the argument, there exist no matching substitution, then the \( \rho \)-redex is reduced to \( \text{stk} \):

\[
\forall \theta_1, \theta_2, \forall B', \ B\theta_1 \mapsto_{\rho, \delta, \gamma} B' \Rightarrow P\theta_2 \neq B' \\
(P \rightarrow A) B \rightarrow_{\text{stk}} \text{stk}
\]

One can easily notice that \( B \) can contain a \( \rho \)-term with an arbitrary (possibly infinite) number of possible reductions which should be all explored in order to decide if the appropriate substitution exists. The condition of this rule is thus undecidable and consequently the operational semantics of the calculus cannot be defined using such a rule. Nevertheless, in practice and particularly when dealing with term rewriting systems we do not need to be so general and a sufficient condition can be used.

**Definition 2.6 (Definitive failures)**

The relation \( \nsubseteq \) on \( P \times T \) is inductively defined by:

\[
\begin{align*}
\text{stk} & \nsubseteq g B & \text{if } g \neq \text{stk} \\
\text{stk} & \nsubseteq Q \rightarrow B \\
f P_1 \ldots P_m & \nsubseteq g B_1 \ldots B_n & \text{if } f \neq g \text{ or } n \neq m \text{ or } \exists i, \ P_i \nsubseteq B_i \\
f P & \nsubseteq \text{stk} \\
f \overline{P} & \nsubseteq Q \rightarrow B
\end{align*}
\]

Starting from this relation, the operational semantics of the \( \rho_{\text{stk}} \)-calculus are defined by the rules (\( \rho \)), (\( \delta \)), (\( \gamma \)) introduced above and by the following rules:

\[
(P \rightarrow A) B \rightarrow_{\text{stk}} \text{stk} \quad \text{if } P \nsubseteq B
\]

\[
\text{stk} \downarrow A \rightarrow_{\text{stk}} A
\]

\[
A \downarrow \text{stk} \rightarrow_{\text{stk}} A
\]

\[
\text{stk} A \rightarrow_{\text{stk}} \text{stk}
\]

As mentioned previously, these rules are used to determine, propagate or eliminate the definitive failures. If the matching between the left-hand side of a rule and the argument the rule is applied on is definitive then the failure is made explicit by transforming the application into a \( \text{stk} \); this is done by the first rule. Structures can be seen as collections of results and thus we want to eliminate all the (matching) failures from these collections; this is done by the
next two rules. On the other hand, a stk term can be seen as an empty set of results; the last rule corresponds then to the \((\delta)\) rule dealing with empty structures and thus, to a propagation of the failure. We will see in Section 3 why the stk-rule corresponding to the \((\gamma)\) rule is not suitable.

The \(\longrightarrow_{\text{stk}}\) induced relations are denoted \(\mapsto_{\text{stk}}\) and \(\mapsto_{\text{stk}}\). The relation \(\mapsto_{\rho\delta} \cup \mapsto_{\text{stk}}\) is denoted \(\mapsto_{\rho\gamma\delta}\) and its transitive and reflexive closure is denoted \(\mapsto_{\rho\gamma\delta}^{\text{tr-rf}}\).

**Example 2.7 (failures)**

The term \((a \rightarrow b \leftarrow b \rightarrow c)\) reduces now to \(b\):

\[
(a \rightarrow b \leftarrow b \rightarrow c) \; a \xrightarrow{\delta} (a \rightarrow b) \; a \leftarrow (b \rightarrow c) \; a \xrightarrow{\rho} b \leftarrow (b \rightarrow c) \; a
\]

\[
\mapsto_{\text{stk}} \; b \leftarrow \text{stk} \xrightarrow{\text{stk}} \; b
\]

2.3 Properties

As we have mentioned in the previous section the \(\rho_d\)-calculus would not be confluent if we did not restrict the application of an abstraction and of a structure to be effective only when the argument is a value. When this restriction is not imposed on the \((\rho)\) rule, potentially non-joinable critical pairs between the rules \((\rho)\) and \((\gamma)\) are obtained. Intuitively, restricting the argument of the application in the rule \((\rho)\) to a value guarantees that it has been reduced enough to check if there exists a unique match between the pattern and the argument. Alternatively, we can accept any term as argument and use a more complex matching algorithm to find the appropriate substitution.

**Example 2.8 (\(\rho\) without values)**

When the conditions on values in the rule \((\rho)\) are omitted, non-confluent reductions can be obtained:

\[
\begin{array}{c}
(x \rightarrow f(x, x)) \; a \leftarrow b \\
\downarrow \rho \\
f(a, a \leftarrow b) \leftarrow f(b, a \leftarrow b) \\
\downarrow \gamma \\
f(a, a \leftarrow b) \leftarrow f(b, a \leftarrow b) \\
\downarrow \rho, \gamma \\
f(a, a) \leftarrow f(b, a) \leftarrow f(b, b)
\end{array}
\]

Similarly, when the argument of the application is not restricted to a value in the \((\delta)\) rule, a critical pair between the rules \((\delta)\) and \((\gamma)\) is obtained. The confluence can be retrieved either by enforcing this condition or by using an associative-commutative underlying theory for the structure operator.

**Example 2.9 (\(\delta\) without values)**

When no conditions are imposed in the rule \((\delta)\) non-confluent reductions can
be obtained:

\[
\begin{align*}
(a \land b) (c \land d) & \quad \gamma \\
(a \land b) c \land (a \land b) d & \quad \delta \\
(a \land d) \land b (c \land d) & \quad \delta, \delta \\
(a \land b) c \land a d \land b d & \quad \gamma, \gamma \\
ac \land b c \land a d \land b d & \quad \delta, \delta \\
ac \land a d \land b c \land b d & \quad \delta, \delta
\end{align*}
\]

For the \((\gamma)\) rule, the condition imposes that the terms in the structure do not reduce to a failure. If one of them can lead to a failure then it should be first reduced to \(stk\) and then eliminated from the structure using the \(stk\) rules.

**Example 2.10 \((\gamma)\) without values**

If the terms of a structure applied to an argument are not restricted to values then the application of the rule \((\gamma)\) can lead to non-confluent reductions:

\[
\begin{align*}
(x \to f(x)) (stk \land a) & \quad \gamma \\
(x \to f(x)) stk \land (x \to f(x)) a & \quad stk \\
(x \to f(x)) a & \quad \rho, \rho \\
f(stk) \land f(a) & \quad \rho
\end{align*}
\]

It is quite clear that using a set of values leads to a call-by-value reduction strategy. The two calculi presented above are confluent in this case.

**Theorem 2.11 (Confluence of left-linear \(\rho_d\)-calculus)**

If all patterns are linear, the relation \(\to_{\rho_d}\) is confluent.

**Proof.** The proof is detailed in [4]. It uses the parallel reduction technique introduced for the \(\lambda\)-calculus in [12].

**Theorem 2.12 (Confluence of left-linear \(\rho_{stk}^d\)-calculus)**

If all patterns are linear, the relation \(\to_{\rho_{stk}^d}\) is confluent.

**Proof.** The proof is detailed in [4]. It is based on the proof introduced in [7, 16].

The unrestricted \(\rho_d\)-calculus is non-confluent since the Klop counter example holds in this case (see [4]).

### 2.4 \(\rho_d\)-calculus modulo some congruence

Defining a similar calculus modulo some congruence is certainly interesting but out of the scope of this paper. Such an extension would induce an encoding
of general term rewriting systems just as our present calculus induces the
encoding of syntactic term rewriting systems presented in the next section.

The main difficulty in defining such a calculus comes from the fact that the
matching modulo the given congruence is generally non-unitary (at least for
classical theories like associativity and commutativity). Indeed there might
exist (infinitely) many solutions and there exists no natural ordering for these
solutions (i.e. substitutions).

For example let us consider the term $(f(x, y) \rightarrow x) f(a, b)$ when working
modulo the commutativity of the symbol $f$. There exists two solutions of
the matching problem: $\{a/x, b/y\}$ and $\{b/x, a/y\}$. Depending on the used
substitution we obtain two possible non-confluent reductions:

$$\begin{align*}
(f(x, y) \rightarrow x) f(a, b) &\xrightarrow{C} (f(y, x) \rightarrow y) f(a, b) \\
&\xrightarrow{\rho} a \bowtie b & b \bowtie a
\end{align*}$$

In order to recover the confluence the only solution may finally consist in
declaring the structure operator as commutative, associative and idempotent.

Moreover, the reduction strategy should be enforced when working modulo
some congruence. The strategy should prevent the matching against unin-
stanciated terms that would lead to non-confluent reductions as shown in the
following example where the symbol “::” is considered associative:

$$\begin{align*}
(z \rightarrow ((x :: y \rightarrow x) (a :: z))) (b :: c) \\
&\xrightarrow{\rho} (x :: y \rightarrow x) (a :: (b :: c)) & (z \rightarrow a) (b :: c) \\
&\xrightarrow{\rho} a \bowtie (a :: b) & a \bowtie (a :: b)
\end{align*}$$

Finally the notion of definitive failures should be adapted to the reduction
strategy of the new calculus in order to guarantee the coherence with the
considered matching.

## 3 Encoding Term Rewriting Systems

We have already shown [6,16] that (the reduction of) convergent term rewriting
systems (TRS) can be encoded in the classical rewriting calculus. The restric-
tion to convergent TRS is due to the “uncomplete” treatment of the structure
operator in the classical rewriting calculus where the application operator is
left-distributive over the structure operator but not right-distributive. As we
have already seen this choice was motivated by the meta-properties the cal-
culus should have. More precisely, adding right-distributivity would lead to a
non-confluent calculus. Nevertheless, this property can be retrieved either by
enforcing a certain discipline on the evaluation (strategy) [5] or by restricting
the term formation as done in this paper.

26
In $\rho_{d_{stk}}$-calculus the ($\gamma$) rule defines the right-distributivity of the application over the structure and in this section we show how this feature can be used to encode (non-confluent) TRS in the $\rho_{d_{stk}}$-calculus.

More precisely, given a TRS $\mathcal{R}$ we build the terms $\Omega_{1}$ and $\Omega$ such that

- $\Omega_{R} m$ represents (i.e. reduces to) the one-step reduced of $m$ w.r.t. $\mathcal{R}$,
- $\Omega_{R} m$ represents the normal form of $m$ w.r.t. $\mathcal{R}$ (if it exists).

### 3.1 Rule selection

As we wish to compute the normal forms, we obviously wish to decide when the reduction is effective, i.e. when some rule of $\mathcal{R}$ can be applied, and then to discriminate cases:

- if some rule of $\mathcal{R}$ can be applied to $m$, then we reduce $m$,
- if not, $m$ is a normal form, and $m$ is left as it is.

This ability to discriminate cases, i.e. to select between two (or more) terms which one can be applied successfully to a given argument, is encoded in the first term usually defined \[7\] in the rewriting calculus by:

$$\text{first} \triangleq u \rightarrow v \rightarrow x \rightarrow (\text{stk} \rightarrow v x \mid y \rightarrow y) (u x)$$

One can easily check that first has the intended behavior:

- $(\text{first} A_{1} A_{2}) t \rightarrow_{\rho_{d_{stk}}} V_{1}^{\rho_{d}}$ if $A_{1} t \rightarrow_{\rho_{d_{stk}}} V_{1}^{\rho_{d}}$;
- $(\text{first} A_{1} A_{2}) t \rightarrow_{\rho_{d_{stk}}} V_{2}^{\rho_{d}}$ if $A_{1} t \rightarrow_{\rho_{d_{stk}}} \text{stk}$ and $A_{2} t \rightarrow_{\rho_{d_{stk}}} V_{2}^{\rho_{d}}$.

Intuitively, if we replace the term $A_{1}$ by the $\rho$-term $R$ encoding a TRS $\mathcal{R}$ and the term $A_{2}$ by the identity then we obtain the desired discrimination facility: the case $R t \rightarrow_{\rho_{d_{stk}}} V_{1}^{\rho_{d}}$ corresponds to a reduction of $t$ w.r.t. $\mathcal{R}$ while $R t \rightarrow_{\rho_{d_{stk}}} \text{stk}$ corresponds to the case where no rule can be applied to $t$ and thus the term is left as it is (in fact the identity is applied to this term).

As the normal form of some terms w.r.t. a non-confluent TRS is not unique, we will obviously have to deal with sets of results. We choose here to encode sets of results as structures. The empty set is represented by $\text{stk}$ and the union of two sets is represented using the structure operator. In the rewriting calculus the representation of some set is not unique as the structure operator is not considered as commutative, associative or idempotent.

Since we wish to discriminate cases such as no rule of $\mathcal{R}$ matches $m$, reformulated as the set of one-step reduced of $m$ is empty, we need to pattern match on $\text{stk}$. The statement if the set $M$ is empty, then $T_{1}$ else $T_{2}$ can be encoded by

$$\text{first} (\text{stk} \rightarrow T_{1}) (x \rightarrow T_{2}) M$$

Since we need the ability to pattern match on $\text{stk}$ we have the rule

$$\text{stk} A \rightarrow_{\text{stk}} \text{stk}$$
that complements the (δ) rule but not the symmetric one

\[ A \text{stk} \rightarrow_{\text{stk}} \text{stk} \]

that would complement the (γ) rule and that would correspond to a *strict* propagation of the failure.\(^1\)

### 3.2 Context propagation

When rewriting *w.r.t.* a rewriting system, the application of the rules can be done on any *subterm* of the rewritten term. In the rewriting calculus, a rule is always applied on the head of the term and thus the encoding of a TRS has to propagate explicitly the application deeper in the term. For example, the application of the rewrite rule \( a \rightarrow b \) to the term \( f(x) \) is naively encoded by the term \( (f(x) \rightarrow f((a \rightarrow b) x)) f(a) \) that eventually reduces, as expected, to \( f(b) \).

If the application of a rewrite rule fails on all the subterms of a given term then the \( \rho \)-term encoding the application should be reduced to \( \text{stk} \). On the other hand, if we apply the same naive methodology as above for propagating the rule application into contexts then the application of the rewrite rule \( a \rightarrow b \) to the term \( f(b) \) is encoded by the term \( (f(x) \rightarrow f((a \rightarrow b) x)) f(b) \) that reduces to \( f(\text{stk}) \) and not to \( \text{stk} \).

More generally, the propagation of \( \text{stk} \) should be performed *w.r.t.* to any context. Therefore, for each symbol \( f \) of arity \( n \geq 1 \) from a signature \( \Sigma \) we define a term \( \Gamma^f_k \):

\[
\Gamma^f_k \triangleq \nu \rightarrow \pi^n \rightarrow (\text{nop}(z) \rightarrow z) \quad \text{(first)} \quad \begin{pmatrix}
\text{stk} \rightarrow \text{nop}(
\nu x_k)

\pi^n \ 
\rightarrow 
\text{nop}(f(x_1, \ldots, x_{k-1}, y, \ldots, x_n))
\end{pmatrix}
\]

where \( \text{nop} \notin \Sigma \) and for any \( n \geq 1 \), \( \pi^n \rightarrow M \triangleq x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_n \rightarrow M \).

Each \( \Gamma^f_k \) allows us to express the application of a given term to the subterm \( M_k \) of some term \( f(M_1, \ldots, M_n) \). The following lemma states the behavior of \( \Gamma^f_k \):

**Lemma 3.1** Let \( f \in \Sigma \) be a symbol of arity \( n \). Let \( M_1, \ldots, M_n \) be some algebraic terms and \( T \) an arbitrary term. Let \( V^\gamma_1, \ldots V^\gamma_p \) be some values in \( V_\gamma \). Then

\[
\Gamma^f_k \ T \ M_1 \ldots M_n \rightarrow_{\rho_\gamma}^\text{stk} \begin{pmatrix}
f(M_1, \ldots, M_{k-1}, V^\gamma_1, \ldots M_n) \\
l \ldots \\
l f(M_1, \ldots, M_{k-1}, V^\gamma_p, \ldots, M_n)
\end{pmatrix}
\]

if \( T \ M_k \rightarrow_{\rho_\gamma} \text{stk} \) and \( \ldots V^\gamma_p \)

\[
\Gamma^f_k \ T \ M_1 \ldots M_n \rightarrow_{\rho_\gamma}^\text{stk} \text{stk} \ 	ext{if} \ T \ M_k \rightarrow_{\rho_\gamma} \text{stk}.
\]

\(^1\) This strict behavior can be obviously encoded using the rule \( \text{stk} \rightarrow \text{stk} \).
Proof. The proof of this lemma just consists in checking that the reductions hold. It is presented in [4].

Let us remark that for any patterns \( P_1 \) and \( P_2 \) the term first \( (P_1 \rightarrow \text{stk}) \) \((P_2 \rightarrow M) \) \( N \) will always reduce to the same term as \( (P_2 \rightarrow M) \) \( N \). Indeed the first operator does not check if \( N \) matches \( P_1 \) but if \( (P_1 \rightarrow \text{stk}) \) \( N \) reduces to \( \text{stk} \) which is always the case. Consequently, the term

\[
\nu \rightarrow x^n \rightarrow \left( \text{first} \left( \begin{array}{c}
\text{stk} \rightarrow \text{stk} \\
y \rightarrow f(x_1, \ldots, x_{k-1}, y, \ldots, x_n)
\end{array} \right) \right)(\nu x_k)
\]

does not have the same behavior as \( \Gamma^f_k \). The use of the constant \( \text{nop} \) in this latter term allows us to claim that a reduction to \( \text{stk} \) is equivalent to a pattern matching failure.

We can now define the term \( \Gamma^f \)

\[
\Gamma^f \triangleq \Gamma^f_1 \wr \ldots \wr \Gamma^f_n
\]

that represents intuitively the application of some term to each subterm \( M_k \) of a term \( M = f(M_1, \ldots, M_n) \). The structure grouping together the different results obtained when a term \( T \) is applied to \( M \) is obtained by reducing the \( \rho \)-term \( \Gamma^f T M \):

\[
\Gamma^f T M \rightarrow^\rho M \rightarrow \Gamma^f_1 T M \wr \ldots \wr \Gamma^f_n T M
\]

3.3 One-step reduction

Let us consider now a term rewriting system \( \mathcal{R} = \{l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n\} \). We denote by \( \rightarrow^\mathcal{R} \) the compatible closure of \( \mathcal{R} \), \( \approx^\mathcal{R} \) its transitive and reflexive closure. The multiset of all one-step reducts of a term \( M \) is denoted \( \{T \mid M \rightarrow^\mathcal{R} T\} \) where the arity of some term \( T \) is the number of one-step reductions from \( M \) to \( T \). Finally we write \( M \approx^\mathcal{R} T \) if and only if \( M \rightarrow^\mathcal{R} T \) and there exists no term \( N \) such that \( T \rightarrow^\mathcal{R} N \). The multiset of all normal forms of a term \( M \) w.r.t. \( \mathcal{R} \) is denoted \( \{T \mid M \rightarrow^\mathcal{R}! T\} \) where the arity of some term \( T \) is the number of multi-step reductions from \( M \) to \( T \).

The term that encodes the one-step reduction w.r.t. a term rewrite system \( \mathcal{R} \) is denoted by \( \Omega^1_{\mathcal{R}} \) and defined by

\[
\Omega^1_{\mathcal{R}} \triangleq \omega^1_{\mathcal{R}} \omega^1_{\mathcal{R}}
\]

where

\[
\omega^1_{\mathcal{R}} \triangleq \pi \rightarrow \left( \begin{array}{c}
\ldots \rightarrow l_i \rightarrow r_i \ldots \rightarrow l \\
\ldots \rightarrow f(x_1, \ldots, x_n) \rightarrow \Gamma^f(\pi \pi) x_1 \ldots x_n \ldots
\end{array} \right) \text{ for all } l_i \rightarrow r_i \in \mathcal{R}
\]

for all \( f \) of arity \( n \geq 1 \).

The definition of \( \omega^1_{\mathcal{R}} \) can be cut in two parts: the first one encodes the rewriting at the head position w.r.t. \( \mathcal{R} \) since we only transcript each rule of \( \mathcal{R} \)
by the corresponding rule in the $\rho_d^{stk}$-calculus; the second one uses the terms $\Gamma_f^k$ to express that we also rewrite inside contexts. The term $\Omega_R^1$ completes a
fixpoint by means of the expression $(\pi \pi)$ in order to iterate the use of the $\Gamma_f^k$
and to get down in the term as much as needed.

**Theorem 3.2** Let $M$ be an algebraic term.

- If $\{T \mid M \xrightarrow{\mathcal{R}} T\} = \emptyset$ then
  \[ \Omega_R^1 M \xrightarrow{\mu_{\rho} \pi} \text{stk} \; . \]
- If $\{T \mid M \xrightarrow{\mathcal{R}} T\} \neq \emptyset$ then
  \[ \Omega_R^1 M \xrightarrow{\mu_{\rho} \pi} \text{stk} T_1 \wr \ldots \wr T_p \text{ with } \{T \mid M \xrightarrow{\mathcal{R}} T\} = \{T_1, \ldots, T_p\} . \]

Moreover as the left-linear $\rho_d^{stk}$-calculus is confluent, if $\mathcal{R}$ is left-linear and
since $T_1 \wr \ldots \wr T_p$ and $\text{stk}$ are in normal form, then these are the unique
normal forms of $\Omega_R^1 M$.

**Proof.** The proof of this Theorem is done by induction on the term $M$. It is
presented in [4].

| 3.4 Normal form reduction |

We now define the term that encodes the normal form reduction w.r.t. a term
rewrite system $\mathcal{R}$.

More precisely, we want to define a term $\Omega_R$ such that its application to
some term $M$, $\Omega_R M$ reduces to $M$ if $\Omega_R^1 M$ reduces to $\text{stk}$ ($M$ is a normal form)
and continues applying the term $\Omega_R$ to the result of $\Omega_R^1 M$ if it is different from
$\text{stk}$. We define thus the term

\[ \Omega_R \triangleq \omega_R \omega_R \]

where

\[ \omega_R \triangleq s \rightarrow x \rightarrow \text{first} (\text{stk} \rightarrow x) (z \rightarrow (s s) z) (\Omega_R^1 x) \]

Let us introduce now the relation $\subset -$ that represents intuitively the ob-
servability of some “result” in a structured set of terms.

**Definition 3.3** The relation $\subset -$ is defined inductively by:

- for any term $M$, $M \subset - M$;
- for any terms $M, N_1$ and $N_2$, $M \subset - N_1 \Rightarrow M \subset - N_1 \wr N_2$ and $M \subset - N_2 \wr N_1$.

Using the above relation we can state the correctness and completeness of
the encoding:

**Theorem 3.4** Given two algebraic terms $M$ and $M'$,

\[ M \xrightarrow{\mathcal{R}^1} M' \iff \exists T, \; \Omega_R M \xrightarrow{\mu_{\rho} \pi} \text{stk} T \text{ and } M' \subset - T. \]
Moreover if $\mathcal{R}$ terminates on $M$ then

$$\Omega_{\mathcal{R}} M \mapsto_{\rho_{stk}^{\mathcal{R}}}^{stkh} T_1 \land \ldots \land T_p \text{ with } \{ T \mid M \mapsto_{\mathcal{R}^!} T \} = \{ T_1, \ldots T_p \}.$$  

Moreover as the left-linear $\rho_{stk}^{\mathcal{R}}$-calculus is confluent, if $\mathcal{R}$ is left-linear and since $T_1 \land \ldots \land T_p$ is a normal form, it is the unique normal form of $\Omega_{\mathcal{R}} M$.

**Proof.** The proof of this theorem is done first by induction on the term $M$, and then by induction on the largest length of a reduction w.r.t. $\mathcal{R}$ of $M$ to a normal form. It uses the Theorem 3.2 and it is presented in [4].

This theorem claims that our encoding of some term rewriting system encodes its reductions in the $\rho_{stk}^{\mathcal{R}}$-calculus. Indeed the $\rho_{stk}^{\mathcal{R}}$-calculus computes the finite multiset of normal forms of any term on which the term rewriting system terminates. Moreover if the system is divergent on some term, all reductions are still encoded since the $\rho_{stk}^{\mathcal{R}}$-calculus computes a non-terminating reduction, generating normal forms as in a breadth first search of the reduction tree. All normal forms are computed at some iteration although the computation never stops, and may even never stop generating new normal forms.

**Example 3.5 (Oriented groups)** Let us consider the group theory axioms oriented as follows:

$$\Sigma = \{e(0), \ i(1), \ f(2)\}$$

$$\begin{align*}
\{ f(x, e) &\rightarrow x \\
\ f(e, x) &\rightarrow x \\
\ f(x, i(x)) &\rightarrow e \\
\ f(i(x), x) &\rightarrow e \\
\ f(f(x, y), z) &\rightarrow f(x, f(y, z))
\end{align*}$$

This TRS is non-confluent since

$$f(f(i(i(a)), i(a)), a) \rightarrow f(e, a) \rightarrow a$$

The terms $\omega_{\mathcal{R}}^1, \Omega_{\mathcal{R}}^1, \omega_{\mathcal{R}}$ and $\Omega_{\mathcal{R}}$ are then defined by:
\[
\begin{align*}
\omega_R^1 \overset{\pi}{\rightarrow} & \\
\omega_R \overset{\omega_R^1 \rightarrow \omega_R^1 \omega_R}{\rightarrow} & \\
\end{align*}
\]

Then we have the following reductions in the \( \rho_{\text{stk}} \)-calculus:

**one-step reductions**

\[
\begin{align*}
\Omega_R^1 & \overset{\text{stk} \rightarrow \text{nop}(\text{stk})}{\rightarrow} f(i(i(a)), i(a)) \\
\Omega_R^1 & \overset{\text{stk} \rightarrow \text{nop}(\text{stk})}{\rightarrow} f(i(i(a)), a) \\
\Omega_R^1 & \overset{\text{stk} \rightarrow \text{nop}(\text{stk})}{\rightarrow} f(i(i(a)), e) \\
\Omega_R^1 & \overset{\text{stk} \rightarrow \text{nop}(\text{stk})}{\rightarrow} i(i(a)) \\
\Omega_R^1 & \overset{\text{stk} \rightarrow \text{nop}(\text{stk})}{\rightarrow} a \\
\Omega_R^1 & \overset{\text{stk} \rightarrow \text{nop}(\text{stk})}{\rightarrow} stk \\
\end{align*}
\]

**normal form reduction**

\[
\begin{align*}
\Omega_R & \overset{\text{stk} \rightarrow \text{nop}(\text{stk})}{\rightarrow} f(i(i(a)), i(a)) \\
\end{align*}
\]

This latter reduction expresses well the non-confluent reductions of the term \( f(f(i(i(a)), i(a)), a) \) w.r.t. the TRS \( R \) since the result \( i(i(a)) \) represents the two normal forms.

32
4 Conclusions

We have studied the confluence and the expressive power of a rewriting calculus featuring left-distributivity of the application over the structure, whereas only right-distributivity was available in former versions. The confluence of the calculus, which is endangered by careless distributivity of one operator over another, has been recovered using a call-by-value reduction, and is proved using the usual parallel reduction technique.

Since, in the rewriting calculus, a structure of \( \rho \)-rules can be seen as a naive encoding of a term rewrite system then the right-distributivity rule describes the application of each rewrite rule in the structure to the argument. Moreover, structures can be also used to denote the sets of results obtained as result of such an application and the left-distributivity describes the application of a given rule (or structure of rules) to many distinct arguments in parallel. Thus, we can encode the simultaneous exploration of many reduction paths in a term.

Using the left-distributivity together with some earlier techniques, we obtain a better handling of matching failures, and we are able to faithfully encode the behavior of any term rewriting system, even non-confluent. This allows for many interesting theoretical developments, such as the computation of all the normal forms of a given term, which is needed, for example, for the completion of a term rewriting system.

The extension to general term rewriting systems is considered as the next step of this work. A major difficulty when dealing with matching modulo some congruence consists in the multiplicity of solutions and since these solutions cannot be ordered in any natural way, the structure operator should be then considered as associative, commutative and idempotent. Moreover the notion of definitive failures should be adapted to the considered matching theories and a call-by-value strategy should be enforced to prevent matching against uninstanciated terms and thus to avoid loosing matching solutions.

Related Work.

V. van Oostrom has widely studied the confluence of a \( \lambda \)-calculus with patterns [15], but which does not feature structures. Our encoding of TRS shares some similarities with the one presented by S. Byun et al. [3] that describes an untyped encoding of every strongly separable orthogonal TRS into \( \lambda \)-calculus. However, they need some really strong assumptions on the confluence of the original system.

Acknowledgments.

We would like to thank Claude Kirchner for the useful interactions we had on the topics of this paper and to Germain Faure for his detailed and very useful comments on a preliminary version of this work.
References


A distributed implementation of Mobile Maude

Francisco Durán, a Adrián Riesco, b and Alberto Verdejo b

a ETSI Informática, Universidad de Málaga, Spain. duran@lcc.uma.es
b Facultad de Informática, Universidad Complutense, Madrid, Spain.
{adrian,alberto}@sip.ucm.es

Abstract

We present a new specification/implementation of the mobile agent language Mobile Maude. This new version uses the external sockets provided by Maude since its 2.2 version, thus obtaining a really distributed implementation of the mobile language, where messages and mobile objects now may travel from one machine to another one in a transparent way. We also show how, even though the complexity of the Mobile Maude specification and the use of reflection, we have managed to use the Maude’s model checker to prove properties about mobile agents applications.

Keywords: Mobile agents, Mobile Maude, model checking.

1 Introduction

Mobile Maude is a mobile agent language extending Maude and supporting mobile computation. It was first presented in [3], and a significant application appeared in [4].

Mobile Maude uses reflection to obtain a simple and general declarative mobile language design and makes possible strong assurances of mobile agent behavior. The formal semantics of Mobile Maude is given by a rewrite theory in rewriting logic. Since this specification is executable, it can be used as a prototype of the language, in which mobile agent systems can be simulated. The two key notions are processes and mobile objects. Processes are located computational environments where mobile objects can reside. Mobile objects have their own code, can move between different processes in different locations, and can communicate asynchronously with each other by means of messages. Mobile Maude’s key characteristics include: (1) reflection as a

* Research partially supported by the MCyT Spanish projects MIDAS (TIC 2003-01000) and TIN2005-09405-C02-01.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
way of endowing mobile objects with “higher-order” capabilities; (2) object-orientation and asynchronous message passing; and (3) a simple semantics without loss in the expressive power of application code.

The code of a mobile object is given by (the metarepresentation of) an object-based module—a rewrite theory—and its data is given by a configuration of objects and messages that represent its state. Such configuration is a valid term in the code module, which is used to execute it. Maude configurations become located computational environments where mobile objects can reside. Mobile objects can interact with other ones (possibly in different locations), and can move from one location to another.

In [3], Durán, Eker, Lincoln, and Meseguer first introduced Mobile Maude. In that work, the authors presented a ‘simulator’ of Mobile Maude, an executable Maude specification on top of Maude 1.0.5, in which the system code was written entirely in Maude, and thus locations and processes were encoded as Maude terms. In the same paper, the authors also gave a development plan including two development efforts: a first step in which a single-host executable was implemented, and a second implementation effort focusing on true distributed execution.

The release of Maude 2.0 allowed taking the first step. This implementation effort was completed in a very short time, utilizing the built-in object system, for object/message fairness, just by simplifying and extending the previous specification. This new version was developed by Durán and Verdejo, and used in several examples, one of which was reported in [4].

The present work summarizes our results in the second development effort. The built-in string handling and internet socket module available in Maude 2.2 has allowed us to build a really distributed implementation. The Maude 2.2 socket modules support non-blocking client and server TCP sockets (at the OS level). In this implementation effort, a Mobile Maude server runs on top of a Maude interpreter and performs the following tasks: keeps track of the current locations of mobile objects created on a host, handles change of location messages, reroutes messages to mobile objects, and runs the code of mobile objects by invoking the metalevel. In fact, we have made a quite significant number of changes on Mobile Maude. Processes and locations are no longer part of the specification of Mobile Maude, now we talk about Maude processes—not terms, OS processes, which may be running on different machines—and IP addresses. We have also introduced the notion of root objects as managers of the configurations of mobile objects in the different processes.

We explain below the design of processes and mobile objects and their rewriting semantics, based on a formal specification of Mobile Maude written in Maude.

The fundamental notions of Mobile Maude, namely processes, mobile objects, and messages are introduced in Section 2. In Section 3, we give a flavor of the rewriting semantics of Mobile Maude. Section 4 we discuss on the con-
nections via sockets between the different processes in a distributed configuration; in particular, we introduce Maude sockets, we explain buffered sockets and then introduce a very simple sample architecture. Section 5 presents a Mobile Maude application code example in which we specify the search of the best offer between several distributed alternatives. Section 6 explains how we have used the model checker to check properties on our Mobile Maude specifications. Section 7 wraps this piece of work with some final conclusions.

2 Processes, mobile objects, and messages

The key entities in Mobile Maude are processes and mobile objects. Mobile objects are modeled as distributed objects in the class MobileObject. A distributed configuration is made up of located configurations. Each located configuration is executed in a Maude process. Such processes can therefore be seen as located computational environments inside which mobile objects can reside, execute, and send and receive messages to and from other mobile objects located in different processes. We assume that each located configuration has one (and only one) root object, of class RootObject, which keeps information on the location of the process, on the mobile objects in such a configuration, and on the whereabouts of the mobile objects created in it, which may have moved to other processes. We assume uniqueness of root object names in a distributed configuration.

Mobile objects carry their own internal state and code (rewrite rules) with them, can move from one process to another, and can communicate with each other by asynchronous message passing. The names of root objects range over the sort Loc, and have the form \( l(IP, N) \) with \( IP \) the IP address of the machine in which the process is being executed and \( N \) a number. The names of mobile objects range over the sort Mid and have the form \( o(L, N) \) with \( L \) the name of the root object of the process in which it was created and \( N \) a number. Figure 1 shows several mobile objects in two processes, with (mobile) object \( o(l(IP, 0), 1) \) moving from the process with root object \( l(IP, 0) \) to the process of root object \( l(IP', 0) \), and with object \( o(l(IP, 0), 0) \) sending a message to \( o(l(IP', 0), 0) \).

Mobile objects are specified as objects of the following class MobileObject:

\[
\text{class MobileObject |}
\begin{align*}
\text{mod} & : \text{Module}, & \text{*** rewrite rules of the mobile object} \\
\text{s} & : \text{Term}, & \text{*** current state} \\
\text{gas} & : \text{Nat}, & \text{*** bound on resources} \\
\text{hops} & : \text{Nat}, & \text{*** number of hops} \\
\text{mode} & : \text{Mode}. & \text{*** objects in motion cannot be active}
\end{align*}
\]

\footnote{\textbf{We use here the Full Maude object-oriented notation for defining classes. However, the actual implementation of Mobile Maude is made in Core Maude, because Full Maude does not support external objects. The complete code for Mobile Maude including the corresponding declarations in Core Maude for the classes MobileObject and RootObject can be found in \url{http://maude.sip.ucm.es/mobilemaude}.}}
The sorts \texttt{Module} and \texttt{Term}, associated to the attributes \texttt{mod} and \texttt{s}, respectively, are sorts in the module \texttt{META-LEVEL}. The mobile object’s \texttt{module} must be object-based, and the mobile object’s \texttt{state} must be the metarepresentation of a pair of configurations meaningful for that module and having the form $C \ & \ C'$, where $C$ is a configuration of objects and messages—unprocessed incoming messages and inter-inner-objects messages, see below—and $C'$ is a multiset of messages—the \texttt{outgoing} messages tray. One of the objects in the configuration of objects and messages is supposed to have the same identifier as the mobile object it is in. We sometimes refer to this object as the \textit{main} one, which in most cases will be the only one. Therefore, we can think of a mobile object as a \textit{wrapper} that encapsulates the state and code of its inner object and mediates its communication with other objects and its mobility. For this reason, Figure 1 depicts mobile objects by two concentric circles, with the inner object and its incoming and outgoing messages contained in the inner circle.

To maintain the forwarding information up to date (see below), the definition of the class \texttt{MobileObject} includes the attribute \texttt{hops}, which stores the number of “hops” from one process to another. To guarantee that all mobile objects eventually have some activity, and as a bound on the resources they can consume, they have a \texttt{gas} attribute. Finally, an object’s \texttt{mode} is only \textbf{active} inside the belly of a process: on-transit objects are \textbf{idle}.

The class \texttt{RootObject} of root objects is declared as follows:

```plaintext
class RootObject |
  cnt : Nat,                  \textit{counter to generate mobile obj. names}
guests : Set{Oid},           \textit{objects in the location}
forward : Map{Nat, Tuple{Loc, Nat}}, \textit{forwarding information}
state : RootObjectState,    \textit{idle, waiting-connection, or active}
neighbors : Map{Loc, Oid},  \textit{associates a socket to each location}
defNeighbor : Default{Oid} . \textit{default socket}
```

We assume that each located configuration contains one and only one root object, plus the messages and mobile objects \textit{currently residing} in such a
Durán, Riesco, and Verdejo

process. Located configurations, running on different Maude processes, make up a distributed configuration. Mobile objects can move from one process (located configuration) to another.

The root object of each process keeps information about the mobile objects currently in it in the guests attribute. Mobile objects are named with identifiers of the form o(L, N). The attribute cnt stores a counter to generate such unique new mobile object names. Since mobile objects may move from one process to another, reaching them by messages is nontrivial. The solution adopted in Mobile Maude [3] is that, when a message’s addressee is not in the current process, the message is forwarded to the addressee’s parent process (the process it was created at). Each root object stores forwarding information about the whereabouts of its children in its forward attribute, a partial function in Map{Nat, Tuple{Loc, Nat}} that maps child number n to a pair consisting of the name of the located process in which the object currently resides, and the number of “hops” to different processes that the mobile object has taken so far. The number of hops is important in disambiguating situations when old messages (containing old location information) arrive after newer messages containing current location. The most current location is that associated with the largest number of hops. Whenever a mobile object moves to a new process, the object’s parent process is always notified. Note that this system does not guarantee message delivery in the case that objects move more rapidly than messages.

In the previous version of Mobile Maude [3,4], all the processes were in the same configuration, and reaching a particular process was represented by one single rule. However, in this new version, when a mobile object moves to a different location, or a message is sent to a mobile object in a different location, since we use TCP sockets to connect processes, we need to know which of the sockets must be used to send the information. The root object in the process is in charge of sending it through the appropriate socket. Assuming that all processes are directly connected to each other is not realistic, would be very limited in the number of processes we could connect, and would make the task of connecting a new process a really expensive one. Fortunately, connectivity between two nodes does not necessarily imply a direct connection between them. An indirect connectivity may be achieved among a set of cooperating nodes. Nevertheless, just because a set of hosts are directly or indirectly connected to each other does not mean that we have succeeded in providing host-to-host connectivity. When a source node wants the network to deliver a message to a certain destination node, it specifies the address of the destination node. If the sending and receiving nodes are not directly connected, then the nodes of the network between them—switchers and routers—use this address to decide how to forward the message toward the destination. The process of determining systematically how to forward

\footnote{As we will see in the coming sections, root objects send messages through buffered sockets. We discuss the used of sockets and buffered sockets in Section 4.}
messages toward the destination node based on its address—which is usually called routing—is nontrivial. Here, we assume a very simple, although quite general, approach consisting in having a routing table in each root object. Such a table gives the socket through which a message must be sent if one wants to reach a particular location. If there is a socket between the source and the target of the message then it reaches its destination in a single step; otherwise the forwarding have to be repeated several times. The neighbors attribute maintains such a routing table as a map associating socket object identifiers to location identifiers. That is, the attribute neighbors stores in a partial function \texttt{Map\{Loc, Oid\}} information on the sockets through which sending the data to reach a particular location.

In case there is no socket associated to a particular location in the map neighbors, there can be a default socket stored in the attribute defNeighbor. Nevertheless, the value of the defNeighbor attribute may also be unspecified. The sort Default\{X\} declared in the module DEFAULT-ELEMENT below adds a default value to the sort used in the instantiation of the module. We define the parameterized functional module DEFAULT-ELEMENT\{X :: TRIV\} in which we declare a sort Default\{X\} as a supersort of the sort Elt of the parameter theory, and a constant \texttt{null} of sort Default\{X\}.

\begin{verbatim}
  fmod DEFAULT-ELEMENT\{X :: TRIV\} is
    sort Default\{X\}.
    subsort X$Elt < Default\{X\}.
    op null : -> Default\{X\} [ctor].
  endfm
\end{verbatim}

Then, since defNeighbor is declared of sort Default\{Oid\}, it can take as value either an object identifier or \texttt{null}.

If there is no socket associated to a particular location and a default one has not been specified then the data is not delivered. Note that this model allows us to represent many different network architectures, and, although we do not care here about it, the routing information may be updated and used in a very flexible way. We will explain how to build a very simple architecture in Section 4.2.

Finally, a root object may be in state \texttt{idle}, \texttt{waiting-connection}, or \texttt{active}. The attribute state will take one of these values. Root objects are only idle when they are created, being their first action either being activated as a client or server socket. They stay in \texttt{waiting-connection} until they get the confirmation from the server socket, passing then to \texttt{active} mode, state in which they will develop their normal activity.

Mobile Maude system code is specified by a relatively small number of rules for root objects, mobile objects, mobility, and message passing. Such

---

\footnote{We only consider the case of a source node wanting to send a message to a single destination node (unicast). The cases of multicasting—the source node wants to send a message to some subset of the nodes on the network—and broadcasting—the source node wants to send a message to all the nodes on the network—could similarly be specified.}
rules work in an *application-independent* way. Application code, on the other hand, can be written as Maude object-based modules with great freedom, except for being aware that, as explained in Section 2, the top level of the state of a mobile object has to be a pair of configurations, with the second component containing outgoing messages and the first containing the inner object(s) and incoming messages.

```maude
sort MobObjState .

op _&_ : Configuration Configuration -> MobObjState [ctor] .
```

The messages being pulled in or out of a mobile object must be of the form `to O : C, go(L), go-find(O, L), newo(Mod, Conf, O), or kill, for L a location (of sort Loc), O a mobile object identifier (of sort Mid), C a term of sort Contents, Mod a term of sort Module, and Conf a term of sort Configuration`. Such messages may in fact be understood as commands that the object—or one of the objects—in the inner configuration of a mobile object gives to it. Thus, an object may send a message with contents `C` to the object `O` with the message `to O : C`; may request to move from its current location to a given location `L` with the `go(L)` message; may request going to the location in which the mobile object `O` resides, which is possibly `L`, with the message `go-find(O, L)`; may request creating a new mobile object with module `Mod`, initial state `Conf`, and temporal identifier of the main object in such a configuration `O`, with the message `newo(Mod, Conf, O)`; or may request the destruction of the mobile object it resides into with the message `kill`. The definition of all these ingredients are defined in the module `MOBILE-OBJECT-ADDITIONAL-DEFS`, which is assumed to be imported by the user in all his Mobile Maude applications.

Note that messages being sent to other mobile objects must be of the form `to_:_`, with the addressee of the message as first argument and a term of sort `Contents` as second argument. The definition of such a sort is left to each particular application (see Section 5), which in fact let the user the freedom to define any kind of message, with the restriction of having the identifier of the addressee as first argument.

### 3 Mobile Maude’s rewriting semantics

The entire semantics of Mobile Maude can be defined by a relatively small number of rewrite rules written in Maude. We should think of such rules as a implementation/specification of the *system code* of Mobile Maude, that operates in an application-independent way providing all the object creation and destruction, message passing, and object mobility primitives.

We give the flavor of Mobile Maude’s rewriting semantics by commenting on some of its rules. In particular, we focus on the rules in charge of delivering inter-object messages since, in addition to illustrating the general approach (a more detailed discussion may be found in [3] and [4]), it is directly related to the main novelty in the new implementation: sockets. The complete specification, including other rules in the same style can be found in...
There are three kinds of communication between objects. Objects inside the same mobile object can communicate with each other by means of messages with any format, and such communication may be synchronous or asynchronous. Objects in different mobile objects may communicate when such mobile objects are in the same process and when they are in different processes; in these cases, the actual kind of communication is transparent to the mobile objects, but such communication must be asynchronous through messages of the form to:_:, as explained above. If the addressee is an object in a different mobile object, then the message must be put by the sender object in the second component of its state (the outgoing messages tray). The system code will then send the message to the addressee object. First the message is pulled out of the object’s outgoing tray.

\[
\text{rl [message-out-to]} : \\
< O : V@MobileObject | \\
\text{mod : MOD, s : '}_&_\text{[T, to:_:[T', T']]}\text{], mode : active, AtS} > \\
=> < O : V@MobileObject | \\
\text{mod : MOD, s : '}_&_\text{[T, 'none.\text{Configuration}', mode : active, AtS} > \\
\text{(to downTerm(T', o(l("null", 0), 0)) \{} T’ \}) .}
\]

Once the message is out of the mobile object, it can be appropriately delivered. The \text{msg-send} rule below redirects messages addressed to mobile objects in different locations.

\[
\text{crl [msg-send]} : \\
< L : V@RootObject | \text{state : active, guests : OS, forward : F, AtS} > \\
\text{(to o(L, N) \{} T \}) \\
=> < L : V@RootObject | \text{state : active, guests : OS, forward : F, AtS} > \\
\text{Send(p1(F[N]), L, to o(L, N) hops p2(F[N]) in p1(F[N]) \{} T \}) \\
\text{if (p1(F[N]) =/= L) /\ (not o(L, N) in OS) .}
\]

Notice the use of the message

\[
\text{op Send : Oid Oid Msg -> Msg [ctor msg format (b o)] .}
\]

to send messages to the appropriate locations. The first and second arguments of the \text{Send} message are, respectively, the addressee and sender of the message, and the third argument is the message being sent. We will see in Section 4 how the \text{Send} messages will be used to send the corresponding data through the appropriate sockets.

The arrival of an inter-object message to a location where the addressee object is, is handled by the following rule. The message is just put in the location so the object can get it.

\[
\text{rl [msg-arrive-to-loc]} : \\
\text{to o(L, N) hops H in L' \{} T' \} } \\
< L' : V@RootObject | \text{state : active, guests : (o(L, N), OS), AtS} > \\
=> < L' : V@RootObject | \text{state : active, guests : (o(L, N), OS), AtS} > \\
\text{to o(L, N) \{} T' \} .}
\]

Once the message reaches its addressee object, the message must be in-
serted in—*push into*—the state of such a mobile object. To make sure that the mobile object will remain in a valid state, we check that the metarepresentation of the corresponding message is a valid message in the module of the object.

\[
\text{rl} \ [\text{msg-in}] :
\text{to} \ O \ { \ T } \\
< O : \text{V}@\text{MobileObject} | \text{mod} : \text{MOD}, s : '_{\&}[T', T''], \text{AtS} > \\
=> \text{if} \ \text{sortLeq(MOD, leastSort(MOD, 'to_:_[upTerm(O), T]), 'Msg)} \\
\text{or} \ \text{sortLeq(MOD, 'Msg, leastSort(MOD, 'to_:_[upTerm(O), T]))} \\
\text{then} < O : \text{V}@\text{MobileObject} | \\
\text{mod} : \text{MOD}, s : '_{\&}['_{'to_:_[upTerm(O), T], T'}], T''], \text{AtS} > \\
\text{else} < O : \text{V}@\text{MobileObject} | \text{mod} : \text{MOD}, s : '_{\&}[T', T''], \text{AtS} > \\
\text{fi} .
\]

4 Socket handling

Maude 2.2 supports rewriting with external objects and an implementation of sockets as the first such external object. Rewriting with external objects is started by the command *erewrite* (abbreviated *erew*) which is like *frewrite* except it allows messages to be exchanged with external objects that do not reside in the configuration.

*Sockets* are accessed using the messages declared in the module SOCKET, which can be found in the file *socket.maude* distributed with Maude. We briefly describe here Maude sockets. For a complete explanation of Maude sockets, their use, and examples, we refer the reader to the Maude manual [2]. Currently only IPv4 TCP sockets are supported; other protocol families and socket types may be added in the future.

The external object named by the constant *socketManager* is a factory for socket objects. To create a client socket, a message *createClientTcpSocket* (*socketManager, ME, ADDRESS, PORT*) has to be sent to the *socketManager*, where *ME* is the name of the object the reply should be sent to, *ADDRESS* is the name of the server you want to connect to, and *PORT* is the port you want to connect to (say 80 for HTTP connections). The reply will be the message *createdSocket*(*ME, socketManager, SOCKET-NAME*) where *SOCKET-NAME* is the name of the newly created socket and *REASON* is the operating system’s terse explanation of what went wrong. All errors on a client socket are handled by closing the socket.

You can then send data to the server with a message *send(SOCKET-NAME, ME, DATA)* which elicits the message *sent(ME, SOCKET-NAME)*. Similarly you can receive data from the server with a message *receive(SOCKET-NAME, ME)* which elicits the message *received(ME, SOCKET-NAME, DATA)*.

To have communication between two Maude interpreter instances, one of them must take the server role and offer a service on a given port. To create a server socket, you send *socketManager* a message

*createServerTcpSocket* (*socketManager, ME, PORT, BACKLOG*)
where \texttt{PORT} is the port number and \texttt{BACKLOG} is the number of queue requests for connection that you will allow. The response is the message
\begin{verbatim}
createdSocket(ME, socketManager, SERVER-SOCKET-NAME).
\end{verbatim}
Here \texttt{SERVER-SOCKET-NAME} refers to a server socket. The only thing you can do with a server socket is to accept clients, by means of the message
\begin{verbatim}
acceptClient(SERVER-SOCKET-NAME, ME)
\end{verbatim}
which elicits the message
\begin{verbatim}
acceptedClient(ME, SERVER-SOCKET-NAME, ADDRESS, NEW-SOCKET-NAME).
\end{verbatim}
Here \texttt{ADDRESS} is the originating address of the client and \texttt{NEW-SOCKET-NAME} is the name of the socket you use to communicate with that client. This new socket behaves just like a client socket for sending and receiving.

As we have seen in Section 3, the specification of Mobile Maude does not know about sockets. The only place where we get close to sockets in such a specification is when using the \texttt{Send} messages, which is in fact not a socket message, but a \textit{buffered socket} one. We introduce in Section 4.1 buffered sockets, a kind of filter class that makes Mobile Maude independent of sockets at the same time it adds some additional functionality. As we will see in Section 6, this independence is precisely what allows us to model check Mobile Maude specifications in a rather clean way. Section 4.2 talks about the architecture of the systems, on how processes get connected, and show how to do it for a very simple architecture.

4.1 Buffered sockets

TCP sockets do not preserve message boundaries. Thus, sending e.g. messages “ONE” and “TWO” might result in the reception of messages “ON” and “ETWO”. Although not relevant in other applications, in the current case we need to guarantee that messages are received as originally sent; for instance, if a mobile object is sent through a socket, we need to be able to recover a valid object, in the same valid state in which it was sent, upon the reception of the message. To guarantee message boundaries we use a filter class \texttt{BufferedSocket}, defined in the module \texttt{BUFFERED-SOCKET}. This module is completely independent of Mobile Maude, and can therefore be used in other applications. We interact with buffered sockets in the same way we interact with sockets, with the only difference that all messages in the module \texttt{SOCKET} have been capitalized to avoid the confusion, being the boundary control completely transparent to the user.

When a buffered socket is created, in addition to the socket object through which the information will be sent, a \texttt{BufferedSocket} object is also created on each side of the socket (one in each one of the configurations between which the communication is established). All messages sent through a buffered socket are manipulated before they are sent through the socket underneath. When a message is sent through a buffered socket, a mark is placed at the end of it;
the BufferedSocket object at the other side of the socket stores all messages received on a buffer, in such a way that when a message is requested the marks placed say which part of the information received must be given as the next message.

An object of class BufferedSocket has three attributes: read, of sort String, which stores the messages read, bState, which indicates whether the filter is idle or active, and waiting, which indicates if we are waiting for a sent message (when we are waiting, we do not allow sending new messages).

\[
\text{sort BState}.
\]

\[
\text{ops idle active : -> BState [ctor].}
\]

\[
\text{class BufferedSocket | read : String, bState : BState, waiting : Bool.}
\]

We do not give here all the rules, but only those related to the sending of messages.

Once a connection has been established, and a BufferedSocket object has been created on each side, messages can be sent and received. When a Send message is received, the buffered socket sends a send message with the same data plus a mark\(^4\) to indicate the end of the message.

\[
\text{rl [send]}:
< b(SOCKET) : \emptyset \text{BufferedSocket | bState : active,}\n\quad \text{waiting : false, Atts} >
\]

\[
\text{Send(b(SOCKET), O, DATA)}
\Rightarrow < b(SOCKET) : \emptyset \text{BufferedSocket | bState : active,}\n\quad \text{waiting : true, Atts} >
\]

\[
\text{send(SOCKET, O, DATA + "}\).}
\]

The key is then in the reception of messages. A BufferedSocket object is always listening to the socket. It sends a receive message at start up and puts all the received messages in its buffer. Notice that a buffered socket goes from idle to active in the buffer-start-up rule. A Receive message is then handled if there is a complete message in the buffer, that is, there is a mark on it, and results in the reception of the first message in the buffer, which is removed from it.

\[
\text{rl [buffer-start-up]}:
< b(SOCKET) : \emptyset \text{BufferedSocket | bState : idle, Atts} >
\Rightarrow < b(SOCKET) : \emptyset \text{BufferedSocket | bState : active, Atts} >
\]

\[
\text{receive(SOCKET, b(SOCKET)).}
\]

\[
\text{rl [received]}:
< b(SOCKET) : \emptyset \text{BufferedSocket | bState : active, read : S, Atts} >
\]

\[
\text{received(b(SOCKET), O, DATA)}
\Rightarrow < b(SOCKET) : \emptyset \text{BufferedSocket | bState : active,}\n\quad \text{read : (S + DATA), Atts} >
\]

\[
\text{receive(SOCKET, b(SOCKET)).}
\]

\[
\text{crl [Received]}:
< b(SOCKET) : \emptyset \text{BufferedSocket | bState : active, read : S, Atts} >
\]

\[\text{4 In the rules we use the string "#" as mark, but any other could be used. Note that the user data sent through the sockets should not contain such a mark.}\]
Receive(b(SOCKET), O)  
  => < b(SOCKET) : V@BufferedSocket | bState : active, read : S', Atts > 
  Received(O, b(SOCKET), DATA) 
  if N := find(S, ",#", 0) \ DATA := substr(S, 0, N) 
    \ S' := substr(S, N + 1, length(S)) .

4.2 A client/server architecture

Although the specification of Mobile Maude presented in the previous sections allow different configuration of processes, we present here a very simple client/server architecture. We distinguish clients and servers by declaring two subclasses ServerRootObject and ClientRootObject of RootObject, with no additional attributes, although with different behavior.

    class ClientRootObject .
    class ServerRootObject .
    subclasses ClientRootObject ServerRootObject < RootObject .

The architecture we present here consists in a process with a server root object, and several processes with client root objects. The server is connected to all clients, and a client is connected only to the server. If a mobile object residing in a client process—a process with a client root object in it—wants to move to (or send a message to a mobile object in) another client process, then it will be sent to the server process, and from there to its final destination. That is, we have a very simple star network, with a server root object in the middle redirecting all messages.

When a ServerRootObject is created it send an AcceptClient message indicating that it is ready to accept clients through the server socket. When a ClientRootObject is created it first tries to establish a connection with the sever by sending a CreateSocket message. In the rule acceptedClient below, in addition to sending messages AcceptClient and Receive indicating, respectively, that it is ready to accept new clients through the server socket, and messages through the new socket, the server root object that gets the AcceptedClient message sends a start-up message new-socket communicating its identifier. Notice that the client knows the address and port of the server root object, but not its identity. In this first message the server sends its name to its client, allowing to this one establishing the association between the socket and the identity of the object in it.

    rl [acceptedClient] :
      < 1(IP, N) : V@ServerRootObject | state : active, AtS >
    AcceptedClient(1(IP, N), SOCKET, IP', NEW-SOCKET)
  => < 1(IP, N) : V@ServerRootObject | state : active, AtS >
    AcceptClient(SOCKET, 1(IP, N))
    Receive(NEW-SOCKET, 1(IP, N))
    Send(NEW-SOCKET, 1(IP, N), msg2string(new-socket(1(IP, N)))) .

Since the third argument of a Send message is a String, the message being sent is transformed with the msg2string function; string2msg does the inverse transformation.
The response to a client root object’s socket connection request is handled by the following rule `connected`, where a client also sends a `new-socket` message right after the socket is created.

\[
\begin{align*}
\text{rl } [\text{connected}] &: \quad < l(IP, N) : V@ClientRootObject \mid \text{state} : \text{waiting-connection}, \text{AtS} > \\
& \quad \text{CreatedSocket}(O, \text{SOCKET-MANAGER}, \text{SOCKET}) \\
& \quad => < l(IP, N) : V@ClientRootObject \mid \text{state} : \text{active}, \text{AtS} > \\
& \quad \text{Receive}(\text{SOCKET}, l(IP, N)) \\
& \quad \quad \text{Send}(\text{SOCKET}, l(IP, N), \text{msg2string}(\text{new-socket}(l(IP, N)))) . \\
\end{align*}
\]

The attributes `neighbors` and `defNeighbor` are key for sending messages through the appropriate sockets. The reason why the first message sent through a socket after its creation is the message `new-socket` is to initialize these attributes. When it is received, depending on whether the receiver is a client or a server, and whether there is already a default neighbor or not, one or another action is taken.

To avoid unintended loops in the delivering of messages, we assume that server root objects do not have default neighbors. For clients, the first connection is made the default one.

\[
\begin{align*}
\text{crl } [\text{Received}] &: \quad < O : V@RootObject \mid \text{state} : \text{active}, \text{neighbors} : \text{empty}, \\
& \quad \quad \text{defNeighbor} : \text{null}, \text{AtS} > \\
& \quad \text{Received}(O, \text{SOCKET}, \text{DATA}) \\
& \quad => < O : V@RootObject \mid \text{state} : \text{active}, \\
& \quad \quad \text{neighbors} : \text{insert}(L, \text{SOCKET}, \text{empty}), \\
& \quad \quad \text{defNeighbor} : \text{if } V@RootObject == \text{ServerRootObject} \\
& \quad \quad \quad \quad \text{then null} \\
& \quad \quad \quad \quad \text{else SOCKET} \\
& \quad \quad \quad \quad \text{fi}, \\
& \quad \quad \text{AtS} > \\
& \quad \quad \text{Receive}(\text{SOCKET}, O) \\
& \quad \quad \text{if } \text{new-socket}(L) := \text{string2msg}(\text{DATA}) . \\
\end{align*}
\]

If not a `new-socket` message, then the message is just left in the configuration.

\[
\begin{align*}
\text{crl } [\text{Received}] &: \quad < O : V@RootObject \mid \text{state} : \text{active}, \text{AtS} > \\
& \quad \text{Received}(O, \text{SOCKET}, \text{DATA}) \\
& \quad => < O : V@RootObject \mid \text{state} : \text{active}, \\
& \quad \quad \text{string2msg}(\text{DATA}) \text{ Receive}(\text{SOCKET}, O) \\
& \quad \text{if not } \text{new-socket}(\text{DATA}) . \\
\end{align*}
\]
5 A buying printers example

In this section we present a simple application to illustrate how mobile application code can be written in Maude and can be wrapped in mobile objects. In this example we have printers, buyers, and sellers; a buyer agent visits several printer sellers which provide him information on their printers. The buyer looks for the cheapest printer, and once he has visited all the sellers, he goes back to the location of the seller offering the best price.

From the previous description, we can identify different actors, which may move freely from one process to another, and therefore they should be represented as mobile objects. In the Mobile Maude approach the specification of the system consists of objects embedded inside mobile objects, which communicate to each other via messages. In addition to the term representing its state, each mobile object carries the code managing the behavior of the configuration of objects and messages representing such a state. The main difference with respect to the specification of systems in Maude is that these objects must be aware of the fact that they are inside mobile objects, and that in order to communicate with (objects in) other mobile objects or to use some of the system messages available, they must follow the appropriate procedure.

In our sample application we have two different classes of mobile objects: sellers and buyers. A buyer visits several sellers. The buyer asks each seller he visits for the description of the seller’s printer (represented here only by its price). The seller sends back this information, which the buyer keeps if it corresponds to a better (cheaper) printer. Otherwise he discards it. Once the buyer has visited all the sellers he knows, he goes back to the location of the best offer.

We represent sellers and buyers as objects of respective classes Buyer and Seller. Such objects in the application code will then be embedded as inner objects of their corresponding mobile objects.

The class Seller has a single attribute description with the printer price (a natural number).

```plaintext
class Seller | description : Nat .
```

Sellers receive messages of the form \texttt{get-printer-price(B)}, with B the identifier of the buyer mobile object sending the message. A seller can send messages of the form \texttt{printer-price(N)}, with N a natural number representing the printer’s price. Both are defined of sort \texttt{Contents}, declared in the module \texttt{MOBILE-OBJECT-ADDITIONAL-DEFS}.

```plaintext
op get-printer-price : Mid -> Contents .
op printer-price : Nat -> Contents .
```

A seller’s behavior is represented by the following single rewrite rule: When a seller receives a description (price) request, it sends the description back to the seller.
Note the use of the \_&\_ constructor. Since the printer description is sent to an object outside the mobile object in which the Seller object is located, the message is placed in its righthand side. The rule \texttt{get-des} is applied only if the outgoing messages tray is empty, making sure in this way that any previous outgoing message has been handled. The \_&\_ operator is the top operator of the term representing the state of the mobile object, and therefore, since there may be other objects and messages in the configuration in its lefthand side, we include a variable \texttt{Conf} of sort \texttt{Configuration} to match the rest. Note also how an object may communicate to objects in other mobile objects, which may be in different locations, in a completely transparent way.

A buyer has an attribute \texttt{sellers} with a list of the identifiers of the known sellers. It also has an attribute \texttt{status} with its current state: \texttt{onArrival}, \texttt{asking}, \texttt{done}, or \texttt{buying}. Finally, the buyer keeps information about the printer with the best price in the attributes \texttt{price} and \texttt{bestSeller} of sorts, respectively, \texttt{Default{Nat}} and \texttt{Default{Oid}}. Initially, these two last attributes are \texttt{null}.

\begin{verbatim}
class Buyer | sellers : List{Mid}, status : Status,  
            price : Default{Nat}, bestSeller : Default{Oid} .
\end{verbatim}

The first rewrite rule, \texttt{move}, handles the travels of the buyer to request information on printers: if it is not in the middle of a request (its status is \texttt{done}) and there is at least one seller name in the \texttt{sellers} attribute, it asks the system to take it to the host where the next seller is.

\begin{verbatim}
rl [move] : 
  < B : V@Buyer | status : done, sellers : o(L, N) . OS, AtS >  
       Conf & none  
=> < B : V@Buyer | status : onArrival, sellers : o(L, N) . OS, AtS > 
       Conf & go-find(o(L, N), L) .
\end{verbatim}

Since Mobile Maude guarantees that mobile objects moving from one location to another are idle, we know that, once the \texttt{go-find} command is given in the \texttt{move} rule, the buyer object will not be able to do anything until the mobile object in which it is embedded is set to active, that is, until it has reached the seller’s process. Therefore, since there is no rule taking a \texttt{Buyer} object in \texttt{onArrival} state and a nonempty outgoing messages tray, this object will not do anything until it reaches its destination.

On arrival, the buyer asks the seller for the printer description.

\begin{verbatim}
rl [onArrival] : 
  < B : V@Buyer | status : onArrival, sellers : S . OS, AtS >  
       Conf & none  
=> < B : V@Buyer | status : asking, sellers : S . OS, AtS > 
       Conf & (to S : get-printer-price(B)) .
\end{verbatim}
When the printer price arrives, if it corresponds to the first buyer (the attribute price is null) the buyer keeps it as the best known price; or compares it with the best known printer and updates its information if needed. Notice that the first identifier in the list of known sellers gives us the identifier of the seller it is currently interacting with.

\[
\text{rl [new-des]} : \quad \begin{array}{l}
\text{(to } B : \text{printer-price}(N)) \\
\quad < B : V@Buyer | \text{status} : \text{asking}, \text{price} : \text{null}, \text{bestSeller} : \text{null}, \\
\quad \text{sellers} : S, \text{OS, AtS} > \\
\quad \Rightarrow < B : V@Buyer | \text{status} : \text{done}, \text{price} : N, \text{bestSeller} : S, \\
\quad \text{sellers} : \text{OS, AtS} > .
\end{array}
\]

\[
\text{rl [new-des]} : \quad \begin{array}{l}
\text{(to } B : \text{printer-price}(N)) \\
\quad < B : V@Buyer | \text{status} : \text{asking}, \text{price} : N', \text{bestSeller} : S', \\
\quad \text{sellers} : S, \text{OS, AtS} > \\
\quad \Rightarrow \text{if } (N < N') \\
\quad \text{then } < B : V@Buyer | \text{status} : \text{done}, \text{price} : N, \text{bestSeller} : S, \\
\quad \text{sellers} : \text{OS, AtS} > \\
\quad \text{else } < B : V@Buyer | \text{status} : \text{done}, \text{price} : N', \text{bestSeller} : S', \\
\quad \text{sellers} : \text{OS, AtS} > \\
\quad \text{fi .}
\end{array}
\]

Notice that since these last rules do not imply the sending of any message out of the mobile object, we do not need to use the $\&$ operator and variable Conf to wrap the whole state.

Finally, when the list of remaining sellers is empty, the buyer travels to find the best buyer and reaches the \textit{buying} status.

\[
\text{rl [buy-it]} : \quad \begin{array}{l}
\quad < B : V@Buyer | \text{status} : \text{done}, \text{sellers} : \text{no-id}, \text{bestSeller} : o(L,N), \text{AtS} > \\
\quad \text{Conf & none} \\
\quad \Rightarrow < B : V@Buyer | \text{status} : \text{buying}, \text{sellers} : \text{no-id}, \\
\quad \text{bestSeller} : o(L,N), \text{AtS} > \\
\quad \text{Conf & go-find(o(L,N), L)} .
\end{array}
\]

Let us see an example of a distributed configuration, and how we can rewrite it by using the \textit{erewrite} command. Our sample buyers/sellers configuration, shown in Figure 2, is constituted by three located configurations, each one to be executed in a Maude process—in this case the three processes run on the same machine, with IP address IP. The first located configuration (shown in the middle of the figure) contains a ServerRootObject, with identifier l(IP, 0), and a mobile object with identifier o(l(IP, 0), 0) with a Seller in its belly. The Maude command to introduce the initial state of this configuration is as follows:

\[
\text{erew <> } < l/IP, 0) : \text{ServerRootObject |} \\
\quad \text{cnt : 1,} \\
\quad \text{guests : o(l/IP, 0), 0),} \\
\quad \text{forward : 0 |-> (l/IP, 0), 0),} \\
\quad \text{neighbors : empty,}
\]

50
Note how the function \texttt{upModule} is used to obtain the metarepresentation of the module \texttt{SELLER}, and how the function \texttt{upTerm} is used to metarepresent the initial state of the inner object.

This configuration must be executed before the other two ones because it contains the object \texttt{ServerRootObject}, which is in the central process of the star network.

The second located configuration (on the left in the figure) contains a \texttt{ClientRootObject}, a \texttt{Buyer} and a \texttt{Seller} with cheaper printers. Finally, the third located configuration (on the right) contains another \texttt{ClientRootObject} and a \texttt{Seller} with the cheapest printers. The Maude commands, introduced in other two different Maude process, are very similar to the previous one.\footnote{The execution of these three commands in three different Maude processes does not finish. And that is because of the blocking behavior of the socket messages like \texttt{receive}. An execution of a Mobile Maude application is not intended to finish since the located configurations are always waiting for messages or mobile objects to come in from other configurations. Due to this fact, it is recommended to execute these applications with the trace on. In this way we can see what is happening in each Maude process. When the execution of a concrete example seems to be finished because we do not see evolution in any of the involved processes, we can finish them by pressing \texttt{^C}.}
Figure 2 shows how the order in which the different actions occur. First the buyer asks to the seller in his same location (price 20). Then the buyer travels to the location on the right (through the location with the ServerRootObject) and asks to the seller who sells printers costing 15. After that, the buyer travels to the middle location and asks to the seller there (price 30). Finally, the buyer travels to the right location to find the seller with the best offer.

6 Model checking Mobile Maude applications

Maude’s model checker [5] allows us to prove properties on Maude specifications when the set of states reachable from an initial state in such a Maude system module is finite. This is supported in Maude by its predefined MODEL-CHECKER module and other related modules, which can be found in the model-checker.maude file distributed with Maude.

The properties to be checked are described by using a specific property specification logic, namely Linear Temporal Logic (LTL) [7,1], which allows specification of properties such as safety properties (ensuring that something bad never happens) and liveness properties (ensuring that something good eventually happens). Then, the model checker can be used to check whether a given initial state, represented by a Maude term, fulfills a given property.

Using the model checker on Mobile Maude is not easy however. Mobile Maude configurations are distributed among several hosts, and therefore the model checker cannot be used directly to prove properties about these global configurations. On the other hand, we would like to check properties on the application code, which is metarepresented in the belly of the mobile objects. We show in the following sections how we have addressed both issues. The former problem has been solved by considering an algebraic specification of the sockets provided by Maude. The later one has been solved by considering two-level properties, stating different properties on each of the reflection levels.

6.1 Redefinition of the SOCKET module

To solve the distribution problem, we have provided an algebraic specification of sockets. We have redefined the SOCKET module, simulating the behavior of sockets on local configurations. This specification expresses processes as terms of a class Process with a single attribute conf. Processes work as hosts in the distributed version, keeping the configuration separated from the others in its attribute. Message passing is then defined between processes instead of between hosts.

Thus, we have specified sockets, socket managers and server sockets to deal with processes:

- The socket manager is now an instance of a class Manager, with a counter attribute to name the new sockets.
- The sockets are instances of a class Socket with attributes source (the
source Process), target (the target Process), and socketState (the socket state). Notice that although we talk about source and target, sockets are bidirectional.

- The server sockets are instances of the class ServerSocket with the attributes address (the server address), port (the server port), and backlog (the number of queue requests for connection that the server will allow). When one object wants to create a server, we create one server socket at process level and the object receives a createdSocket message with the server socket identifier.

Note that there is no need for a client sockets class, they are only processes, so to create a client socket we create a socket with target the server and source the process.

The class Process allows to represent in a single term a whole distributed configuration. The rest of the above mentioned classes and the rewrite rules defined in the new module SOCKET allow to use the specification of Mobile Maude with no more changes. So in order to prove a property about a distributed configuration we have to prove it on the corresponding “local” configuration by using Processes.

6.2 Two-level atomic propositions for the buying printers example

To use the model checker we just need to make explicit two things: the intended sort of states, Configuration, and the relevant state predicates, that is, the relevant LTL atomic propositions.

To be able to model check Mobile Maude application code, we propose defining these predicates at two different levels: the processes level and the inner objects level. In the processes level we look for inner objects which have some properties; in the inner objects level we check such properties.

Let us see an example about the buying printers case study. Suppose we want to prove that the buyer always finds the best price, and that, when he has visited all sellers, he finishes in the process of the seller who has such a best price. If bestPrice & Seller represents the state predicate asserting that the buyer is in the process of the seller with the best offer, then the LTL formula we want to check is $\Diamond \Box \text{bestPrice \& Seller}$, that is, it is always possible to reach an state where the property bestPrice & Seller is fulfilled and from that state the property keeps invariant.

First, we define when a top configuration of processes fulfill such a property. For it, we use an auxiliary predicate bestPrice & Seller with an argument, (the metarepresentation of) the best price, obtained by means of the auxiliary function minPrice.

```plaintext
op bestPrice & Seller : -> Prop .
op bestPrice & Seller : Term -> Prop .
eq C |= bestPrice & Seller = C |= bestPrice & Seller(minPrice(C)) .
```

The definition of bestPrice & Seller(N) recursively traverses all the pro-
cesses going inside each configuration looking for a seller with the given price and a buyer who has it as the best price.

\[
\text{op existsSeller : Term} \rightarrow \text{Prop}.
\]

\[
\text{op existsBuyer : Term} \rightarrow \text{Prop}.
\]

\[
\text{eq (C < PID : Process | conf : C' >) \mid= bestPrice&Seller(N)} = \text{(C \mid= bestPrice\&Seller(N)) or}
\]

\[
((C' \mid= \text{existsSeller(N)}) \text{ and (C' \mid= existsBuyer(N))}).
\]

\[
\text{eq C \mid= bestPrice\&Seller(N) = false [owise].}
\]

\[
\text{eq (< O : MobileObject | s : ('_&_\text{TERM, TERM'}), AtS > C) \mid= existsSeller(N)} =
\]

\[
\text{(getTerm(metaReduce(upModule('EXAMPLE-PREDS, false),}
\]

\[
\text{'_|=_\text{TERM, 'exSeller[N]}]) == 'true.Bool)
\]

\[
\text{or (C \mid= \text{existsSeller(N)}}).
\]

\[
\text{eq C \mid= existsSeller(N) = false [owise].}
\]

The definition of \text{existsSeller(N)} uses the predicate \text{exSeller} defined at the inner objects level. The predicate \text{existsBuyer(N)} is defined in the same way. The module \text{EXAMPLE-PREDS} includes the definition of the predicates \text{exSeller} and \text{exBuyer}.

\[
\text{op exSeller : Nat} \rightarrow \text{Prop}.
\]

\[
\text{op exBuyer : Nat} \rightarrow \text{Prop}.
\]

\[
\text{eq < S : Seller | description : N, AtS > C \mid= exSeller(N) = true}.
\]

\[
\text{eq C \mid= exSeller(N) = false [owise].}
\]

\[
\text{eq < B : Buyer | price : N, status : buying, AtS > C \mid= exBuyer(N)} =
\]

\[
\text{true}.
\]

\[
\text{eq C \mid= exBuyer(N) = false [owise]}.\]

Notice that these atomic propositions are defined at the level of the application code.

After having defined these predicates, the Maude command to use the model checker for examining whether an initial configuration \text{initial} fulfills the formula \text{\Diamond bestPrice\&Seller} is as follows:

\[
\text{Maude> red modelCheck(initial, <> [] bestPrice\&Seller).}
\]

\[
\text{result Bool: true}
\]

7 Conclusions

We have presented a distributed implementation of Mobile Maude where mobile objects, carrying its code and internal state, can travel from one machine to another one. Sockets now provided by Maude are used to achieve this goal in a really distributed setting.

Although the main concepts and design decisions have been maintained as they were presented in the first implementation of the language [3], the parts regarding how the distributed state is represented and controlling how messages and mobile objects are transferred between different machines are
completely new. We have designed these new parts in a way as independent of the concrete underlying architecture as possible.

We have used the language to implement several case studies. Here we have shown an application where a printer buyer has to choose the seller offering the cheapest printer. The conference reviewing system presented in [4] has also been migrated to this new version of the language.

By explicitly representing the different processes in which a distributed application is allocated, we can represent the whole distributed state in a single term, and by redefining the predefined module SOCKET we can use the Mobile Maude implementation shown in this paper to execute/simulate the behavior of such application. This compact representation can be used to prove properties it fulfills by means of Maude’s model checker.

References


Partial Order Reduction for Rewriting Semantics of Programming Languages

Azadeh Farzan and José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign
Urbana, USA

Abstract
Software model checkers are typically language-specific, require substantial development efforts, and are hard to reuse for other languages. Adding partial order reduction (POR) capabilities to such tools typically requires sophisticated changes to the tool’s model checking algorithms. This paper proposes a new method to make software model checkers language-independent and improving their performance through POR. Getting the POR capabilities does not require making any changes to the underlying model checking algorithms: for each language $L$, they are instead achieved through a theory transformation $R_L \rightarrow R_L^{+POR}$ of $L$’s formal semantics, rewrite theory $R_L$. Under very minimal assumptions, this can be done for any language $L$ with relatively little effort. Our experiments with the JVM, a Promela-like language and Maude indicate that significant state space reductions and time speedups can be gained for tools generated this way.

Key words: Partial order reduction, model checking, programming language semantics, rewriting logic, Maude.

1 Introduction
This paper proposes a new method to make software model checking tools language-independent; and to substantially improve their performance using partial order reduction (POR) (see, for example, [22,14,15,25,1,12,3,18]). The key insight of POR is that the state space explosion caused by the many interleaving computations of a concurrent program can be tamed using the fact that many such computations are semantically equivalent, because they are different descriptions of the same partial order of events. This means that only a representative subset of all the interleaving computations has to be model checked, without losing completeness: the model checking results are
the same as if all computations had been analyzed, but performance can thus be greatly increased.

Although the theoretical foundations of POR are very general and can be applied to many different languages, at present POR-enabled software model checkers are typically language-specific: they typically only work for programs in a particular language such as Java, C, Promela, and so on. One exception to this common situation is the Verisoft tool [15]. However, Verisoft is applied in practice to a limited family of languages. The question this paper raises and presents an affirmative answer to is: can POR-enabled software model checking tools become language-independent in the strong sense of being generic, that is, being applicable not just to a few, but to an unlimited class of languages satisfying some very basic abstract requirements?

This work is part of a broader research effort to develop a range of generic software analysis tools based on formal semantic definitions of programming languages [20,21]. The tools are generic (i.e., language-independent) because their specialization to each particular language, say Java, is based on providing the generic tool with an executable formal semantics of the language in question. We use rewriting logic [19] as a semantic framework, because the distinction between equations and rules in a rewrite theory provides powerful abstraction capabilities [21], and also because its high-performance Maude implementation and tools [5,7] offer a good basis for developing efficient generic tools that have competitive performance.

Within this larger research effort this paper gives an answer to the question of how POR capabilities can likewise be made generic at the tool level. The practical advantages of our approach come from the fact that the prevalent way of developing a software model checker for a specific programming language is a labor-intensive process often requiring man-years of effort; and that language-specific checkers are typically hard to reuse for other languages. Furthermore, adding POR capabilities to a language-specific model checker typically requires sophisticated changes to the tool’s model checking algorithms. By contrast, our generic method can be specialized to any particular language of choice in a few weeks, including the time of developing the formal semantic definition of the chosen language (e.g., Java). In particular, the specific task of adding the POR capabilities to the tool thus obtained can be accomplished in a few hours.

Our method is based on a theory transformation $\mathcal{R}_L \rightarrow \mathcal{R}_{L+POR}$ in which the original rewrite theory $\mathcal{R}_L$ specifying the semantics of the language $L$ is transformed into a stuttering equivalent rewrite theory $\mathcal{R}_{L+POR}$ that accomplishes the desired partial order reduction when used for model checking a given program. This theory transformation approach means that no changes to the underlying model checker are needed to achieve the desired partial order reduction, which is one of the reasons why developing a POR-enabled LTL model checker for a language $L$ using our method requires such little effort. Besides its genericity and short development time, our method has two
additional advantages:

1. **Flexible Partial Order Heuristic Algorithm.** The heuristic algorithm can be specified using a few equations. Although our basic version of the heuristic can in principle work for any programming language, additional optimizations, based on specific knowledge of the given programming language or of the types of programs to be verified, can make the POR reduction considerably more efficient. The tool builder can easily customize the heuristic algorithm, which compares favorably with having to change the source code of a model checker.

2. **Flexible Dependence Relation.** Although a basic dependence relation can generally hold for a certain programming language, additional knowledge of the types of programs that one needs to verify can result in removing some dependencies; for example, Java supports shared memory in general, so we have to assume that memory read/write pairs are generally interdependent; but if the programs being verified do not use the shared memory at all, we can remove this dependency for such programs. Having the dependence relation as an explicit parameter of the partial order reduction module not only contributes to the generality of the method, but also gives the tool builder the advantage of specializing it, based on the type of input programs.

Our generic theory transformation assumes a simple basic interface of functionality in the language $L$. This allows a first phase of automatic transformation of the theory $\mathcal{R}_L$. But this can be followed by a second *language-specific customization* phase supporting features 1–2 above. This can be easily accomplished by adding or customizing a few equations in this second phase, so that detailed knowledge of $L$’s semantics can be used to optimize the reduction; for example, by optimizing the heuristic algorithm and/or by defining a more precise dependence relation using static analysis techniques.

Besides developing its theoretical foundations and establishing its correctness, the practical usefulness of a generic method like the one we propose should be evaluated experimentally. Therefore, we have developed a prototype tool in Maude that, given an original semantics of a language $L$ specified as a rewrite theory $\mathcal{R}_L$, performs the theory transformation $\mathcal{R}_L \mapsto \mathcal{R}_L + \text{POR}$ and can be used to model check LTL properties of programs in $L$ using Maude’s generic LTL model checker. We have applied this prototype to the rewriting semantics of the Java bytecode, a simple Promela-like language, and Maude; and have evaluated the performance of our POR methods for these languages using several benchmarks. The goal of this prototype and experimentation is a *proof-of-concept* one. Therefore, we have not incorporated a number of well-known optimizations that a mature tool should support. Nevertheless, our experiments indicate that, even without such optimizations, substantial gains in time and space can be obtained using our POR method.

The rest of the paper is organized as follows: Section 2 contains the background needed in Section 3, where we discuss the generic method in detail; Section 4 presents the experimental results including the instantiation of the method for the Java bytecode and for a Promela-like language, as well as
Farzan and Meseguer present some performance figures; and Section 5 discusses related work, conclusions, and future directions.

2 Preliminaries

2.1 Rewriting Logic Language Specification

The rewriting logic semantics of a programming language \([20]\) combines and extends both equational/denotational semantics based on *semantic equations*, and structural operational semantics (SOS) based on *semantic rules*. Given a programming language \(L\), its rewriting logic semantics is defined as a rewrite theory \(\mathcal{R}_L = (\Sigma_L, E_L, R_L)\), with \(\Sigma_L\) a signature specifying both the syntax of \(L\) and of operations on auxiliary semantic entities like the store, environment, and so on, with \((\Sigma_L, E_L)\) an equational theory specifying the semantics of the sequential features of \(L\), and with \(R_L\) a collection of (possibly conditional) rewrite rules specifying the semantics of \(L\)'s concurrent features. Under the assumption that \(\mathcal{R}_L\) is coherent \([26]\), equations in \(E_L\) (corresponding to execution of sequential features) are applied until reaching a canonical form, and then rules in \(R_L\) (corresponding to execution of concurrent features) are applied. This key distinction between equations and rules immediately gives the advantage of reductions similar to those in \([24]\). The *invisible states* in \([24]\) are closely related to the reduction steps done by equations in \(E_L\). Only when no more equations from \(E_L\) apply to the state, does a rewrite with a rule in \(R_L\) take place. This makes any sequence of *sequential* instructions in a thread to be executed as an atomic block, without any interleavings. Note that this kind of state reduction is available at the level of the original semantics \(\mathcal{R}_L\); what is now further needed, which is the topic of this paper, is to achieve an additional POR state space reduction by reducing the state explosion due to the execution of concurrent features.

Specifying formally the semantics of a concurrent programming language \(L\) in the Maude rewriting logic language not only yields a language interpreter for free, but also, thanks to the generic analysis tools for rewriting logic specifications that are provided as part of the Maude system \([4]\), additional analysis tools are also automatically provided for \(L\), including a semi-decision procedure to find failures of safety properties, and an LTL model checker. There is already a substantial experience on the practical use of such language definitions and the associated analysis tools for real languages such as Java, the JVM, and a substantial subset of OCaml \([20,11,9]\).

2.2 Background on Partial Order Reduction

A finite transition system is a tuple \((S, S_0, T, AP, L)\), where \(S\) is a finite set of states, \(S_0 \subseteq S\) is the set of initial states, \(T\) is a finite set of transitions such that \(\alpha \in T\) is a partial function \(\alpha : S \rightarrow S\), \(AP\) is a finite set of propositions and \(L : S \rightarrow 2^{AP}\) is the labeling function. A transition \(\alpha\) is *enabled* in a state
if \( \alpha(s) \) is defined. Denote by enabled\((s)\) the set of transitions enabled in \( s \). The main goal of partial order reductions is to find a subset of enabled transitions ample\((s) \subseteq \text{enabled}(s) \) that is used to construct a reduced state space that is behaviorally equivalent. Partial order reduction is based on several observations about the nature of concurrent computations. The first observation is that concurrent transitions are often commutative, which is expressed in terms of an independence relation, \( I \subseteq T \times T \), that is, a symmetric and anti-reflexive relation which satisfies the following condition: for each \((\alpha, \beta) \in I\), and for each state \( s \), if \( \alpha, \beta \in \text{enabled}(s) \) then: (1) \( \alpha \in \text{enabled}(\beta(s)) \) and \( \beta \in \text{enabled}(\alpha(s)) \), and (2) \( \alpha(\beta(s)) = \beta(\alpha(s)) \). Note that \( D = (T \times T) \setminus I \) is the dependence relation. The second observation is that in many cases only a few transitions can change the value of the propositions, which suggests the concept of visibility; a transition \( \alpha \in T \) is invisible if for each \( s \in S \), if \( s' = \alpha(s) \) then \( L(s) = L(s') \).

There are several existing heuristics to compute ample\((s)\). [2] gives a set of four conditions that, if satisfied by ample\((s)\), guarantee a correct reduction of the given state transition system. In Section 3.3, we present a special case of the conditions in [2] which are used in this paper.

### 3 Partial Order Reduction for Language Definitions

#### 3.1 Some Assumptions

In order to devise a general partial order reduction module for semantic definitions of concurrent programming languages, we have to make some basic assumptions about these semantic definitions. These assumptions are quite reasonable and do not limit in practice the class of semantic definitions that we can deal with. They simply specify a standard interface between the semantic definition module and the partial order reduction module. We can enumerate these assumptions as follows: (1) In each program there are entities equivalent to threads which can be uniquely identified by a thread identifier. The computation is performed as the combination of local computations inside individual threads, and communication between these threads through any possible discipline such as shared memory, synchronous or asynchronous message passing, and so on. (2) In any computation step (transition) a single thread is always involved. In other words, threads are the entities that carry out the computations in the system.

#### 3.2 The Theory Transformation

The rewrite theory \( R_L = (\Sigma_L, E_L, R_L) \) specifying the semantics of a concurrent programming language \( L \) is transformed in two steps into the semantically equivalent theory \( R_{L+POR} = (\Sigma_{L+POR}, E_{L+POR}, R_{L+POR}) \) that is equipped with partial order reduction capabilities.
**The Marked-State Theory.**

The objective of the first step of this transformation is to change the original theory \( \mathcal{R}_L \) in order to facilitate the addition of the partial order module. In the transformed theory \( \widehat{\mathcal{R}}_L = (\widehat{\mathcal{S}}_L, \widehat{E}_L, \widehat{R}_L) \): (1) the rewrite rules of \( R_L \) are changed syntactically to only allow one-step rewrites, and (2) the structure of the states of \( \mathcal{R} \) is enriched to allow a specific thread to be marked as enabled. Rewrite rules are then modified to only allow the threads that are marked enabled to make a transition. This way, when the POR heuristic decides on an ample set, the corresponding threads can be marked as enabled, and this causes only the ample transitions to be explored next. Here we give a detailed construction of \( \widehat{R}_L \) and show that \( \mathcal{R}_L \) and \( \widehat{R}_L \) are one-step bisimilar.

We assume that \( \mathcal{R}_L \) is coherent [26] and that all rules in \( \mathcal{R}_L \) are of the form \( l(u(t)) \rightarrow r(u'(t)) \) where terms \( l \) and \( r \) are of sort \( \text{State} \), and where the subterms \( u(t) \) and \( u'(t) \) are thread expressions of sort \( \text{Thread} \), and \( t \) is variable ranging over thread identifiers of sort \( \text{Tid} \). Note that based on the assumptions we made (see Section 3.1), there is going to be exactly one such thread expression \( u(t) \) on either side of a rule. We also assume that the equations in \( E_L \) are thread-preserving, that is, in any two state expressions equated by \( E_L \) both must have the same number of thread expressions and there is a bijective correspondence between such thread expressions preserving their thread identifiers.

We define \( \widehat{\Sigma}_L \) by adding fresh new sorts: \( M\text{State} \) and \( M\text{Thread} \). A new constructor enabled : \( \text{Thread} \ 	ext{Bool} \rightarrow \text{MThread} \) is introduced for the sort \( \text{MThread} \) to instrument threads with this additional flag that allow us to mark them as enabled or not for the next execution step. The use of the sort \( \text{Thread} \) in all state constructors is everywhere replaced by the sort \( \text{MThread} \). We also add two unary operators \( \{ \}, [\_] : \text{State} \rightarrow \text{MState} \). The equations in \( \widehat{E}_L \) are systematically derived from those in \( E_L \) by replacing in each equation in \( E_L \) each occurrence of a thread expression \( u(t) \) by the expression \( \text{enabled}(u(t), b_t) \), where \( b_t \) is a fresh new variable of sort \( \text{Bool} \) depending on \( t \). For every rewrite rule \( l(u(t)) \rightarrow r(u'(t)) \) if \( C \) in \( R_L \), the corresponding rewrite rule in \( \widehat{R}_L \) is then of the form \( \{Ct(l(\text{enabled}(u(t), true)))\} \rightarrow [Ct(r(\text{enabled}(u'(t), true)))] \) if \( \widehat{C} \), where \( Ct(.) \) is the context expression for the application of the rule in case \( r \) does not rewrite the entire state but only a state fragment\(^2\), and where \( \widehat{C} \) is the conjunction of equations obtained from \( C \) by changing each equation in \( C \) containing thread expressions as done in the definition of \( \widehat{E}_L \), and leaving all other equations untouched. Note that the use of the operators \( \{\_\}, [\_] \) in the rules in \( \widehat{R}_L \) means that in \( \widehat{R}_L \) only one-step rewrites are possible, since the operator \([\_]\) in the right-hand side blocks the application of any further rules.

As an example of the above transformation, consider the following rewrite

\(^2\) If the rule \( r \) rewrites the global state of the computation, the context \( Ct(.) \) is empty, i.e. \( Ct(l(u)) = l(u) \). We do however allow language specifications in which a rule \( r \) can be local to some fragment of the state. In this second case, it is important to make explicit a pattern \( Ct(.) \) for the context in which the rule is applied.
rule specifying the semantics of the \texttt{monitorenter} instruction of Java byte-code:

\[
\begin{align*}
& \text{rl} < T: \text{JavaThread} | \text{callStack}:([\text{PC, monitorenter, Pgm, ...}, (\text{REF}(K) \# \text{OperandStack}), ...] \text{CallStack}), ... > < O: \text{JavaObject} | \text{Addr}: K, ... > \Rightarrow < T: \text{JavaThread} | \text{callStack}: ([\text{PC} + 2, \text{Pgm}(\text{PC} + 2), \text{Pgm, ...}, \text{OperandStack}, ...] \text{CallStack}), ... > < O: \text{JavaObject} | \text{Addr}: K, ... , \text{Lock}: \text{Lock}(\text{OIL, T, 1}) > .
\end{align*}
\]

the transformed rewrite rule has the following form:

\[
\begin{align*}
& \text{rl} \{ \text{enabled}( < T: \text{JavaThread} | \text{callStack}:([\text{PC, monitorenter, Pgm, ...}, (\text{REF}(K) \# \text{OperandStack}), ...] \text{CallStack}), ... >, \text{true} ) < O: \text{JavaObject} | \text{Addr}: K, ... > \} \Rightarrow [ \text{enabled}( < T: \text{JavaThread} | \text{callStack}: ([\text{PC} + 2, \text{Pgm}(\text{PC} + 2), \text{Pgm, ...}, \text{OperandStack}, ...] \text{CallStack}), ... >, \text{true} ) < O: \text{JavaObject} | \text{Addr}: K, ... , \text{Lock}: \text{Lock}(\text{OIL, T, 1}) > ] .
\end{align*}
\]

The key point about the transformation \( R_L \mapsto \hat{R}_L \) is then:

**Proposition 3.1** The surjective projection \( \pi \) mapping terms of sort \( M\text{State} \) to terms of sort \( \text{State} \) defined by: (1) erasing the operators \( \{.,[]\} \), and (2) erasing the enabled operators, the corresponding flags and the context expression defines a one-step bisimulation between the corresponding rewrite theories.

That is, if we have a one-step rewrite \( u \rightarrow v \) with \( \hat{R}_L \), then we have also a corresponding one-step rewrite \( \pi(u) \rightarrow \pi(v) \) with \( R_L \); and conversely, if we have a one-step rewrite \( u' \rightarrow v' \) with \( R_L \), then we can find \( u \in \pi^{-1}(u') \) \( v \in \pi^{-1}(v') \) such that we have a one-step rewrite \( u \rightarrow v \) with \( \hat{R}_L \) (see \cite{10} for proof).

**The Partial Order Reduction Theory.**

In the second step, the theory \( \hat{R}_L = (\hat{\Sigma}_L, \hat{E}_L, \hat{R}_L) \) is transformed into \( R_{L+\text{POR}} = (\Sigma_{L+\text{POR}}, E_{L+\text{POR}}, R_{L+\text{POR}}) \) which adds to \( \hat{R}_L \) the partial order reduction module. Components of the transformed theory are defined based on the components of \( \hat{R}_L \) as follows:

- \( \Sigma_{L+\text{POR}} = \hat{\Sigma}_L \cup \Sigma_{\text{POR}} \cup \Sigma_{\text{AUX}} \), that is, the signature \( \hat{\Sigma}_L \) is extended with the signature \( \Sigma_{\text{POR}} \) of operators used in implementing the partial order heuristic algorithm, plus the signature of auxiliary operators \( \Sigma_{\text{AUX}} \) that are used for implementation purposes.

- \( E_{L+\text{POR}} = \hat{E}_L \cup E_{\text{POR}} \cup E_{\text{AUX}} \), that is, the set of equations \( \hat{E}_L \) are extended with the equations \( E_{\text{POR}} \) which specify the partial order heuristic algorithm, plus the equations \( E_{\text{AUX}} \) which define the auxiliary operators.

- \( R_{L+\text{POR}} = \hat{R}_L \cup \{ r_{\text{step}} \} \). In the case of the rewrite rules, only one new rewrite rule is added. We label this rule as \( r_{\text{step}} \). It is the only rule applicable to the new state, and therefore the only rule which will determine the transitions of the system at a given state.
The New State.

There is a new fresh sort \( PState \), as part of \( \Sigma_{POR} \), representing the new state of the system. A new sort \( StateInfoSet \) also belongs to \( \Sigma_{POR} \), capturing all the information necessary for the reduction algorithm (see Section 3.3). A new constructor operator \( \{\_\mid \_\} : MState \rightarrow PState \) is introduced for the new state. Therefore, a state in \( R_{PO} \) is a pair \( \{s\mid I\} \), where \( s \) is a state in \( \hat{R}_L \), and \( I \) is a term containing information necessary for the reduction algorithm.

The New Rule (step).

A single new conditional rule \( r_{step} \) in \( R_{L+POR} \) simulates one step rewrites of the original system:

\[
\text{step} : \{s\mid I\} \rightarrow [s'\mid I] \quad \text{if} \quad s \rightarrow s' \wedge s \neq s'
\]

where \( s \) and \( s' \) are variables of sort \( MState \), and the operators \( \{\_\mid \_\} \) and \( [\_\mid \_] \) are state constructors for the sort \( PState \) and are frozen operators \([4]\), that is, no rewriting is allowed below these operators. \( I \) is a variable of sort \( StateInfoSet \). By using this single rewrite rule, only one rewrite at a time can happen, which changes the given state to one of its successor states. Since the resulting state is in \( [\_\mid \_] \) format, no rewrite rule is applicable to it anymore, until it is changed to the \( \{\_\mid \_\} \) format. This is the point at which the partial order heuristic algorithm is applied, using an equation that completes the effect of the above rule:

\[
[s \mid I] = \{\text{state}(\text{MarkAmples}(s, I)) \mid \text{stateInfo}(\text{MarkAmples}(s, I))\}. \quad (*)
\]

The partial order reduction is applied at state \( s \), using the information in \( I \), by means of a single operation \( \text{MarkAmples} \). This operation takes a pair of elements of sorts \( MState \) and \( StateInfoSet \) as an input, and returns a pair of the same sort. The \( \text{MarkAmples} \) operation computes the ample set for the current state and returns the state with the ample transitions marked as specified by the POR algorithm. It also returns an updated version of \( StateInfoSet \) (see the POR algorithm part of Section 3.3). In the next section, we discuss in detail how the \( \text{MarkAmples} \) operations is specified.

3.3 The Partial Order Reduction Module

This module performs two main tasks: (1) extracting the set of enabled transitions at a given state, and (2) finding an ample subset of these transitions.

First, we have to define a transition in this context. Having the rewriting semantics \( (\Sigma_L, E_L, R_L) \) of a concurrent programming language \( L \), one can view the initial state of the system (a program and its inputs) as a \( \Sigma_L \)-term \( t \) being rewritten by the equations \( E_L \) and the rewrite rules \( R_L \) of the specification.
In a state transition system, a given state $s$ has a set of immediate successor states $\{s_1, s_2, \ldots, s_k\}$, and each pair $(s, s_i)$ is an enabled transition from state $s$. In the rewriting semantics, state $s$ is a term, and the set of enabled transitions leading to successor states can be represented as a set of pairs $(r_i, p_j)$, where $r_i \in R_L$ and $p_j$ is a position in term $s$. In other words, if a certain rule $r_i : l(u) \rightarrow r(v)$ is enabled at a position $p_j$ in term $s$, then we have a transition from $s$ to its successor $s[l(u)\backslash r(v)]$.

In general a position $p$ can be any position in the term tree. However, in our special case of semantics of concurrent programming languages together with the general assumptions discussed in Section 3.1, a thread identifier will uniquely specify a position, since we have assumed that a single thread is involved in each rewrite. Therefore, a pair $(t_i, r_j)$ consisting of a thread identifier $t_i$ together with an applicable rule $r_j$ uniquely characterizes a transition. This gives us a considerable practical advantage; because when the algorithm decides on an ample subset of the transitions, it suffices to mark the corresponding threads as enabled (see Section 3.1), which makes it unnecessary for all the unmarked threads (transitions) to be explored. Note that in the transformed theory, although the only rule applied to the state of the system is the rule $\text{step}$, in fact an application of $\text{step}$ always simulates some rewrite rule $r_i$ from the original system, and it is that rule that we consider in the above pair.

### 3.3.1 Extracting Enabled Transitions

As discussed above, a transition is a pair $(t_i, r_j)$ of a thread identifier and a rewrite rule. We can add a third component $I_k$ to this tuple, which includes all the information about context (i.e., names of variables, functions, locks, ...). This information can later help resolving some dependencies between the transitions, which may result in fewer dependencies and possibly in a better reduction.

At a given state $s$, we have to find all pairs $(t_i, r_j : l(u) \rightarrow r(v))$ where the rewrite rule $r_j$ is enabled for the term $s$ at the position associated with the thread $t_i$. In other words, we have to go over all the rewrite rules $r_j \in R_L$ and find all the positions at which $r_j$ can be applied to the term $s$. To do this, we generate a new set of equations, based on the rewrite rules in $R_L$, with exactly one equation per rule in the following manner. Let us assume that a rewrite rule $r \in \hat{R}_L$ is of the following general form:

$$ r : \{l(u(t))\} \Rightarrow [r(u'(t))] \text{ if } C $$

where $u(t)$ and $u'(t)$ are subterms of sort $\text{Thread}$, $t$ is a variable of sort $\text{Tid}$, and $C$ is the rule’s condition. The corresponding equation for $r$ is then:

$$ \langle T_e, l(u(t)) \rangle = \langle T_e \cup \{< t, r, I >\}, l(u(t)) \rangle \text{ if } C \land T_e \cup \{< t, r, I >\} \neq T_e $$

where $T_e$ is a set that accumulates enabled transitions. Note that rewrite rules
in $\hat{R}_L$ are already modified to capture the context in which the corresponding original rule of $R_L$ would have been applied. Starting from the pair $< \emptyset, t_s >$, by applying all equations of the above form, we will converge to the pair $< T_e, t_s >$, where $T_e$ is the set of all enabled transitions.

Since the context information $I$ depends on the specific programming language $L$ and on the way the semantics of $L$ is defined, the $I$ component has to be left as a null constant when these equations are generated automatically based on the rules. However, a tool builder familiar with the language semantics can customize these equations to include whatever context information may be useful later. In our experience with several rewriting semantics for different programming languages, there are relatively few rewrite rules in the semantic definitions (that is, $E_L$ is much bigger than $R_L$), so this process is rather quick and easy.

### 3.3.2 Computing the Ample Set

#### Dependence Relation.

The Definition of a dependence relation between the transitions is required for computing the ample sets. The dependence relation is represented by the operator $\text{Dependence: Transition \ Transition \rightarrow \ Bool}$. Clearly, the dependence relation is different for different programming languages. Some common dependence properties can be shared by many programming languages, such as: “all the transitions in a single thread are interdependent”, which is expressed by the following equation:

$$\text{Dependence}(< t, r, I >, < t, r', I' >) = \text{true}$$

where $t$ is a variable ranging over thread identifiers, $r$ and $r'$ are variables ranging over rule names, and $I$ and $I'$ are variables ranging over context information.

In order to have the best possible reduction, the language specifier/tool builder should supply the definition of the dependence relation for the given language as a set of additional equations. The dependence relation can often be defined through a few equations, even for complicated languages. See Section 4 for the definition of the dependence relation for the Java bytecode. Note that, in general, since the dependence relation is defined by a set of equations (that can potentially be conditional) we can naturally support the case of conditional dependence as in [6,17].

#### The Heuristic Algorithm.

Since the core of the heuristic algorithm can be specified using a few equations, we have specified two different heuristics. Many additional optimizations for these heuristics and also other heuristics can likewise be specified with little effort (see Section 5), but they are beyond the scope of this work. Figure 1 shows both algorithms. Functions $C'_1$, $C_2$, and $C_3$ check the three
conditions discussed in the next section, returning true or false. These procedures are called at each state (see Section 2) to compute the ample set at that state. The algorithm on the left is a simpler version, which only considers ample sets including transitions of a single thread. The algorithm on the right extends the former to consider ample sets that can include transitions of more than one thread, which can result in a better reduction. If we have \( n \) threads, and at some point no single thread can be a candidate for ample, we may be able to find a subset of threads that can satisfy the conditions as a whole. To do so, we use the transitive closure of the dependence relation \( D \) defined on the set \( T \) of transitions as follows:

\[
D : T \times T \rightarrow \{true, false\} \\
\text{where } c_{D,S}(T) \text{ computes all the transitions of } S \text{ which are immediately dependent on transitions in } T. \text{ Since } S \text{ is a finite set of transitions, } c_{D,S} \text{ is monotonic; if we reapply } c_{D,S} \text{ repeatedly, we eventually reach a set } T \text{ (a fixpoint) where } c_{D,S}(T) = T. \text{ The function } \mu_{c_{D,T_e}}(t) \text{ represents this fixpoint. The set } \mu_{c_{D,T_e}}(t) \text{ is a good candidate for an ample set, since we know that at least no transition outside the set } \mu_{c_{D,T_e}}(t) \text{ is dependent on anything inside it. A good method to find the best ample set is to sort the sets } \mu_{c_{D,T_e}}(t), \text{ for all } t \in T_e \text{ based on }
\]

\[
1 \text{ Take a transition } t \text{ from } T_e. \\
2 \text{ Let } T_a = tr(t). \\
3 \text{ If } C_1'(T_a) \text{ and } C_2(T_a, P) \text{ and } C_3(T_a). \\
4 \text{ then } \\
\text{ mark thread of } t \text{ as ample.} \\
\text{ quit.} \\
5 \text{ else } \\
\text{ go to step 1.} \\
6 \text{ Mark all threads as ample.} \\
\]

\[
1 \text{ Take a transition } t \text{ from } T_e. \\
2 \text{ Let } T_a = tr(t). \\
3 \text{ Let } S = \mu_{\mu_{c_{D,T_e}}(T_a)}. \\
4 \text{ If } C_1'(S) \text{ and } C_2(S, P) \text{ and } C_3(S). \\
5 \text{ then } \\
\text{ mark thread of } t \text{ as ample.} \\
\text{ quit.} \\
6 \text{ else } \\
\text{ go to step 1.} \\
7 \text{ Mark all threads as ample.} \\
\]

Fig. 1. Two Partial Order Reduction Heuristics.
their cardinality, and then start checking the conditions, beginning with the smallest one. This way, if we verify all the conditions for a candidate set, we are sure that it is the smallest possible ample set, and we are done.

3.3.3 Checking The Conditions.

The most involved part of the partial order reduction algorithm is checking the conditions in [2]. Conditions C2 and C3 are exactly the same as in [2]. Condition C′1 is a stronger version (see [10]) of condition C1 from [2] (since the original C1 from the POR theory is not locally verifiable) and very similar to the variation of it in the heuristic proposed in [2]. Since the algorithm always works on nonempty sets, we are left to check three out of the four conditions. Here, we describe how the conditions are checked for a candidate set of transitions (ample set). The special case of a single transition as a candidate (as in [2]) follows from this easily.

\( T_c \) represents the set of all enabled transitions in the current state. Note that, as argued before, the notions of transition and of enabled thread are equivalent in our framework, so we often switch between the two.

\[ \text{C′1: if transition set } T \subset T_e \text{ is a an ample set, then no thread in } T_e - T \text{ should have a transition in the future that is dependent on } t. \]

To compute future transitions of a thread \( t_i \in T_e - T \), a conservative flow-insensitive context-insensitive static analysis of the code is performed. This kind of static analysis can be done locally, and is different for different programming languages. Therefore, the language specifier/tool builder needs to provide it. In the definition of the algorithm we assume that there is an operation \( \text{ThreadTransitions} \) which takes the thread identifier and the current state of the system and returns all the future transitions of the thread in the form of a set of tuples (transition format) through a purely static analysis of the code of the input program which usually offers an overestimation of the actual set. Having the future transitions of all the threads in \( T_e - T \), condition C′1 can then be easily checked by using the dependence relation. To see that C′1 implies C1 in [2], see [10].

\[ \text{C2: ample transitions should be invisible if the state is not fully expanded.} \]

This condition is the simplest of the three to verify. The set of propositions used in the desired property is given as an input. The check just has to go over this set, element by element, and check whether each proposition has the same truth value in state \( s \) and in its successor state with respect to all transitions in the ample candidate set.
C3: Cycle-closeness Condition.

This condition ensures that no transition is enabled over a cycle in the state transition graph and is never taken in the ample set. This condition can be easily checked when the partial order reduction algorithm is embedded in a model checker, since the stack of states being explored is available. In our case, we use exactly the same method, but we simulate part of that stack as part of the state. The second component of the new system state, \textit{StateInfoSet} takes care of this. Whenever in a state \(s\) there is a transition \(t\) outside the ample set, the pair \((t, s)\) will be stored in the \textit{StateInfoSet} component. As soon as a transition is taken in some future step, the pair is removed from the \textit{StateInfoSet}. If a pair \((t, s)\) is still there when we revisit \(s\), we know that we are closing a cycle, so we must take the transition.

3.4 Correctness of the Theory Transformation

The correctness of our theory transformation can be now stated as the following theorem, whose proof is sketched in \([10]\):

\[\textbf{Theorem 3.2} \] Assishing that a set \(AP\) of atomic state predicates has already been added to \(R_L\) by means of a set of equational definitions, the Kripke structures associated to the rewrite theories \(R_L\) (with \textit{State} as its sort of states) and to \(R_{L+POR}\) (with \textit{PorState} as its sort of states) are stuttering bisimilar.

4 Applications of the Method and Experiments

We have implemented the theory transformation for our generic POR reduction method in a Maude \([4]\) prototype and have used it to build POR units for Java bytecode and for a Promela-like language. In this section we illustrate how the method was used to build the POR unit for Java bytecode, which has been added to JavaFAN \([11]\), a tool to formally analyze Java programs based on a rewriting semantics of both Java source code and bytecode. We also present some performance figures for both the JVM and the Promela-like language to show that the generic partial order module can result in drastic reductions in the state space of programs in the above languages.

4.1 The JVM POR Unit

By briefly discussing this example, we illustrate how the language-dependent parts are defined in Maude for the Java bytecode semantics to give a better understanding of these parts, and also to show that they can be specified by the tool builder with relatively little effort and in a program-independent way.

\textit{Extracting Transitions.}

There are 16 equations, corresponding to the 16 rewrite rules in the semantics of the Java bytecode, which extract all the enabled transitions from a given state. Here is an example of one of these equations:
Farzan and Meseguer

\[ \text{ceq} \triangleleft \text{S}, \triangleleft \text{T}: \text{JavaThread} | \text{callStack}: ([\text{PC}, \text{monitorenter}, ..., (\text{REF}(\text{K}) \# \text{OperandStack}), ...] \text{CallStack}), ... \triangleright < \text{O}: \text{JavaObject}|\text{Addr: K, ...}, \text{Lock: Lock(OIL, NoThread, 0)} \triangleright \text{Ct} \triangleright = \text{ceq} \{\text{MONITORENTER, T, noInfo}\}, \< \text{T}: \text{JavaThread} | \text{callStack}: ([\text{PC}, \text{monitorenter}, ..., \text{OperandStack}, ...] \text{CallStack}), \text{Status: scheduled, ...} \triangleright < \text{O}: \text{JavaObject} | \text{Addr: K, ...}, \text{Lock: Lock(OIL, NoThread, 0)} \triangleright \text{Ct} \triangleright \text{if S} \{\text{MONITORENTER, T, noInfo}\} =/= \text{S} . \]

where \( S \) is the enabled transitions set. The equation says that if in the current state (containing a thread \( T \), an object \( O \), and a context \( Ct \) which captures the rest of the JVM state that is a multiset), \( T \) is ready to execute a \text{monitorenter} (lock) instruction, and \( O \) is not locked by any other thread, it means that the tuple \( \{\text{MONITORENTER, T, noInfo}\} \) is an enabled transition, and it is added to the set \( S \) if it is not already in it.

\textit{Dependence Relation.}

The dependence relation for Java bytecode is defined based on the following facts: (1) two accesses to the same location are dependent if at least one of them is a write. This is defined through a few equations to cover the access to the instance fields as well as static fields; (2) two lock operations accessing the same lock are dependent. This is defined through a few equations to cover synchronized method calls, the \text{monitorenter} instruction, as well as the \text{notifyAll} built-in method of Java.

As an example of equations defining the dependence relation we have:

\[ \text{eq Dependence} \{(T, 'PutField, I), \{T', 'GetField, I'}\) = true . \]
\[ \text{eq Dependence} \{(T, 'InvokeStatic, C), \{T', 'InvokeStatic, C'} = true . \]

which specify that a read and a write to an instance field (first line) are always dependent, and (second line) two synchronized static method calls are dependent if they are locking the same class, \( C \).

\textit{Thread Transitions.}

As mentioned at the end of Section 3.3, to check condition \( C'1 \), the operation \text{ThreadTransitions}, which conservatively computes the set of future transitions of a thread, has to be specified by the user. In the case of Java bytecode the idea is to start from the current point in \( t_i \) and add all the future instructions (transition steps) of the current method executing, and upon a method call, add in all the instructions (transitions) of the code of that method as well (avoiding repetition). This is conservative, in the sense that in the cases where more than one method can be the potential resolution of a call site, all of them are considered, and also in transitions such as reading/writing a field of an object where the object cannot be resolved until the point of execution, conservatively all possible objects will be considered.
4.2 Experiments

Table 1 presents the reduction comparison of sieve of eratosthenes modeled in Promela language and Maude. The first column shows the result for the Promela program model checked with SPIN. The second column contains the result for the same model written in Maude language together with our POR unit. The third column shows the result of the same Promela program when it is model checked using the semantics of Promela together with our POR unit in Maude. The overhead of interpreting Promela in this case results in a larger number of states.

<table>
<thead>
<tr>
<th></th>
<th>Promela–SPIN</th>
<th>Maude–Ours</th>
<th>Promela–Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No Reduction</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>States</td>
<td>703</td>
<td>130</td>
<td>61,842</td>
</tr>
<tr>
<td>Time</td>
<td>0.4s</td>
<td>0.2s</td>
<td>41s</td>
</tr>
<tr>
<td><strong>Reduction</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>States</td>
<td>114</td>
<td>26</td>
<td>174</td>
</tr>
<tr>
<td>Time</td>
<td>0.06s</td>
<td>0.01s</td>
<td>0.3s</td>
</tr>
</tbody>
</table>

Table 1
Time and Space Reduction Comparisons.

Table 2 shows the results of time/space reduction for a deadlock-free version of dining philosophers with different number of philosophers in the Promela-like language. Entries left empty indicate that we could not model check the example on our platform, a PC running Linux with a 2.4GHz processor and 4GB of memory.

Table 3 illustrates a dining philosophers program (5 philosophers) model

<table>
<thead>
<tr>
<th>Program</th>
<th>Reduction</th>
<th>Time</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP(5)</td>
<td>No</td>
<td>25.1s</td>
<td>56,212</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>7.3s</td>
<td>3,033</td>
</tr>
<tr>
<td>DP(6)</td>
<td>No</td>
<td>146.2.0s</td>
<td>623,644</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>30.0s</td>
<td>22,822</td>
</tr>
<tr>
<td>DP(7)</td>
<td>No</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>5m</td>
<td>168,565</td>
</tr>
<tr>
<td>DP(8)</td>
<td>No</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>66m</td>
<td>1,412,908</td>
</tr>
</tbody>
</table>

Table 2
Dining Philosophers.
checked in JavaFAN, where two versions of the dependency relation are compared. In the “basic” version, the dependency relation is the general version (presented in Section 4) that holds for all Java programs. The “NotShared” version lifts the dependencies of read/write memory accesses, since we know that the dining philosophers code does not use any shared memory and works merely based on locks. As shown in Table 3, a simple change like this (which means commenting out a few equations in the definition of the dependency relation) can result in a considerably better performance.

<table>
<thead>
<tr>
<th>Test</th>
<th>Basic(t)</th>
<th>Basic(n)</th>
<th>NotShared(t)</th>
<th>NotShared(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dining Philosophers</td>
<td>7m</td>
<td>6991</td>
<td>41s</td>
<td>2690</td>
</tr>
</tbody>
</table>

Table 3
Changing Dependency Relation.

Table 4 shows the state reduction obtained when the partial order reduction module is used. The JavaFAN tool reduces the number of states substantially by itself, since it uses the rewrite rules to model only the concurrent parts of Java (see [11] for details). But, the partial order reduction can still add a substantial reduction to that. PL is a two stage pipeline, DP is a deadlock-free version of the dining philosophers, RA is NASA’s remote agent benchmark, and SE is a distributed sieve of Eratosthenes. All programs in these experiments, as well as the semantic definitions of the JVM and the Promela-like language and their POR-transformations by our method are available in [8].

<table>
<thead>
<tr>
<th>Test</th>
<th>States (w POR)</th>
<th>States(wo POR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL</td>
<td>6612</td>
<td>18074</td>
</tr>
<tr>
<td>DP(5)</td>
<td>6991</td>
<td>16248</td>
</tr>
<tr>
<td>RA</td>
<td>24</td>
<td>33</td>
</tr>
<tr>
<td>SE</td>
<td>186</td>
<td>247</td>
</tr>
</tbody>
</table>

Table 4
Partial Order Reduction Results.

5 Related Work and Conclusions

Related Work.

There are two well-known approaches to attack the state-explosion problem while model checking. The first approach consists of partial order methods introduced by Peled in [22]. The generic method proposed in this paper fits within this approach. Several different variations [14,15,25,1,12,3,18] of the POR approach have been introduced since.
A first class of POR methods—including the stubborn sets method of [25], the persistent sets method of [16], and the ample sets method of [23]—are based on modifying the search algorithm and applying the reduction dynamically. [12] takes the matter even further, and dynamically tracks the interactions between threads based on initially exploring an arbitrary interleaving of them. Details of the reduction heuristic are orthogonal to our method; although we propose two different heuristics in this paper, many other heuristics can be implemented with little effort. A second class of POR methods such as the one in [18] use a static approach in which all partial order reduction information is computed statically, and then an already reduced model is generated to be model checked.

In the dynamic methods, one has to alter the existing model checker to include the reduction, while static methods suffer from the fact that only a limited amount of information is available at compile time. We believe that our method addresses both problems: it can work with an existing model checker, so it has the advantages of the static methods, but it applies the reduction dynamically and therefore can benefit from the runtime information.

It seems fair to say that current POR-enabled model checkers are mostly language-specific, or, by using for example a static approach such as [18], achieve only a limited “genericity by translation into a common intermediate language”. Tools such as Verisoft [15] that can monitor and control the execution of programs in different languages for model checking purposes are in practice applied to a limited family of languages and cannot benefit from any optimizations that can potentially be introduced using some sort of static analysis of the program, which is not the case in our method. To the best of our knowledge this work represents the first attempt to develop LTL model checkers with POR capabilities for concurrent languages in a generic way using their semantic definitions.

Besides the POR methods, a second state space reduction approach, which could be called transaction-based, consists of more recent techniques that consider various kinds of exclusive access predicates for shared variables specifying some synchronization disciplines [24,13,6]. These predicates can be used to reduce the search space during the state space explorations. The POR techniques (including the method proposed in this paper) are complementary to these other methods. We discussed how our method exploits some ideas from [24] in Section 2.1. We strongly believe that the reductions in [13] can be achieved using a very similar method to that presented in this paper (see below for more details).

**Conclusions.**

We have presented a general method to make software model checkers with POR capabilities language-independent, so that they can be specialized to any programming language $L$ of interest. Our method is based on a theory transformation of the rewriting logic formal semantics of the given language.
The language specialization can be done semi-automatically and with relatively little effort by a tool builder. Language-specific optimizations can also be added, because the heuristic algorithm and the dependence relation are explicit parameters of the theory transformation. Since all POR computations are performed in the transformed theory itself, the method does not require any modifications to the underlying LTL model checker. Our experience evaluating this method in practice for the JVM, a Promela-like language, and Maude, indicates that significant state space reductions and time speedups can be gained.

Our method is also generic at the semantic framework level: we have developed it in detail within rewriting logic, but the same idea can be applied within other frameworks, for example SOS. In any such framework, the semantics of $L$ will have a specification $S_L$. We then would obtain the POR capabilities by a suitable theory transformation $S_L \mapsto S_{L+POR}$.

The current prototype implementation of our method does not support various well-known optimization strategies, but many of these can be incorporated into our framework in a straightforward way. These strategies are often based on assumptions about the structure of the programming language under consideration. Therefore, they belong to the second, language-specific customization phase of our theory transformation, although in some cases they can be applied to entire families of languages. For example, a reduction strategy proposed in [6] for concurrent object oriented software is detecting heap objects that are thread-local to sharpen the dependence relation. All the static/dynamic analysis in [6] that leads to detecting the thread locality is possible in our framework, since we have both the static and dynamic information available. A more extensive experimentation with a broader set of language instantiations and incorporating the above optimizations should be performed in the future. Furthermore, the mechanical verification of the correctness of our theory transformation along the lines of the proof sketched in [10] should be investigated.

Another interesting direction for future work is extending our generic method beyond POR to also support what we have called “transaction-based reductions” in Section 1. Such reductions are complementary to those obtained by POR methods. We conjecture that a similar theory transformation would allow us to achieve transaction-based reductions in a generic way. The equation (*) in Section 3.2 works as a nondeterministic scheduler which in the present method schedules all the threads belonging to the ample set for the next step. In a transaction-based method the role currently played by the MarkAmples operation could instead schedule a single thread $t$, provided $t$ is inside a transaction, and the component $I$ could then be used for the instrumentation predicates.
References


D. Peled. All from one, one for all: on model checking using representatives. In CAV’93, LNCS, pages 409–423, 1993.


A Rewrite Framework for Language Definitions and for Generation of Efficient Interpreters

Mark Hills, Traian Şerbănuţă, Grigore Roşu

Abstract
A rewrite logic semantic definitional framework for programming languages is introduced, called $K$, together with partially automated translations of $K$ language definitions into rewriting logic and into C. The framework is exemplified by defining SILF, a simple imperative language with functions. The translation of $K$ definitions into rewriting logic enables the use of the various analysis tools developed for rewrite logic specifications, while the translation into C allows for very efficient interpreters. A suite of tests show the performance of interpreters compiled from $K$ definitions.

Key words: programming languages, rewriting logic, language interpreters.

1 Introduction
The $K$ language definition framework [9] is a rewrite logic based framework for specifying programming languages. It includes both a notation, the $K$-notation, consisting of a series of domain-specific syntactic-sugar conventions aiming at simplifying and enhancing readability of language definitions, and a language definition technique, the $K$-technique, based on a first-order representation of continuations. As part of our ongoing research, we are developing a number of tools around $K$ to assist in defining and analyzing programming languages.

Here, we show two pieces of this work. First, we show the semantics of a simple programming language with functions defined using $K$. This language has standard imperative features, including a controlled jump in the form of...
a function return. Second, we provide some details of a translation from our notation in \( K \) to an interpreter for the language, written in C. We are actively working on providing for the automated construction of interpreters from \( K \) definitions of languages, and currently have a semi-automated translation.

In Section 2, we present an overview of the \( K \) notation together with details of how it can be translated into rewrite logic. In Section 3 we show \( K \) at work by defining the Simple Imperative Language with Functions, or \( \text{SILF} \). In Section 4 we provide details of our translation from \( K \) to C, including some initial performance figures of comparisons with equivalent programs written in other languages. Section 4 discusses related work, while Section 5 discusses future work and concludes the paper.

2 The \( K \) Language Definition Framework

Here we briefly recall the \( K \)-framework \cite{9}, useful to compactly, modularly and intuitively define languages in rewrite logic. It consists of the \( K \)-notation, i.e., a series of notational conventions for matching modulo axioms, for eliding unnecessary variables, for sort inference, and for context transformers, and of the \( K \)-technique, which is a continuation-based technique to define languages algebraically. The \( K \)-framework is described in detail in \cite{9}.

Matching Modulo. Despite its general intractability \cite{3}, matching modulo Associativity, Commutativity, and Identity, or \( ACI \)-matching, tends to be relatively efficient in practice. Many rewrite engines support it in its full generality. \( ACI \)-matching leads to compact and elegant, yet efficiently executable specifications. Different languages have different ways to state that binary operations are associative and/or commutative and/or have identities; to keep the discussion generic, we assume that all \( ACI \) operations are written using the \textit{mixfix} concatenation notation \( \_ \_ \) and have identity \( \_ \) , while all but one\(^4 \) of the \( AI \) operations use the comma notation \( \_ \_ \_ \) and have identity written also \( \_ \_ \_ \). In particular implementations of \( K \) specifications, to avoid confusion one may want to use different names for the different \( ACI \) or \( AI \) operations. \( ACI \) operations correspond to multi-sets, while the \( AI \) operations correspond to lists. Therefore, for any sort \( \text{Sort} \), we tacitly add supersorts \( \text{SortSet} \), \( \text{SortNeSet} \), \( \text{SortList} \), and \( \text{SortNeList} \) of \( \text{Sort} \) (with the \( \text{Ne} \) versions being non-empty), constant operations \( \_ : \rightarrow \text{SortSet} \) and \( \_ : \rightarrow \text{SortList} \), and \( ACI \) operation \( \_ \_ : \text{SortSet} \times \text{SortSet} \rightarrow \text{SortSet} \) and \( AI \) operation \( \_ \_ : \text{SortList} \times \text{SortList} \rightarrow \text{SortList} \) both with identities \( \_ \) .

\( ACI \) operations will be used to define states as “soups” of attributes; e.g., the state of a language can be a “soup” containing a store, locks which are busy, input/output buffers, etc., as well as a set of threads. Soups can be nested; for example, a thread may contain itself a soup of thread attributes, such as an

\(^4 \) The exception to the comma notation for \( AI \) operations will be the “continuation”; defined later, it will follow, just for ease of reading, the notation \( \_ \rightarrow \_ \).

77
environment, a set of locks that it holds, several stacks (for functions, exceptions, loops, etc.); an environment is further a soup of pairs (name, location), etc. Lists will be used to specify structures where the order of the attributes matters, such as buffers (for input/output), parameters of functions, etc.

For example, let us define an operation \( \text{update} : \text{Environment} \times \text{Name} \times \text{Location} \rightarrow \text{Environment} \), where \( \text{Environment} \) is the set sort \( \text{NameLocationSet} \) associated to a pairing sort \( \text{NameLocation} \) with one constructor pairing operation \( (\_, \_) : \text{Name} \times \text{Location} \rightarrow \text{NameLocation} \). \( \text{update} \) is the same as \( \text{Env} \) except in the location of \( X \), which should be replaced by \( L \):

\[
(\forall X : \text{Name}; L, L' : \text{Location}; \text{Env}: \text{Environment}) \quad \text{update}((X, L') \ \text{Env}, X, L) = (X, L') \ \text{Env}.
\]

The ACI-matching algorithm “knows” that the first argument of \( \text{update} \) has an ACI constructor, so it will be able to match the lhs of this equation even though the pair \( (X, L') \) does not appear on the first position in the environment.

**Sort Inference.** Surprisingly, the variable declarations part of the equation of \( \text{update} \) takes almost half the size of the sentence. It is often the case in our experiments with defining languages in Maude that variable declarations take a significant amount of space, sometimes more than half the entire language specification. However, in most cases the sorts of variables can be automatically inferred from the context. To simplify this process, we assume that all variable names start with a capital letter. Consider, e.g., the two terms of the equation above, \( \text{update}((X, L') \ \text{Env}, X, L) \) and \( (X, L) \ \text{Env} \). Since the arity of \( \text{update} \) is \( \text{Environment} \times \text{Name} \times \text{Location} \rightarrow \text{Environment} \), one can immediately infer that the sorts of \( X \) and \( L \) are \( \text{Name} \) and \( \text{Location} \), respectively. Further, since the first argument of \( \text{update} \) has the sort \( \text{Environment} \) and since environments are constructed using the operation \( \_ \_ : \text{Environment} \times \text{Environment} \rightarrow \text{Environment} \), one can infer that the sort of \( \text{Env} \) is \( \text{Environment} \).

Because of subsorting, a variable occurring on a position in a term may have multiple sorts. For example, the variable \( \text{Env} \) above can have both the sort \( \text{Environment} \) (which aliases \( \text{NameLocationSet} \)) and the sort \( \text{NameLocation} \). The report [9] discusses in more depth the subtleties of sort inference in the presence of subsorting. Here we only recall that if an occurrence of a variable can have multiple sorts, we assume by default, or by convention, that that variable occurrence has *the largest* sort among those that it can have; this convention corresponds to the intuition that we assume the “least” information about each variable occurrence. If the same variable appears on multiple positions then we infer for that variable the “most concrete” sort that it can have among them. Technically, this is the intersection of all the largest sorts inferred for that variable on the different positions where it appears. If the variable sort-inference process is ambiguous, or if one is not sure, or if one really wants a different sort than the inferred one, or even simply for clarity, one is given the possibility to sort variables “on-the-fly”: we append the
sort to the variable using “:”, e.g., $X : \text{Sort}$. For example, from the term $\text{update(Env, X, L)}$ one can only infer that the sort of $\text{Env}$ is $\text{Environment}$, the most general possible under the circumstances. If for any reason one wants to refer to a “special” environment of just one pair, then one can write $\text{update(Env:NameLocation, X, L)}$.

Underscore Variables and Tuples. With the sort inference conventions, the equation defining the operation $\text{update}$ can be therefore written as

$$\text{update}((X, L') \text{ Env, X, L}) = (X, L) \text{ Env}.$$  

Note that the location $L'$ that occurs in the lhs is not needed; it is only used for “structural” purposes, i.e., it is there only to say that the name $X$ is allocated at some location, but we do not care what that location is (we change it anyway). Since this will be a common phenomenon in our language definitions, we take the liberty to replace unnecessary letter variables by underscores, like in Prolog. Therefore, the equation above can be written

$$\text{update}((X, _) \text{ Env, X, L}) = (X, L) \text{ Env}.$$  

Like we need to pair names and locations to create environments, we will often need to tuple two or more terms in order to “save” current information for later processing. In $K$, by convention we allow all tupling operations without defining them explicitly. Like the sorts of variables, their arities can also be inferred from the context. Concretely, if the term $(X_1 : \text{Sort1}, X_2 : \text{Sort2}, \ldots, X_n : \text{Sortn})$ appears in some context (the variable sorts may be inferred), then we implicitly add to the signature the sort $\text{Sort1Sort2...Sortn}$ and the operation $(\_, \_, \ldots, \_) : \text{Sort1} \times \text{Sort2} \times \cdots \times \text{Sortn} \rightarrow \text{Sort1Sort2...Sortn}$.

Contextual Notation for Rewrite Rules. All the subsequent rewrite rules will apply on just one (large) term, encoding the state of the program. Specifically, most of them will apply on subterms selected via matching, but only if the structure of the state permits it. In other words, most of our rules will be of the form $C[t_1] \cdots [t_n] \rightarrow C[t'_1] \cdots [t'_n]$, where $C$ is some context term with $n \geq 0$ “holes” and $t_1, \ldots, t_n$ are subterms that need to be replaced by $t'_1, \ldots, t'_n$ in that context. $C$ needs not match the entire state, but nevertheless sometimes it can be quite large. To simplify notation and ease reading, in $K$ we write rules as $C[t_1] \cdots [t_n]$. This notation follows a natural intuition: first write the state context in which the transformation is intended to take place, then underline what needs to change, then write the changes under the line. Our contextual notation above proves to be particularly useful when combined with the “_” variables: if “_” appears in a context $C$, then it means that we do not care what is there but that we do not change it either.

Matching Prefixes, Suffixes and Fragments. We here introduce one more piece of notation that will help us further compact our language definitions by eliminating the need to mention unnecessary underscore variables. Many state attribute “soups” will be wrapped with specific operators
to keep them distinct from other soups. For example, environments will be
wrapped with an operation $env : \text{Environment} \rightarrow \text{Attribute}$ before they are
placed in their threads' state attribute soup. Thus, if we want to find the
location of a name $X$ in the environment, then we match the environment
attribute against the “pattern” term $env((X, L) \_)$ and thus find the desired
location $L$; the underscore variable matches the rest of the environment. The
underscores make pattern terms look heavier and harder to read than needed,
especially when the state is defined using deeply nested soups of attributes
(not the case in this paper). What one really wants to say above is that one
is interested in the pair $(X, L)$ that appears somewhere in the environment.
In our particular domain of language definitions, we believe, subjectively, that
the notation $env((X, L))$ for the same pattern term is better than the one
using the underscores. By convention, whenever “$\_ \circ \_”$ is an ACI or AI operator
wrapped by some attribute operator, say $att$, we write

$$att(T) \text{ (i.e., \text{left parenthesis right angle}) as syntactic sugar for}\ att(T \circ \_),$$
$$att(T) \text{ (i.e., \text{left angle right parenthesis}) as syntactic sugar for}\ att(\_ \circ T),$$
$$att(T) \text{ (i.e., \text{left and right angles}) as syntactic sugar for}\ att(\_ \circ T \circ \_).$$

If “$\_ \circ \_”$ is an ACI operator then the three notations above have the same effect,
namely that of matching $T$ inside the soup wrapped by $att$; for simplicity, in
this case we just use the third notation, $att\langle T \rangle$. The intuition for this notation
comes from the fact that the left and the right angles can be regarded as some
hybrid between corresponding “directions” and parentheses. For example, if
“$\_ \circ \_”$ is AI (not C) then $\langle T \rangle$ can be thought of as a list starting with $T$
(the left parenthesis) and continuing however it wishes (the right angle); in
other words, it says that $T$ is the \text{prefix} of the list wrapped by the attribute
$att$. Similarly, $\langle T \rangle$ says that $T$ is a \text{suffix} and $\langle T \rangle$ says that $T$ is a contiguous
\text{fragment} within the list wrapped by $att$. If “$\_ \circ \_”$ is also commutative, i.e., an
ACI operator, then the notions of prefix, suffix and fragment are equivalent,
all saying that $T$ is a subset of the set wrapped by $att$.

This notational convention will be particularly useful in combination with
other conventions part of the $K$ notation. For example, the input and output
of the programming language defined in the sequel will be modeled as comma
separated lists of integers, using an AI binary operation “$\_, \_”$ of identity \text{“\_”};
then in order to read (consume) the next two integers $N_1, N_2$ from the input
buffer, or to output (produce) integers $N_1, N_2$ to the output buffer, all one
needs to do (as part of a larger context that we do not mention here) is:

$$\text{in}(N_1, N_2) \quad \text{and, respectively,} \quad \text{out}(\cdot)$$

$$\cdot \quad N_1, N_2$$

The first matches the first two integers in the buffer and removes them (the "\_."
underneath the line), while the second matches the end of the buffer (the "\_."
above the line) and appends the two integers there. Note that the later works
because of the matching modulo identity: $\text{out}(\cdot)$ is a shorthand for $\text{out}(\_, \cdot)$,
where the underscore matches the entire list; replacing \text{“\_.”} by the list $N_1, N_2$.
is nothing but appending the two integers to the end of the list wrapped by \textit{out}. As another interesting example, this time using an ACI operator, consider changing the location of an identifier $I$ in the environment to another location, say $L$; this could be necessary in the definition of a language allowing declarations of local variables, when a variable with the same identifier, $I$, is declared locally and thus “shadows” a previously declared variable with the same name. This can be done as follows (part of a larger context): $\text{env}(I, \ _ )$.

\textbf{Context Transformers} are the most subtle aspect of the $K$ notation, based on the observation that, in programming language definitions, it is always the case that the state of the program does not change its significant structure during the execution of the program. For example, the store will always stay at the same level in the state structure, typically at the top level. If certain state infrastructure is known to stay unchanged during the evaluation of any program, and if one is interested in certain attributes that can be unambiguously located in that state infrastructure, then we only mention those attributes as part of the context assuming that the remaining part of the context can be generated automatically (statically). Since SILF does not have threads, exceptions or other complex control sensitive language features, context transformers do not make a difference in this paper, so we do not discuss them in more detail. The reader interested in the role of context transformers in compactness and modularity of language definitions is referred to [9].

\textbf{Translating $K$ to Maude.} We currently perform the translation from $K$ rules to Maude[1] by hand, with ongoing work on an automated translation. As an example, consider the rule shown below, which is for function application:

\[
\begin{array}{c}
\text{k}(\text{val}(_ ) \rightarrow \text{apply}(I) \rightarrow K) \rightarrow \text{fstack}(\ _ , \ _ ) \rightarrow \text{env}(\ _ ) \rightarrow \text{fenv}(I, K') \rightarrow \text{genv}(GEnv) \rightarrow K' \rightarrow (\text{Env}, K) \rightarrow GEnv
\end{array}
\]

In words, this rule states that, to apply the function with identifier $I$ to a (possibly empty) list of values, we need to replace the \textit{apply} continuation item and the continuation $K$ with the continuation $K'$ associated with the function $I$ in the function environment, put $K$ and environment $\text{Env}$ on a stack, and replace $\text{Env}$ with the global environment $GEnv$, which will give us access to global names while hiding names declared in the calling context. We make use of many of the conventions we discussed in this section within this rule. For instance, the values are unnamed since we do not use them at this point. Also, since the stack is an associative list, we are adding something to the head of the list by replacing the identity on the left with the item we are stacking, a tuple. The function environment is a set, so we match against the function name to get the proper tuple in the set without the need to specify the rest of the set. We need only mark those parts of the state that are changing by putting the changes under what is being changed; the parts of the state that
remain the same need no further notation.

For comparison, here is the Maude equation for this rule, including variable declarations. The same variable names have been used as above for variables appearing in both:

\[
\begin{align*}
\text{var } I & : \text{Id} . \ \text{vars } K, K' : \text{Continuation} . \\
\text{var } ICS & : <\text{Id}><\text{Continuation}>\text{Set} . \ \text{var } Vl : \text{ValueList} . \\
\text{var } ECL & : <<\text{Id}><\text{Location}>\text{Set}><\text{Continuation}>\text{List} . \\
\text{vars Env, GEnv} & : <\text{Id}><\text{Location}>\text{Set} .
\end{align*}
\]

\[
\begin{align*}
\text{eq } k(\text{val}(Vl) \rightarrow \text{apply}(I) \rightarrow K) & \ fstack(ECL) \ \text{env}(Env) \ \text{fenv}(ICS [I,K']) \\
\text{genv}(GEnv) & = \\
k(\text{val}(Vl) \rightarrow K') & \ fstack([Env,K], ECL) \ \text{env}(GEnv) \ \text{fenv}(ICS [I,K']) \\
\text{genv}(GEnv) .
\end{align*}
\]

Note here that we first need to declare a number of variables. Also, note that we need to name items that we are not concerned about, such as the list of values, and we need to include items mentioned on the left-hand side on the right-hand side as well, even if they do not change.

3 SILF: A Simple Imperative Language with Functions

Using the \( K \) notation, we now define a simple imperative language with functions, which we will herein refer to as SILF. The BNF syntax for SILF is shown in Figure 1. Note that a program is made up of an optional statement, which is assumed to be global variable declarations (not just any arbitrary statement), followed by one or more functions, one of which should be called \texttt{main}. We assume below that programs are well formed and type correct, and that we do not need to worry about issues such as precedence. We adopt the \texttt{mix-fix} notation for syntax in algebraic notation, with the standard conversion, adding a new sort for each non-terminal, and a new operation for each production. For instance, the declaration of a function will be:

\[
\text{function } \_ (_-) \begin{array}{c} \text{begin} \\
\end{array} \text{end} : \text{Id} \times \text{IdList} \times \text{Stmt} \rightarrow \text{FunDecl}.
\]

In the presentation of the rules below, vertical lines are occasionally used to separate rules on the same line (for instance, in the rules below for function return). These vertical lines have no semantic significance.

\textbf{State Infrastructure.} Since the rules in the semantics given below act on the SILF state, it is important to understand the state structure. The state of the program is made up of a number of “ingredients” in the state “soup”, in this case all at the top level. The continuation, indicated by \( k \), keeps track of the current control context. The \texttt{fstack} is the function stack, and holds information about the computation to resume on return – this is similar to a stack frame. The \texttt{env} and \texttt{genv} hold name to location mappings for the local and global environment, while the \texttt{fenv} holds mappings from function names to continuations for the bodies. The \texttt{store} holds location to value mappings. Input and output are represented by \texttt{in} and \texttt{out}, respectively. Finally, the next location in the store to allocate is tracked with \texttt{nextLoc}. This is represented
Formally, one declares the state structure by means of an algebraic signature, where each “ingredient” is wrapped by an appropriate operation that we call “attribute”, and where ingredients are in the “soup” via an AC concatenation operation. Some of the soup ingredients are lists (e.g., I/O “buffers”, function stacks, continuations), others are sets (e.g., environments, stores), while others are just plain numbers (e.g., the next location). Like the mix-fix algebraic signature associated to the BNF in Figure 1, we do not define the state signature here either, because it is straightforward.

When a program is executed, we need to construct its initial state. We do this using an eval operation. For SILF, this operation would take a program, \( Pgm \), and an input list of integers, \( Nl \), and “insert” them into a starting state:

\[
\text{eval}(Pgm, Nl) = k(Pgm) fstack(k)(env(k)(gen(k)(fenv(k)(input(k)(Nl)) output(k)) store(k)) nextLoc(0))
\]

The continuation structure wrapped by \( k \) keeps an ordered list of tasks to be performed to continue the computation. We add additional sorts to represent the abstract syntax, including values \( (V) \), environments \( (Env) \), continuations \( (K) \), locations \( (L) \), and stores \( (Mem) \), with appropriate lists and sets for each.

**Programs.** A program is made up of a number of global variable declarations, followed by a number of functions. There is no inherent order to the functions.
– all functions can see all other functions. To execute a program, we need to process all global variable declarations, create the global environment, process all function declarations, and then invoke the main function:

$$k\left( \text{pgm}(S \ FDs) \right)$$

$$\text{stmt}(S) \leadsto \text{mkGenv} \leadsto \text{fdecl}(FDs) \leadsto \text{stmt}(\text{main})$$

How \text{stmt}(S) is processed is described later in this section. One can view \text{stmt}(S) and \text{exp}(E) as “compiling” the statement \(S\) or expression \(E\), turning it into a continuation. As seen shortly, when \(S\) contains only variable declarations, \text{stmt}(S) at the top of the continuation eventually produces a corresponding environment in the attribute \text{env}. Then, \text{mkGenv} only needs to move that environment into \text{genv}(\text{this will allow us to easily refer to the global variable environment later}):

$$k(\text{mkGenv}) \text{env}(\text{Env}) \text{genv}(\_ )$$

Function declarations are processed one by one:

$$fdecl(\text{FD:FunDecl} \ FDs: \text{FunDeclNeSet})$$

$$\text{fdecl(\text{FD})} \leadsto fdecl(\text{FDs})$$

**Functions.** Function semantics cover three main constructs: function declaration, function invocation, and function return. We cover each below in turn. We first need to add the declared functions into the function environment. We do assume that function names are distinct and that declarations all occur at the start of the function. We add the necessary structure to the function body to bind the input values to the formal parameters, so we do not need to add this in the invocation semantics (the semantics of \text{bind} will be given shortly):

$$k(\text{fdecl(function } I(\text{Is}) \text{ begin } S \text{ end}) \equiv \text{fenv}(\_ , \_ ) \equiv \text{fenv}(\text{bind}(\text{Is} \leadsto \text{stmt}(S)))$$

Functions can be used as either expressions or statements:

$$\text{exp}(I(\text{El})) \equiv \text{apply}(I)$$

$$\text{stmt}(I(\text{El})) \equiv \text{apply}(I) \equiv \text{discard}$$

The continuation item \text{exp}(\text{El}), when at the top of the continuation, evaluates the list of expressions \(\text{El}\) sequentially and produces their corresponding values, a term of the form \text{val}(\text{Vl}). When used as a statement, we put a discard continuation item into the continuation to throw away the return value (this will be defined shortly). Once the arguments have been evaluated, we can apply the function. Since functions are stored just as identifier/continuation pairs, we can just grab out the continuation for the function. Also, we save the current continuation and environment so we can quickly recover these when we exit the function on a return:

$$k(\text{val}(\_ ) \leadsto \text{apply}(I) \leadsto K) \equiv \text{fstack}(\_ \_ ) \equiv \text{env}(\_ \_ ) \equiv \text{fenv}(\text{bind}(\_ \_ ) \equiv \text{genv}(GEnv)$$
When we encounter a return, first we need to evaluate the expression whose value we are returning. Once the value has been calculated, we can then switch context back to the caller, which we do by replacing the current environment and continuation with those saved at the top of the function stack:

\[
\frac{stmt(\text{return } E \ )}{exp(E) \rightsquigarrow \text{return} \quad \frac{k(\text{val}(\_)) \rightsquigarrow \text{return } \_ \ ) \ fstack(\ Env, K) \ } \ env(\_ \ )}{Env}
\]

**State Helper Operations.** Many of the rules in the SILF semantics perform similar changes to the state. We have abstracted these changes into a number of rules which can then be used across different parts of the semantics. The operation \( \text{bind} \) creates new bindings in the environment. This operation binds a list of values to a list of identifiers, adding the identifier to the environment and the value to the store, linked by a shared location. To create a new binding in the environment without a value, we use a variant of the \( \text{bind} \) operation, which binds a list of identifiers to a list of locations but does not alter the store (\( \text{len} \) is the usual length operation on lists and \( Ll \) is the location list \( [L, L + 1, \ldots, L + \text{len}(Il) - 1] \)):

\[
\frac{k(\text{val}(Vl) \rightsquigarrow \text{bind}(Il)) \ } {env(\ Env \ ) \ store(\ Mem \ ) \ nextLoc(\ L \ ) \cdot \text{Env}[Il \leftarrow Ll] \ Mem[Il \leftarrow Vl]} \ L + \text{len}(Il)
\]

\[
\frac{k(\text{bind}(Il)) \ } {env(\ Env \ ) \ nextLoc(\ L \ ) \cdot \text{Env}[Il \leftarrow \text{locs}(L, \text{len}(Il))]} \ L + \text{len}(Il)
\]

The \( [\_ \leftarrow \_] \) operation will properly update the set, using the list on the left as a list of “keys” to either add a new key/value pair to the set or replace an existing key/value pair with a new pair. The definition is straightforward, and is not shown here.

We can also bind blocks of storage. This will just bind the first location to the identifier and then advance the next location an arbitrary amount. This can be used to represent allocating a block of memory for an array.

\[
\frac{k(\text{val}(\text{int}(N)) \rightleftharpoons \text{bindBlock}(I)) \ } {env(\ Env \ ) \ nextLoc(\ L \ ) \cdot \text{Env}[I \leftarrow L]} \ L + N
\]

For assignment, \( \text{assignTo} \) assigns a value to the store in two steps, first converting identifier assignment (\( \text{assignTo} \)) to location assignment (\( \text{assignToLoc} \)) then carrying out the assignment:

\[
\frac{k(\text{val}(V) \rightleftharpoons \text{assignTo}(I)) \ } {\text{env}(\langle I, L \rangle \ ) \ assignToLoc(L) \cdot \text{Mem}[L \leftarrow V]}
\]
We also have a similar version for arrays, which will assign at an offset.
\[
\begin{align*}
& k(\text{val}(\text{int}(N), V) \rightarrow \text{arrayAssign}(I)) \ni \text{env}(I, L)) \\
& \text{val}(V) \ni \text{assignToLoc}(L + N)
\end{align*}
\]

Similarly we have two lookup operations:
\[
\begin{align*}
& k(\text{lookupLoc}(L) \rightarrow \text{store}((L, V))) \ni \text{env}(I, L)) \\
& \text{val}(V) \ni \text{assignToLoc}(L + N)
\end{align*}
\]

Occasionally we will want to discard a value from the continuation. To do so, we use \textit{discard} with the following semantics: \(k(\text{val}(V) \ni \text{discard})\).

\textbf{Variable Declarations.} In SILF we have two different types of variable declarations – integers and integer arrays. Arrays can only be declared of a fixed (positive integer) size. In both cases, the declaration does not set an initial value – this corresponds to a concept of “junk” in the memory before assignment, and any read attempts of “junk” will fail. We treat arrays identically to C (arrays are 0 indexed, so an array of 10 elements is indexed from 0 to 9) with the location of the array name the same as location 0:

\[
\begin{align*}
& \text{stmt}(\text{var } I) \\
& \text{bind}(I)
\end{align*}
\]

\[
\begin{align*}
& \text{stmt}(\text{var } I[N]) \\
& \text{val}(\text{int}(N)) \ni \text{bindBlock}(I)
\end{align*}
\]

\textbf{Lookups and Simple Expressions.} Some of SILF’s most basic expressions are lookups of name and indexed array values, as well as literal expressions. For a literal integer, we just return a value with the integer encapsulated in a value wrapper: \(\text{exp}(N)\). For both identifiers and arrays, we return the current value, either assigned to the identifier or to the given element of the array. We will process this in two steps, first retrieving the value’s location, then retrieving the value:

\[
\begin{align*}
& k(\text{exp}(I) \rightarrow \text{lookupLoc}(L)) \ni \text{env}(I, L)) \\
& \text{exp}(E, E') \ni + \ni \text{val}(\text{int}(N), \text{int}(N')) \ni + \\
& \text{val}(\text{int}(N + \text{int} N')) \ni + \\
& \text{exp}(E') \ni < \\
& \text{exp}(E, E') \ni < \\
& \text{val}(\text{int}(N), \text{int}(N')) \ni < \\
& \text{val}(\text{bool}(N < \text{int} N'))
\end{align*}
\]

\textbf{Arithmetic, Relational, and Logical Operations.} All three operation types follow the same general pattern. When we encounter an addition expression, e.g., we first need to evaluate both operands. We also need to keep track of what operation we are performing. So, we will replace an expression such as \(E + E'\) with one were we evaluate \(E\) and \(E'\) and put + on the continuation to remind ourselves what we need to do with the results. Once we get back the values from evaluating the two expressions (here, expected to both be integers) on top of a +, we return their sum (using integer addition):

\[
\begin{align*}
& \text{exp}(E + E') \\
& \text{val}(\text{int}(N), \text{int}(N')) \ni + \\
& \text{val}(\text{int}(N + \text{int} N')) \ni + \\
& \text{exp}(E') \ni < \\
& \text{exp}(E, E') \ni < \\
& \text{val}(\text{int}(N), \text{int}(N')) \ni < \\
& \text{val}(\text{bool}(N < \text{int} N'))
\end{align*}
\]

Relational operators work identically to arithmetic operators, except we apply relational operations on the results and return boolean values:

\[
\begin{align*}
& \text{exp}(E < E') \\
& \text{val}(\text{int}(N), \text{int}(N')) \ni < \\
& \text{val}(\text{bool}(N < \text{int} N'))
\end{align*}
\]
Logical operations are handled almost exactly the same:

\[
\begin{align*}
\exp(E \and E') & \rightsquigarrow E \and E' \\
\text{val}(\text{bool}(B), \text{bool}(B')) & \rightsquigarrow \text{and} \\
\text{val}(\text{bool}(B \and \text{bool}(B'))) & \rightsquigarrow \text{and}
\end{align*}
\]

All the arithmetic, relational, and logical operations are defined in Appendix A.

**Assignment Statements.** SILF has two types of assignment:

\[
\begin{align*}
\exp(E) & \rightsquigarrow \text{assignTo}(I) \\
\exp(E[E']) & \rightsquigarrow \text{arrayAssign}(I)
\end{align*}
\]

**Conditional Statements.** SILF has two conditionals, one with just a true branch, one with true and false branches. We convert the first into the second:

\[
\begin{align*}
\text{if } E \text{ then } St \text{ fi} \\
\text{if } E \text{ then } St \text{ else skip fi}
\end{align*}
\]

For the general conditional, we first evaluate the condition, “compiling” the two branches and storing them in the continuation, wrapped by \( \text{if}(\_,\_) \):

\[
\begin{align*}
\text{stmt}(\text{if } E \text{ then } St \text{ else } Sf \text{ fi}) \\
\exp(E) & \rightsquigarrow \text{if}(\text{stmt}(St), \text{stmt}(Sf))
\end{align*}
\]

If the result is true, then we will evaluate the first branch (which we have already converted into a continuation), and if false we will evaluate the second:

\[
\begin{align*}
\text{val(\text{bool(true})}) & \rightsquigarrow \text{if}(Kt, Kf) \\
\text{val(\text{bool(false}}) & \rightsquigarrow \text{if}(Kt, Kf)
\end{align*}
\]

**Loop Statements.** We transform “for” loops into “while” loops:

\[
\text{for } I := E_1 \text{ to } E_2 \text{ do } S \text{ od}
\]

\[
I := E_1; \text{while } I \leq E_2 \text{ do } S ; \text{ } I := I + 1 \text{ od}
\]

We give semantics to “while” loops by changing the while statement into a while continuation that contains the (“compiled”) guard expression and the while body, at the same time evaluating the guard:

\[
\begin{align*}
\text{stmt(while } E \text{ do } S \text{ od}) \\
\exp(E) & \rightsquigarrow \text{while}(\exp(E), \text{stmt}(S))
\end{align*}
\]

Next, based on whether the guard evaluates to true or false, we do or do not need to evaluate the body of the while:

\[
\begin{align*}
\text{val(\text{bool(true})}) & \rightsquigarrow \text{while}(Ke, Ks) \\
Ks & \rightsquigarrow \text{while}(Ke, Ks)
\end{align*}
\]

**I/O Statements.** SILF allows for rudimentary I/O, with the ability to read and write integers. For input, we take the next available integer:

\[
\begin{align*}
\text{k(\_ exp(read) \_) input(} N \text{)} \rightsquigarrow \text{val(int}(N))
\end{align*}
\]
For output, we evaluate the expression, then add it to the end of the output:

\[
\begin{align*}
\text{stmt}(\text{write } E) & \quad \quad \quad k(\text{val}(\text{int}(N)) \rhd \text{write}) \quad \text{output(·)} \\
\text{exp}(E) \rhd \text{write} & \quad \quad \quad N
\end{align*}
\]

**Sequential Composition** is straightforward:

\[
\text{stmt}(S; S') \quad \quad \quad \text{stmt}(S) \rhd \text{stmt}(S')
\]

4 Towards Automatic Synthesis of Language Interpreters

An important goal which we set for the K framework is that it should allow us to automatically generate efficient interpreters from language definitions. While this goal is still ahead of us, here we briefly present the semi-automatic generation of an interpreter for SILF.

**Preprocessing.** We currently assume as input a well-formed, type-checked program, which is then preprocessed to yield a simpler yet semantically equivalent program. During preprocessing, identifiers are replaced by wrapped numbers (wrapped with \(l\) for local and \(g\) for global identifiers) and variable declarations by memory allocation commands. Integers are wrapped (e.g., \(i(0)\) for 0), and functions are named with indices and parameter list sizes to aid with allocation (e.g., \(f(3)(5)\) for function number 3 with 5 parameters). This essentially eliminates the environment, which is now just an index into the store, similar to a frame pointer. We can best illustrate this with an example. In Figure 3, we have two programs. The program on the left is a program in SILF, while the program on the right is the equivalent program after translation. Note that translation can be performed statically and automatically.

**Precompilation and instruction generation.** We chose to clearly divide the semantic rules into precompilation and execution rules. The precompilation

```plaintext
function writeBinary(x) begin
  var i;
  var b[32];
  var j;
  i := 0;
  while x > 0 do
    b[i] := x % 2;
    x := x / 2;
    i := i + 1
  od
  j := i - 1;
  while j >= 0 do
    write b[j];
    j := j - 1
  od
end

function main(void) begin
  writeBinary(read)
end
```

```plaintext
function f(1)(1) {
  alloc(1);
  alloc(32);
  alloc(1);
  l(i(1)) := i(0);
  while l(i(0)) > i(0) do {
    1(i(1)) := 1(i(0)) % i(2);
    1(i(0)) := 1(i(0)) / i(2);
    1(i(1)) := 1(i(1)) + i(1)
  }
  l(i(34)) := l(i(1)) - i(1);
  while l(i(34)) >= i(0) do {
    writeInt(1(l(i(34)) + i(1)));
    l(i(34)) := l(i(34)) - i(1)
  }
}

function f(0)(0) {
  f(1)(readInt)
}
```

Fig. 3. Source and Translated Programs
Fig. 4. Evaluation Results

phase reduces the program to a continuation, which the execution phase then runs to modify the state. In our case, we can divide the semantic rules into two groups: those in which the left-hand-side is a state and those in which it is a continuation. We precompile only the latter, dividing each language task (e.g., assignment, function call) into a series of smaller tasks. Bytecode is then generated from a precompiled form of the program by a process of flattening, translating the graph-like structure of the continuation into an array. The bytecode “instructions” are given by the continuation items. This process is mostly automatic, with our instructions determining the structure of the virtual machine.

**Execution.** The execution rules act on a modified version of the state, with a separate stack for values and a control stack for continuations. This requires a change in some of the rules, which we believe can be automated. This then aligns with the interpreted view of the rules, with stores and stacks represented as arrays, and stack operations represented as array index manipulation. The interpreter executes program by referencing the item on top of the continuation and the values on top of the stacks, which uniquely determine the rule to apply (with the continuation item alone determining most of the rules). The virtual machine then executes an infinite loop which selects the next continuation item and runs the code for the selected rule.

**Evaluation.** For evaluation we have chosen several programs, each exercising different execution tasks. perm is an all-permutations generation algorithm using recursive backtracking with globals and returns. binary computes the base two representation for all numbers up to the input number by successive divisions by 2, and exercises iterative function calls with local array declarations. sieve is the Eratosthenes’ sieve algorithm for computing primes up to the input number, which exercises addressing large arrays. Finally, hanoi is the standard recursive solution for the Hanoi towers problem, exercising recursive functions. Results are shown in Figure 4. We don’t have results for BC on sieve, since BC only allows 16 bit array indexes. The C interpreter for SILF outperforms BC and is competitive with C, and occasionally outperforms Java (additional work is needed to determine under what circum-

<table>
<thead>
<tr>
<th>Program</th>
<th>K to Maude</th>
<th>K to C</th>
<th>BC</th>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>perm(6)</td>
<td>80.840</td>
<td>0.048</td>
<td>0.155</td>
<td>0.003</td>
<td>0.174</td>
</tr>
<tr>
<td>perm(9)</td>
<td>*</td>
<td>45.560</td>
<td>154.016</td>
<td>1.615</td>
<td>11.342</td>
</tr>
<tr>
<td>binary(1,000)</td>
<td>17.037</td>
<td>0.019</td>
<td>0.100</td>
<td>0.004</td>
<td>0.190</td>
</tr>
<tr>
<td>binary(1,000,000)</td>
<td>*</td>
<td>32.631</td>
<td>209.949</td>
<td>4.955</td>
<td>55.782</td>
</tr>
<tr>
<td>sieve(10,000,000)</td>
<td>*</td>
<td>27.671</td>
<td>-</td>
<td>1.199</td>
<td>3.591</td>
</tr>
<tr>
<td>hanoi(23)</td>
<td>*</td>
<td>18.140</td>
<td>86.432</td>
<td>4.394</td>
<td>57.761</td>
</tr>
</tbody>
</table>

Execution times in seconds. − indicates test not performed, * indicates test timed out. Evaluation performed on Intel® Pentium® 4 CPU 2.00GHz with 1GB RAM, gcc version 3.3.6, compilation flags: -O3 -march=pentium4 -pipe -fomit-frame-pointer
stances). *Maude*’s times are higher because of extensive *ACL*-matching, reducing speeds from millions of rewrites to around tens of thousands of rewrites per second. Because of this, we do not have figures for *Maude* for the larger test cases.

5 Related Work

There are a number of different methods for specifying the semantics of programming languages, including operational methods such as Plotkin’s SOS [8], denotational methods such as those from Scott and Strachey [10], Mosses’s action semantics [6] and MSOS [7], and Meseguer and Roșu’s rewriting logic semantics [4], among many others. *K* allows for complex control flow, such as loop break and continue, exceptions, and call/cc, which are difficult to specify using operational methods such as SOS or MSOS, but does not yet have the same “toolset” developed for language-related proofs, such as is common with SOS definitions using inductive techniques (for subject reduction, for instance). Denotational methods and *K* seem to provide similar power for defining language features (at least in a setting without concurrency), but arguably the mathematics involved in rewriting logic is simpler than that in denotational methods, especially those making use of category theory such as Moggi[5].

There is also significant work on executable definitions of language semantics, including the aforementioned rewriting logic semantics. Another interesting example is Centaur [2], which includes a Prolog engine for executing formal language specifications. We believe the high-performance nature of rewriting engines provides a more realistic platform for running interpreters, although we have not yet done specific performance comparisons. Another executable semantic framework is ASF+SDF [11], which also uses term rewriting to define programming languages, but our contextual, continuation-based methodology, involving explicit access to the control state, appears quite different.

One appealing aspect of rewriting logic semantics is that precisely the same rewrite logic definition of a language gives both an algebraic denotational semantics (an initial model semantics) and an operational semantics (the initial model is executable). Of the above, our work is most similar to rewrite logic semantics; more precisely, our framework can be regarded as a domain-specific syntactically sugared rewriting logic semantical framework (i.e. de-sugaring would give us a standard rewrite logic representation of the language semantics).

**K and Rewriting Logic.** One question that naturally arises is how language definitions using *K* are different from those given directly in rewriting logic. We believe that *K* provides several distinct advantages.

- In our experience using rewriting logic to define languages, we have noticed that long rules, especially those with complex, nested control structures, can be very difficult to read. This creates a barrier to those that would like
to use rewriting logic to define languages but find it to be too complex. The compactness of the $K$ rules, in our opinion, improves greatly on readability;

- We have also noticed that, with long rules, it is easier to make mistakes, either when the rule is initially written or when it is later modified. Again, the shortness of the $K$ rules, especially the ability to both leave out inferrible context and list unchanged parts of the term only once, help alleviate this problem;

- As mentioned, variable definitions often constitute a significant percentage of a module. The ability to infer sorts of variables keeps definitions shorter, while still allowing explicit sort annotations for documentation purposes;

- The ability to elide parts of the context which are not necessary for a rule allows the context, especially those parts not mentioned in the rule, to change. This increases the modularity of the rules, since adding new features then rarely requires changes to the existing rules.

6 Conclusions and Future Work

In this paper we introduced the $K$ language definition framework and used it to define a simple imperative language with functions. We also showed an example of translating this definition into an interpreter in C. Based on current encouraging results, we believe this is a promising strategy for automatically deriving interpreters from definitions of language semantics.

There is much future work yet to do. We are still looking for ways to improve $K$ as we gain more experience using it to define languages. We are also continuing work on automatically generating interpreters in rewriting logic and C from $K$ definitions, which is currently a mix of manual and automated processes. We believe there is no reason this cannot be done in a fully automatic fashion. Along with this, we are looking for ways to more closely define both the syntax and semantics of languages, to allow for the automatic generation of language parsers and other static tools which process program text using rules we have defined in $K$.

Finally, we would like to thank the valuable feedback from the anonymous reviewers, which has allowed us to improve the quality of this paper.

References


Here we include the additional rules in the semantics for SILF which were not included above. These rules are similar to those shown in Section 3.

### A.1 Arithmetic Operations

\begin{align*}
\frac{\text{exp}(E + E')}{\text{exp}(E,E') \rightarrow +}
\frac{\text{val}(\text{int}(N), \text{int}(N')) \rightarrow +}{\text{val}(\text{int}(N + \text{int } N'))}
\frac{\text{exp}(E - E')}{\text{exp}(E,E') \rightarrow -}
\frac{\text{val}(\text{int}(N), \text{int}(N')) \rightarrow -}{\text{val}(\text{int}(N - \text{int } N'))}
\frac{\text{exp}(E * E')}{\text{exp}(E,E') \rightarrow *}
\frac{\text{val}(\text{int}(N), \text{int}(N')) \rightarrow *}{\text{val}(\text{int}(N * \text{int } N'))}
\frac{\text{exp}(E / E')}{\text{exp}(E,E') \rightarrow /}
\frac{\text{val}(\text{int}(N), \text{int}(N')) \rightarrow /}{\text{val}(\text{int}(N / \text{int } N'))}
\frac{\text{exp}(E \% E')}{\text{exp}(E,E') \rightarrow \%}
\end{align*}
\[
\frac{\text{val}(\text{int}(N), \text{int}(N')) \rhd \%}{\text{val}((\text{int}(N) \times \text{int}(N')))}
\]

\[
\frac{\text{val}(\text{int}(N)) \rhd u-}{\text{val}(\text{int}(-\text{int}(N)))}
\]

### A.2 Relational Operations

\[
\frac{\exp(E < E')}{\exp(E, E') \rhd <}
\]

\[
\frac{\text{val}(\text{int}(N), \text{int}(N')) \rhd <}{\text{val}(\text{bool}(N <_{\text{int}} N'))}
\]

\[
\frac{\exp(E <= E')}{\exp(E, E') \rhd <=}
\]

\[
\frac{\text{val}(\text{int}(N), \text{int}(N')) \rhd <=}{\text{val}(\text{bool}(N <=_{\text{int}} N'))}
\]

\[
\frac{\exp(E > E')}{\exp(E, E') \rhd >}
\]

\[
\frac{\text{val}(\text{int}(N), \text{int}(N')) \rhd >}{\text{val}(\text{bool}(N >_{\text{int}} N'))}
\]

\[
\frac{\text{exp}(E) - \exp(E')}{\exp(E, E') \rhd -}
\]

\[
\frac{\text{val}(\text{int}(N)) \rhd u-}{\text{val}(\text{int}(-\text{int}(N)))}
\]

### A.3 Logical Operations

Note that these operations are not short-circuit, since we evaluate both operands to and or at once. We could make them short-circuit by instead evaluating only the first operand, and storing the second with the continuation for the operator. Based on the result of evaluating the first operand, we could then either return the proper value or evaluate the second operand to give us the value of the operation.

\[
\frac{\text{exp}(E \text{ and } E')}{\exp(E, E') \rhd \text{ and}}
\]

\[
\frac{\text{val bool}(B), \text{bool}(B') \rhd \text{ and}}{\text{val bool}(B \text{ and bool } B')}
\]
\[
\begin{align*}
\frac{\text{\textit{exp}}(E \textbf{ or } E')}{\text{\textit{exp}}(E, E') \sim \textbf{or}} \\
\frac{\text{\textit{val}}(\text{\textit{bool}}(B), \text{\textit{bool}}(B')) \sim \textbf{or}}{\text{\textit{val}}(\text{\textit{bool}}(B \textbf{ or } B'), \text{\textit{bool}}(B'))} \\
\frac{\text{\textit{exp}}(\textbf{not} E)}{\text{\textit{exp}}(E) \sim \textbf{not}} \\
\frac{\text{\textit{val}}(\text{\textit{bool}}(\textbf{not} B)) \sim \textbf{not}}{\text{\textit{val}}(\text{\textit{bool}}(B))}
\end{align*}
\]
A Rewriting Semantics for ABEL with Applications to Hardware/Software Co-Design and Analysis

Michael Katelman and José Meseguer

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801, U.S.A.

Abstract

We present a rewriting logic semantics in Maude of the ABEL hardware description language. Based on this semantics, and on Maude’s underlying LTL model checker, we propose a scalable formal analysis framework and tool for hardware/software co-design. The analysis method is based on trace checking of finite system behaviors against LTL temporal logic formulas. The formal properties of the hardware, the embedded software, and the interactions between both can all be analyzed this way. We present two case studies illustrating our method and tool.

Key words: co-verification, co-design, rewriting logic, program semantics, hardware description languages, Maude

1 Introduction

The restricted class of functionality required of an embedded system suggests the possibility that its hardware and software components might be designed concurrently, commonly called hardware/software co-design [18]. The typical target for a co-design is much more limited than a general purpose microprocessor: it is usually an embedded system such as those found in consumer electronics or in control systems of various kinds, for example, those found in automobiles or power plants. One important recent trend with many applications is towards distributed embedded systems having their own communication resources and being connected through wireless networks. In sheer size, the number of such systems now dwarfs by several orders of magnitude the number of standard computers.

1 Email: {katelman, meseguer}@uiuc.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Co-design engineering is a complex process that goes well beyond simply delegating tasks to the appropriate engineer. A critical step in the co-design process, though not necessarily the first, is to partition the system into major functional blocks, and then to match these functional blocks with specific design technologies. In particular, a decision is made as to which components will be implemented as hardware and which will be implemented as software. The choices made during this step impact directly the cost and complexity of implementing the system, as well as the implementation’s performance. A system that is required to compute Fourier transforms, for example, could choose to do so in software on a general purpose microprocessor, in software on a digital signal processor (DSP), or directly in hardware as part of an application-specific integrated circuit (ASIC).

Contemporary research into embedded system design \cite{22,13,7} has advocated a significant change to the process outlined above, whereby the major functionalities are first modeled at a high level before being mapped to specific technologies, such as a DSP or ASIC. Modeling is realized through a collection of formalisms covering a variety communication, concurrency, and computation schemes. In the Ptolemy II system design environment \cite{13}, for example, an engineer can describe computations using combinations of the many supported models, including data-flow network and discrete-event models. By separating functionality from implementation, tools such as Ptolemy II attempt to coordinate the system design process and produce verified designs more quickly by leveraging modularity and abstraction. In limited cases, automated tools have been able to produce implementations directly from these high-level models.

1.1 Embedded System Verification

Our view of co-verification, that is, the process and mechanisms through which a co-design is verified, is summarized by the generalized diagram presented as Fig. 1. Depending on the particular verification regime applied and the level at which the system’s components are described, the individual boxes can be resolved more concretely. In this paper we focus on co-verification of implementation level designs, and in particular we focus on the hardware/software interactions at this level. We see this work as being part of a longer term and more ambitious project to formally specify and analyze embedded systems, especially the real-time and hybrid components of a system’s operating environment.

Regarding Fig. 1, we need a natural hardware model to capture implementation level designs. Register-transfer level (RTL) code is the most pervasive and natural choice, encompassing (subsets of) well known hardware description languages (HDL) such as Verilog and VHDL. For our work in this paper, however, we chose another language, ABEL \cite{24}, which is owned by Xilinx Inc. and is primarily used to program FPGAs, a widely utilized technology in
embedded systems. For a piece of software to be able to ‘run’ on a hardware component designed in ABEL, it must be at a level that the hardware can understand, and so we have focused on assembly language programming for our software model. The exact format and set of instructions allowed as part of the assembly code must be tailored to the particular microprocessor that it is targeting. Therefore, the verification environment must be updated to support a new instruction set when analyzing a new microprocessor component.

The final component in Fig. 1 is reserved for specifying what should be verified, which in our case will be properties of RTL microprocessors and of assembly language programs written to run on those processors. There are many interesting properties about these systems that can only be described with properties that span hardware, software, and meta-level properties of the software and its data structures. The goal of this paper is to demonstrate the natural way in which rewriting logic can be used to unify the varying levels at which the properties need to be described, so that they can be reasoned about together. For this purpose we present a Maude-based co-verification environment.

The particular verification methodology that our environment supports is a very practical kind of formal analysis, namely trace checking, a formal method widely used for verifying both hardware [4,1,6] and software [11,19]. In trace checking, formal properties expressed as temporal logic formulas are checked for finite traces, or runs, of the system. This has the practical advantage of saving the engineering effort that would be required to build a self checking test bench to do the same thing, and helps the validation engineer focus on the properties, rather than on how they are checked.

The rest of this paper is organized as follows. Section 2 presents, by way of ABEL, a high-level overview of how the syntax and semantics of a hardware description language are specified in Maude. Section 3 explains in detail our framework for specifying and trace checking co-designs and illustrates our proposed method through two in-depth case studies. Section 4 discusses related
work, and Section 5 presents some conclusions.

2 Syntax and Semantics for ABEL

This section presents a high-level overview of our ABEL semantics focusing on major design decisions rather than on the particular details of ABEL. In places where concreteness can get a point across more quickly, we use ABEL syntax and explain the necessary details.

Rewriting semantics for many different types of programming languages has been described previously (e.g. see [15]). The goal of this section, in part, is to explain those areas of the RTL semantics that differ from software languages. Our second goal, which, in fact, falls out from the first, is to give enough semantic detail to support our discussion of trace checking in §3. Lastly, by way of ABEL specifically, we also show that the framework presented in §3 would permit a hardware designer to build a digital circuit in a way that is consistent with current engineering practice.

2.1 Circuit Semantics

Synchronous digital circuits need to respect two separate notions of execution order. First, during each clock cycle, the topology of the combinational (non-state carrying) network must be respected: each internal node value within the circuit is considered accurate only after its inputs become accurate, and some amount of time has elapsed to compute the value from the stabilized inputs. Second, latches should respect a *global clock* and should all be updated in parallel.

Although a synthesized circuit will mix combinational and sequential elements, it is convenient for the purposes of simulation to separate the two. Many HDL simulators do this, whereby the circuit is treated as a combinational network only. This is accomplished by changing the circuit’s input/output interface to include new bits for each state element. The new inputs and outputs represent the current-state and next-state values, respectively. This transformation is used as the basis for our ABEL semantics.

Each clock cycle, every identifier (specified in the ABEL modules, that is, in the ABEL syntax) gets associated with at least one value in the transformed circuit. These associations take the form of a triple of sort $\text{NodeValue}$, and individual triples are gathered together into a set.

\[ \text{op } [\_,\_,\_] : \text{NodeType Identifier Value } \rightarrow \text{NodeValue} . \]

The first argument is a token describing what *type* of node it is in the transformed circuit: primary input (pi), current-state input (cs), primary output (po), next-state output (ns), or internal node (in). Fig. 2 shows an ABEL module (suitably modified for Maude parsing) that stores a value from the current clock cycle, allowing it to be used during the next clock cycle. Optionally, this output can be suppressed. The single bit wires, or *pins*, represented by
module "DFFE"
declarations
    'D, 'EN input pin ;
    'Q node istype reg ;
    'OUT output pin ;
equations
    'Q := 'D ;
    when 'EN then
    'OUT = 'Q ;
    else
    'OUT = .Z. ;
end-module

Fig. 2. Example Module and Circuit Diagram

the identifiers 'D and 'EN are inputs to the module; 'Q is an internal node; and 'OUT is the only output. The fact that 'Q takes its assigned value during the next clock cycle is specified with the istype reg qualifier. In the equations section for the circuit, 'Q is assigned, for the following clock cycle, the current value of 'D. 'OUT gets the same value as 'Q if the enable pin 'EN is asserted (i.e. has logical value 1), and the special value .Z. otherwise. The current state of the circuit could look as follows:

[pi, 'D, 0] [pi, 'EN, 1] [cs, 'Q, 1] [ns, 'Q, 0] [po, 'OUT, 1]

Indicating that during the previous clock cycle 'D was 1, and the 'EN pin is asserted.

Our semantics has a single rewrite rule that transitions the current state (a set of triples of sort NodeValue) to the next clock tick by evaluating the ABEL constructs in the current state and with user-specified inputs. The equational rewrite rules necessary to facilitate the transition from one clock cycle to the next are very much like the ones used to specify the semantics for a simple imperative programming language in [15]. Equations are evaluated based on the order in which they are written down in the module, and, moreover, ABEL only supports what are essentially the equivalent of assignment, if-then-else, and switch statements, which are very easy to specify. In some cases the semantics proved tedious due to idiosyncrasies of ABEL, but were not technically difficult to handle. The major difference, then, from the semantics for a simple imperative programming language is that we explicitly evaluate the “program” over and over again, instead of only once.

An alternative semantics for digital circuits is described in [10], where instead of a synchronizing rewrite rule, each node value is tagged with the cycle number for which the node takes on the calculated value. In that scheme, it is not necessary to have any rewrite rules, but for use with the Maude model checker, it was cleaner and more useful to specify synchronization as explained above.
3 Trace Checking the Hardware/Software Interface

This section describes how we have designed an effective co-verification environment in Maude for use on implementation level system designs. A large component of our framework consists of the ABEL semantics presented in §2, so in what follows we mainly focus on how our framework exploits the ABEL semantics for trace checking. We describe both how the semantics allows us to use Maude’s LTL model checker for trace checking, and the related subject of how to build an effective propositional language for writing meaningful LTL formulas. The language covers hardware, software, and meta-level properties of the software and its data structures, so that all of these entities can be reasoned about together and their interactions can be explored. §3.2 presents a case study.

3.1 Co-Verification Environment

Hardware/software co-design is fundamentally about balance: deciding how the functionality of a system should be partitioned into hardware and software components. Understanding the purpose and goals of the project can help with defining metrics that guide the engineering team toward a suitable partition, one that strikes a balance among the various metrics. Common axes that are considered and influence design decisions include performance, power, size, programmability, and reduced complexity.

Clearly, the verification of a co-design project should include the interactions between hardware and software; both to ensure system wide correctness and to validate the engineering choices that led to the partitioning scheme. If, for example, the instruction throughput is never high enough to take advantage of all of the functional units, then it might be worthwhile to use this information to reevaluate the hardware partition and scale back the design. If the software is complicated and depends on many subtle interactions with the available hardware, then this inefficiency might be very hard to uncover without an implementation to test on, or without formal analysis. If, as a second example, the co-design effort tried to minimize power consumption, then it would be useful to formally validate how the power saving features perform when the software is executing on the chip. Hence, validation of the co-design decisions after implementation may be necessary to guide redesign or other efforts in the future.

Creating a Maude specification for the semantics of an HDL such as ABEL represents our first step in an effort to facilitate co-verification. The strong modeling properties of rewriting logic, and Maude in particular, allow a user to take both the hardware and software components of a design and instantly embed them into a unified mathematical framework. In addition, meta-level properties of the system (e.g. software data structures) can also be conveniently modeled and related to the rest of the design. This allows the engineer to specify properties of his/her system in the most appropriate and natural
language, hence promoting a modular verification effort that abstracts away unnecessary details and helps avoid error-prone encodings.

Our current verification framework is built around a combination of simulation and LTL model checking, called trace checking. For trace checking to be applicable, the state of the co-designed system must evolve in exactly one way, corresponding to a concrete simulation run on fixed inputs. Our ABEL semantics supports this by design: there is only one rewrite sequence, modulo ACI axioms. When using a propositional model checker such as the one in Maude, it is necessary to develop a strong propositional language; one that makes it easy to write properties of the circuit at a high level of abstraction. For example, a circuit design used to control a traffic light should allow the current state of the light to be specified as red, yellow, or green, as opposed to the signal values that each color corresponds to in the circuit. The same remains true when the model checker is being used for the more restricted case of trace checking, and so the propositional language that we have developed for trace checking co-designs has many of these features. On the hardware side, node values associated with an identifier can be checked using the propositions

\[
\begin{align*}
\text{op } \_\text{asserted} &: \text{Identifier } \rightarrow \text{Proposition} \\
\text{op } \_\text{deasserted} &: \text{Identifier } \rightarrow \text{Proposition} \\
\text{op } \_\text{::=} &: \text{Identifier Int } \rightarrow \text{Proposition} \\
\text{op } \_\text{::=} &: \text{Identifier Identifier } \rightarrow \text{Proposition} \\
\text{op } \_p &: \text{Proposition } \rightarrow \text{Prop} 
\end{align*}
\]
For identifiers associated with multiple node values in the transformed circuit, the value used is the one associated with the current-state. Single bit nodes are asserted if they have the value 1 in the current state, and de-asserted if they have the value 0. Multi-bit values can be checked for equality with an integer constant or with the value of another multi-bit node, again by checking the current state. The ::= symbol is used for this purpose, to avoid conflicts with other sorts defined in Maude. In addition, it is convenient and easy to name constant values, such as the colors in the traffic light example. If the module declares each color as a constant value, then the identifiers associated with each constant value can be used directly by writing

\texttt{\'LIGHT ::= \'}GREEN

As usual for LTL model checking in Maude, the semantics of state propositions is defined equationally, with equations that query the current state values to decide the truth or falsity of the proposition. For assembly programs we have added other propositions: for example we have propositions that check for when an instruction is executed, when an instruction is committed to programmer visible state, and so on. In the case study we present other abstractions that let us reason about specific programs and data structures. Fig. 3 summarizes the components of the verification environment and their dependencies, the solid arrows can be read from tail to head as ‘is dependent on’.

3.2 A HW/SW Case Study

Our case study has three major components: a simple microprocessor designed in ABEL, a set of programs for the processor written in assembly, and several properties to be verified. All of these components are processed by Maude using internally defined sorts and operators; the dashed arrows in Fig. 3 show which modules are responsible for defining these external representations (i.e. RTL and assembly) that can be processed as terms in Maude.

3.2.1 Simple Microprocessor

Our microprocessor design is based on the classic five-stage, in-order pipeline example from \cite{12} and supports the instruction set architecture (ISA) presented in Table 1. Notable features that are missing from our processor design, but would be expected in a full featured chip, include I/O subsystems, caches, and interrupt handling. All are left for future work. Programs are written under the assumption of unit latencies and are converted into machine code by the Assembler in Fig. 3. Unit assumed latency means that each instruction dynamically executed on the processor should be processed as if all preceding instructions had finished.

Each instruction flows through the five pipeline stages, eventually finishing and updating the programmer-visible state. An instruction is first fetched (IF), then decoded (ID), executed (EX), allowed to access memory (MEM), and,
Table 1
Simple Processor ISA:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>addition</td>
</tr>
<tr>
<td>sub</td>
<td>subtraction</td>
</tr>
<tr>
<td>mul</td>
<td>multiply</td>
</tr>
<tr>
<td>nand</td>
<td>bitwise nand</td>
</tr>
<tr>
<td>lw</td>
<td>load</td>
</tr>
<tr>
<td>sw</td>
<td>store</td>
</tr>
<tr>
<td>beq</td>
<td>branch on equal</td>
</tr>
<tr>
<td>bgt</td>
<td>branch on greater than</td>
</tr>
<tr>
<td>halt</td>
<td></td>
</tr>
</tbody>
</table>

finally, allowed to write the register file (WB). To improve performance, a value that has already been computed, but has not yet made it back to the register file, is forwarded to any instruction that consumes that value. In addition, branches are statically predicted to be not-taken, and instruction fetch continues speculatively until such a prediction is proved wrong, at which point the bad instructions are invalidated. Because memory accesses happen after the execution stage, instructions fetched during the next cycle after a load and dependent on its result are stalled for one cycle in the fetch stage. All of these are interesting features that can be monitored during simulation. If there is a violation of one of these properties, the executing software can help to put the trace into a more understandable context. On the other hand, if a piece of software is producing incorrect results, but the property passes the test, then the programmer might want to look at the software itself.

3.2.2 Software and Verification Examples

The first set of formulas that we specified focus on internal properties of the microprocessor. The major components that we targeted were the forwarding, branching, and stall logic. We checked that each instruction gets the appropriate operand values in the execution stage, that no speculative instructions are allowed to update the state when a branch is taken, and that only certain conditions cause stalls, and these stalls are of a fixed number of cycles. Table 2 lists, and gives a short description of, each of the processor-specific formulas that we verified. The main property we use to verify the forwarding logic is hw-02, which is specified by the LTL formula

\[ \text{op hw-02 : } \rightarrow \text{Formula} \]
\[ \text{eq hw-02 = } \]
\[ \Box p((\text{MEM-WB-valid : 0 asserted}) \]
\[ \wedge (\text{IF-ID-reg-A-used : 3 asserted}) \]
\[ \rightarrow \]
\[ ((\text{alu-input-A : 2 ::= register-value(\text{IF-ID-reg-A : 3, 0}))) \]
\[ \text{...same for second register source operand.} \]

The propositions above show an addition to the language not described previously, and which is used to reason about the pipeline stages. The proposition \'alu-input-A : 2 ::= register-value(\text{IF-ID-reg-A : 3, 0})
evaluates to true when the internal multi-bit node named alu-input-A, which feeds into the EX stage ALU, carried a value 2 cycles ago that equals the current state of the register specified in the ‘A’ position of the instruction word fetched 3 cycles ago. So the LTL formula basically states that for any instruction which completes, the register operands it got in the EX stage of the pipeline was equal to the register file contents when it committed. Of course, a more complete formula would verify that the internal signals referenced actually have their intended meaning. However, as given, the formula would be able to catch most of the common errors with respect to operand forwarding.

Turning to the software component, we started by writing a bubble sort program in the assembly language of our microprocessor. In addition to the properties from Table 2, we added four new ones specific to the software component and its functionality at the meta-level. Meta-level reasoning is facilitated by defining a sort function for integer arrays directly in Maude, together with a map that extracts an array from a software pointer value and length. This makes it possible to compare the mathematical definition of a sorted array with the assembly code sorting program and its data structures. Getting the pointer and length values for an array would generally depend on the application binary interface (ABI) for argument and stack conventions, and also on the array implementation assumed. It would also rely on the specific register allocation map used. The ABI would be defined before any circuit design work is started, and the register map would be known by the engineer who wrote the assembly code or the compiler that generated it.

Correctness of the bubble sort program is checked for by an LTL formula that ensures that whenever there is an inversion in the array, the swap code in the inner loop is executed at some point in the future. The proposition for checking for an inversion in an ordered integer array is given by

Table 2: LTL Formulas for Processor Verification

<table>
<thead>
<tr>
<th>Formula Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hw-01</td>
<td>Instructions only stall in IF.</td>
</tr>
<tr>
<td>hw-02</td>
<td>ALU operands reflect register file at commit.</td>
</tr>
<tr>
<td>hw-03</td>
<td>No stalls longer than 3 cycles.</td>
</tr>
<tr>
<td>hw-04</td>
<td>No state change after halt.</td>
</tr>
<tr>
<td>hw-05</td>
<td>On branch, next committed instruction is target.</td>
</tr>
<tr>
<td>hw-06</td>
<td>r0 is never overwritten.</td>
</tr>
<tr>
<td>hw-07</td>
<td>Taken branch flushes IF, ID, and EX.</td>
</tr>
<tr>
<td>hw-08</td>
<td>Stalls only occur on branch taken and load-use.</td>
</tr>
</tbody>
</table>
op inversion : IntArray -> Prop .
eq inversion(IX:Int) = false .
eq inversion(IX:Int, IY:Int, ILX:IntList) = 
      IX > IY or inversion(IY, ILX) .

In addition to verifying correctness using the inversion check, the *running time* of the function can also be monitored. Run time properties can be stated in terms of the number of cycles, relative to the size of the input array, until the halt instruction is executed. In the case of bubble sort, this should be some constant multiple of the square of the array size. For a more sophisticated program it would probably be convenient to track the running time by inspecting the data structures using an appropriate meta-level abstraction, like the one for integer arrays. In the case of sorting, the running time could be specified in terms of the number of inversions in the input array, or in the case of matrices some measure of sparsity might be useful.

We also checked that *the program is not self-modifying and doesn’t read or write outside the array boundaries*. Table 3 lists all of the properties specific to the sorting program, along with a short description for each one. Checking all of the properties from Tables 2 and 3 for a run of the sorting program with an array of size 13 took about 4 minutes and 5000 simulation cycles on a 2.5GHz/2GB PowerPC machine.

<table>
<thead>
<tr>
<th>Formula Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bs-01</td>
<td>Array inversion leads to swap.</td>
</tr>
<tr>
<td>bs-02</td>
<td>Run time is &lt; 10n² cycles.</td>
</tr>
<tr>
<td>bs-03</td>
<td>No self-modifying code.</td>
</tr>
<tr>
<td>bs-04</td>
<td>No reads or writes outside the array bounds.</td>
</tr>
</tbody>
</table>

The second program we studied was taken from graphics programming. It is part of a shader routine that calculates the normal to a surface. Short assembly code fragments that need to obtain a high percentage of the chip’s theoretical peak performance are routinely written for low level graphics operations. Using this reasoning we set about writing the corresponding code for our microprocessor with the intent that the instructions should be scheduled for highest performance. Therefore, we specified that the code should not have any dynamic pipeline stalls and that it should *complete in less than 100 cycles*. All of the properties we checked are given in Table 4.

In trying to schedule the instructions for high performance, we initially made a mistake that caused the program to compute incorrect results. This was caught by ps-02 and we updated the code. When we fixed the code, a load-use dependency was unintentionally created and caused a pipeline stall.
Table 4
Formulas for Phong Shader

<table>
<thead>
<tr>
<th>Formula Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ps-01</td>
<td>No pipeline stalls in the main code body.</td>
</tr>
<tr>
<td>ps-02</td>
<td>Computes the correct value.</td>
</tr>
<tr>
<td>ps-03</td>
<td>No self-modifying code.</td>
</tr>
<tr>
<td>ps-04</td>
<td>No reads or writes outside of array bounds.</td>
</tr>
<tr>
<td>ps-05</td>
<td>Finishes in less than 100 cycles.</td>
</tr>
</tbody>
</table>

this was immediately found by ps-01 and fixed.

4 Related Work

IBM’s FoCs program [1] turns RCTL [4] assertions into Verilog modules that hook directly into the simulation infrastructure and monitor execution. When the circuit is simulated, each of the monitors watches how the system state and internal signals change over time and reports any problems. SHERLOCK [6] is a second trace checking tool for digital circuits. It can check simulation traces for violations of properties specified in a linear temporal logic augmented with first-order variables, arrays, and queues. The data structures permit property specification at a higher level of abstraction, thus improving usability and reducing low-level specification errors. However, neither of these tools can easily be used to understand hardware/software interactions.

In the embedded design space, there are industrial tools that allow a certain degree of hardware/software debugging. The Xilinx EDK [25], for example, allows the user to debug his/her software with GDB and to scope the internal hardware signals when a breakpoint is triggered. However, it does not work in the other direction: hardware events cannot be used to stop the software. Furthermore, formal analysis is not supported by these debugging tools.

Ptolemy II [13] is a project focusing on embedded system modeling and design. An entire system, including both hardware and software, can be modeled at varying levels of abstraction using the supported ‘models of computation’. The intent is for the modeling of the system to guide its implementation, which in some cases can even be done automatically from the abstracted view. Ptolemy II captures both hardware and software components of an embedded system but is not, in particular, focused on validating a specific implementation of the system.

Trace checking has also been used successfully in software monitoring. For example, see [11,9]. In fact, this has developed into a new software engineering research area called runtime verification [19].
5 Conclusions and Future Work

We have presented a formal semantics in Maude of the ABEL HDL, and have explained how this executable semantics can be used as the basis of a co-verification framework. To the best of our knowledge, both the ABEL semantics, and the capabilities for trace checking ABEL co-designs are new contributions. Our experience so far has been quite encouraging, in that rewriting logic and Maude have given us a flexible framework in which to specify and analyze sophisticated properties of both the hardware and the software and how they interact. However, this is still work in progress open to many improvements and new developments. For example, we should improve the efficiency of the HDL simulator to enable larger designs and also add support for other HDLs, or synthesizable subsets of them. Similarly, we should incorporate known techniques for optimizing trace checking, so as to help performance. Also, adding first order variables to the specification logic would be extremely useful, and shouldn’t pose a problem for trace checking. Furthermore, we should develop a larger and more ambitions suite of real-life case studies.

Our longer-term goal is to develop new methods and tools to formally specify and analyze embedded systems. This paper advances that goal but does not address a number of important issues that, besides the improvements and extensions mentioned above, will be topics of future research. We have pointed out that for many embedded systems, specifying the environments in which they operate and verifying the properties related to their interactions with such environments, is as crucial as specifying and verifying the hardware/software system itself: both tasks should be done together. Although we have addressed some real-time and performance issues in our case studies, a full modeling of environments, though very important, is beyond the scope of this paper. The natural approach for modeling such environments is viewing embedded systems as real-time systems that can be hybrid, and can even be both stochastic and hybrid. Therefore, from a rewriting logic perspective the natural techniques and tools to use will include real-time rewrite theories [20], probabilistic rewrite theories [3], the modeling of stochastic hybrid systems [17], and tools such as Real-Time Maude [21] and the upcoming PMaude [3] and SHYMaude [17]. This should enable us to handle mechanical and sensing interfaces, such as those in many control systems, for example, anti-lock brakes systems. In the terminology of this paper it will also provide a natural extension of the meta-level properties that can be formally specified and analyzed.

Acknowledgement

We thank the referees for their detailed comments on a previous version that have helped us improve the presentation. In addition, we thank Sean Keller
for reviewing a draft of the paper, and Andrew Colombi for a version of the graphics program. This research has been supported in part by ONR grant N00014–02-1-0715.

References


H. Zheng, Overview of the Ptolemy Project, Technical report, Department
of Electrical Engineering and Computer Science, University of California Berkeley
(2003).

edition.


[16] Manuel Clavel and Francisco Durán and Steven Eker and Patrick Lincoln and
Narciso Martí-Oliet and José Meseguer and Carolyn Talcott, “Maude Manual
(Version 2.2),” URL http://maude.cs.uiuc.edu/

[17] Meseguer, J. and R. Sharykin, Specification and Analysis of Distributed Object-
Based Stochastic Hybrid Systems, Technical report, University of Illinois at
Urbana-Champaign, CS Department (2005), to appear in Proc. Hybrid Systems
2006, Springer LNCS.


Verification,” Electronic Notes in Theoretical Computer Science 89, Elsevier,
2003.


Workshop on Rewriting Logic and its Applications.

[22] Sgroi, M., L. Lavagno and A. Sangiovanni-Vincentelli, Formal Models for

[23] Vardi, M., Designing a Property Specification Language (2005), presentation at
UIUC Formal Methods Seminar.


Abstraction and Model Checking of CORE ERLANG Programs in MAUDE

Martin Neuhäußer ¹ Thomas Noll ²

Software Modeling and Verification Group
RWTH Aachen University
D–52056 Aachen, Germany
Fax: +49 241 80 22217

Abstract

This paper provides a contribution to the formal verification of programs written in the concurrent functional programming language ERLANG, which is designed for telecommunication applications. We present a formalization of this language in the Rewriting Logic framework, employing equations for defining abstraction mappings on the state space of the system. Moreover we give a sketch of an implementation in the MAUDE system, and demonstrate the use of the model checker to verify simple system properties.

1 Introduction

In this paper we address the software verification issue in the context of the functional programming language ERLANG [1], which was developed by Ericsson corporation to address the complexities of developing large–scale programs within a concurrent and distributed setting. Our interest in this language is twofold. On the one hand, it is often and successfully used in the design and implementation of telecommunication systems. On the other hand, its relatively compact syntax and its clean semantics support the application of formal reasoning methods.

Here we try to employ fully–automatic model–checking techniques [5] to establish correctness properties of communication systems implemented in ERLANG. While simulation and testing explore some of the possible executions of a system, model checking conducts an exhaustive exploration of all its behaviors. In this paper we concentrate on the first part of the verification procedure, the construction of the (transition–system) model to be checked.

¹ Email: neuhaeusser@cs.rwth-aachen.de
² Email: noll@cs.rwth-aachen.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
More concretely, our approach is based on Core Erlang [3,4], an intermediate language being used in the Erlang compiler which, however, is very close to the original language. We formally describe its semantics employing the Rewriting Logic framework, which was proposed in [9] as a unified semantic framework for concurrency. It has proven to be an adequate modeling formalism for many concrete specification and programming languages [8]. In this approach the state of a system is represented by an equivalence class of terms modulo a given set of equations, and transitions correspond to rewriting operations on the representatives. Hence Rewriting Logic supports both the definition of programming formalisms and, by employing (equational) term rewriting methods, the execution or simulation of concrete systems. We will see that the equations can be used to define abstraction mappings which reduce the state space of the system.

Furthermore we will show that by employing an executable implementation of the Rewriting Logic framework, Maude [6], it is possible to automatically derive the transition system of a given Erlang program, and to verify its properties using the MAUDE model checker.

The remainder of this paper is organized as follows. Section 2 presents the Core Erlang programming language by sketching its syntactic constructs and their intuitive meaning. Section 3 briefly introduces the Rewriting Logic Framework. Finally, Sections 4 and 5 constitute the main part of this paper in which the Rewriting Logic specification of the operational semantics of Core Erlang and its implementation in MAUDE are studied.

2 Core Erlang

ERLANG/OTP is a programming platform providing the necessary functionality for programming open distributed (telecommunication) systems: the functional language Erlang with support for communication and concurrency, and the OTP (Open Telecom Platform) middleware providing ready–to–use components (libraries) and services such as e.g. a distributed data base manager, support for “hot code replacement”, and design guidelines for using the components.

Today many commercially available products offered by Ericsson are at least partly implemented in ERLANG. The software of such products is typically organized into many, relatively small source modules, which at runtime are executed as a dynamically varying number of processes operating in parallel and communicating through asynchronous message passing. The highly concurrent and dynamic nature of such software makes it particularly hard to debug and test by manual methods.

In the following we consider the core version of the ERLANG programming language which has been introduced in [3], and which is used as an intermediate language in the ERLANG compiler. It supports the implementation of dynamic networks of processes operating on data types such as atomic
constants (atoms), integers, lists, tuples, and process identifiers (pids), using asynchronous, call–by–value communication via unbounded ordered message queues called mailboxes. Full ERLANG has several additional features such as distribution of processes (onto nodes), and support for robust programming and for interoperation with non–ERLANG code written in, e.g., C or Java.

The syntax of CORE ERLANG is defined by the context–free grammar\(^3\) in Figure 1. For further explanations, please refer to [3,4].

As an introductory example we consider a short program which implements a simple resource locker, i.e., an arbiter which, upon receiving corresponding requests from client processes (two in this case), grants access to a single resource. Its code is given in Figure 2.

Any CORE ERLANG program consists of a set of modules. Each module is identified by a name (locker in our example), followed by a list of exported functions, and a list of function definitions. In our example the system is defined in one module. It is initialized using the start function. By calling the spawn builtin function, the latter generates two additional processes both running the client function from the locker module. Here the self builtin function returns the process identifier (pid) of the locker process, which is then passed as an argument to the clients such that these are enabled to communicate with the locker.

The locker process runs the locker function in a non–terminating loop. It employs the receive construct to check whether a request has arrived.

\(^3\) To simplify notation, let the placeholder symbols \(c, v, e\), etc. denote elements of the language generated by the respective nonterminals. Moreover, let \(a\) and \(x\) denote atoms and variables, respectively.
The latter is expected to be a pair composed of a \texttt{req} tag and a client process identifier (which is matched by the variable \texttt{Client}). The \texttt{after} clause can be used to specify the behavior of the process when no matching request arrives within a certain time limit; here it is deactivated by giving the \texttt{infinity} atom as the timeout value.

The client is then granted access to the resource by sending an \texttt{ok} flag. Finally, after receiving the \texttt{rel} (release) message from the respective client, the locker returns to its initial state.

A client process exhibits the complementary behavior. By issuing a request, it demands access to the resource. Here again the \texttt{self} built-in function is used to determine the pid of the client process, which is then used by the locker process as a handle to the client. After receiving the \texttt{ok} message it accesses the resource, and releases it afterwards.

The desirable correctness properties of such a system are straightforward:

\textbf{no deadlock}: there exists no cyclic chain of processes waiting for each other to continue, i.e., the locker should always be enabled to receive a new request or a release,

\textbf{mutual exclusion}: no two clients should gain access to the resource at the same time, and

\textbf{no starvation}: all clients enabled to enter the critical section should eventually be granted their demanded access.
Later we will exemplarily see how to check the second property by constructing the transition system of the above program.

3 The Rewriting Logic Framework

The Rewriting Logic framework has been presented by J. Meseguer in [9]. An introduction to this approach together with an extensive bibliography can be found in [8].

Rewriting Logic is intended to serve as a unifying mathematical model and uses notions from rewrite systems over equational theories. It separately describes the static and the dynamic aspects of a concurrent system. More exactly, it distinguishes the laws describing the structure of the states of the system from the rules which specify its possible transitions. The two aspects are respectively formalized as a set of equations and as a (conditional) term rewriting system. Both structures operate on states, represented as (equivalence classes of) \( \Sigma \)-terms where \( \Sigma \) is the signature of the specification language under consideration.

More concretely, in Meseguer’s approach the syntax of Rewriting Logic is given by a \textit{rewrite theory} \( \mathfrak{T} = (\Sigma, E, R) \) where

- \( \Sigma \) is a signature, i.e., a ranked alphabet of function symbols,
- \( E \subseteq T_\Sigma(X) \times T_\Sigma(X) \) is a finite set of equations over the set \( T_\Sigma(X) \) of \( \Sigma \)-terms with variables from a given set \( X \), and
- \( R \) is a finite set of \textit{(conditional) transition rules} of the form

\[
\frac{c_1 \rightarrow d_1 \ldots c_k \rightarrow d_k}{l \rightarrow r}
\]

With regard to the semantics of Rewriting Logic, Meseguer defines that a rewrite theory \( \mathfrak{T} \) \textit{entails} a sequent \([s]_E \rightarrow [t]_E\) and writes

\[
\mathfrak{T} \vdash [s]_E \rightarrow [t]_E
\]

if this sequent can be obtained by a finite number of applications of certain \textit{rules of deduction} which specify how to apply the above transition rules. In this way it is possible to reason about concurrent systems whose states are represented by terms and which are evolving by means of transitions. Here, the states are structured according to the signature and equations are used to identify terms which differ only in their syntactic representation. Later we will see that they can also be employed to define abstraction mappings on the state space.

It is a fact, however, that (conditional) term rewriting modulo equational theories is generally too complex or even undecidable. Hence it is not possible to admit arbitrary equations in \( E \). Following the ideas of P. Viry in [11], we therefore propose to decompose \( E \) into a set of directed equations (that is,

---

4 In Maude, the Rewriting Logic specification is parameterized by a membership equational logic theory with its many–kinded signatures, cf. [2].
a term rewriting system), \( ER \), and into a set \( A \) expressing associativity and commutativity of certain binary operators in \( \Sigma \). Given that \( ER \) is terminating modulo \( A \), rewriting by \( R \) modulo \( E \) can be implemented by a combination of normalizing by \( ER \) and rewriting by \( R \), both modulo \( A \). Here the steps induced by \( R \) represent the actual state transitions of the system while the reductions defined by \( ER \) have to be considered as internal, non-observable computations.

4 Operational Semantics of CORE ERLANG

Given a CORE ERLANG program and an initial expression, we define its transition–system semantics by first considering only local evaluation steps. In Section 4.2 we then extend our transitions in order to capture the concurrent semantics, i.e., the semantics of evaluations which are afflicted with side effects. For an in–depth description of the small–step operational semantics including a formal definition of error handling in CORE ERLANG, see [10].

4.1 Sequential Semantics

Let \( \text{Exp} \) denote the set of valid CORE ERLANG expressions according to Figure 1 and \( \text{fv}(e) \) denote the set of variables with a free occurrence in the term \( e \). We then formalize the semantics of closed CORE ERLANG expressions, i.e., expressions without free variables, by an associated transition system \( T_e \):

Definition 4.1 Let \( e_0 \in \text{Exp} \) and \( \text{fv}(e_0) = \emptyset \). The associated transition system is \( T_e = (E, e_0, \text{Act}_e, \rightarrow_e) \) where \( E := \{e \in \text{Exp} \mid \text{fv}(e) = \emptyset\} \uplus \{\bot\} \) denotes the set of states with a marked initial expression \( e_0 \) and \( \rightarrow_e \subseteq E \times \text{Act}_e \times E \) denotes the transition relation. Transitions are defined according to the following inference rules, and are labeled by actions from the set \( \text{Act}_e \) where \( \tau \in \text{Act}_e \) denotes an unobservable action and the other labels represent the observable evaluation steps. \( \bot \) denotes an undefined value that arises from errors occurring during expression evaluation.

The standard implementation of CORE ERLANG employs a leftmost–innermost evaluation strategy. To formalize argument evaluation, we use the concept of reduction contexts that was first introduced in [7]: intuitively, a reduction context is a CORE ERLANG term with a placeholder symbol “\( \cdot \)” in it, which identifies the subterm where the next evaluation step takes place in case the placeholder is substituted by a reducible expression. Formally, the set of reduction contexts is defined by the context–free grammar in Figure 3. Let \( \text{Ctx} \) denote the set of reduction contexts. The following inference rule then formalizes the leftmost innermost evaluation strategy:

Definition 4.2 Let \( e, e' \in \text{Exp}, \alpha \in \text{Act}_e \) and \( r \in \text{Ctx} \) such that there exist no \( \tilde{e} \in \text{Exp} \setminus \text{Val} \) and \( \tilde{r} \in \text{Ctx} \setminus \{\cdot\} \) such that \( e = \tilde{r}[\tilde{e}] \). Then, the following inference rule is applicable:
Fig. 3. Reduction contexts of Core Erlang expressions

\[ e \xrightarrow{\alpha} e' \quad r \neq \cdot \] (Context)

\[ r[e] \xrightarrow{\alpha} r'[e'] \] (Context)

\[ \text{do } r[e] \xrightarrow{\alpha} r'[e'] \] (Seq)

By the conditions imposed in Definition 4.2, the (Context) rule is only applicable wrt. a maximal reduction context, i.e., \( e \) can be evaluated directly without any further descent into its subterms. Using this concept, the semantics of the sequencing operator is captured by the following inference rules:

\[ e \xrightarrow{\alpha} e' \cdot r \neq \cdot \] (Context)

According to Definition 4.2, the iterated application of the first rule\(^5\) evaluates the first subexpression. As soon as it is completely evaluated, the second rule becomes applicable which formalizes the sequencing semantics in that the result is discarded and evaluation continued with the second subexpression.

4.1.1 Pattern Matching Semantics

To formalize the semantics of Core Erlang pattern matching, substitutions are used to syntactically replace free occurrences of variables (taken from the set \( \text{Var} \)) by their respective values.

**Definition 4.3** A substitution is a partial mapping \( \sigma : \text{Var} \uplus \text{FunName} \rightarrow \text{Const} \). \([x_1 \mapsto c_1, \ldots, x_n \mapsto c_n] \) denotes the finite substitution where \( x_i \) is replaced by the constant \( c_i \) for \( 1 \leq i \leq n \). Substitutions are extended to arbitrary Core Erlang expressions and clauses\(^6\):  

\[ x \sigma := \begin{cases} c_i & \text{if } \sigma(x) = c_i \\ x & \text{if } \sigma(x) = \perp \end{cases} \]

\[ \{e_1, \ldots, e_n\} \sigma := \{e_1 \sigma, \ldots, e_n \sigma\} \]

\[ [e_1, e_2] \sigma := [e_1 \sigma | e_2 \sigma] \]

\[ \text{let } <x_1, \ldots, x_n> = e \text{ in } e' \sigma := \text{let } <x_1, \ldots, x_n> = e \sigma \text{ in } e' \]

where \( \sigma' : \text{Var} \uplus \text{FunName} \rightarrow \text{Const} : x \mapsto \begin{cases} \sigma(x) & \text{if } x \notin \{x_1, \ldots, x_n\} \\ \perp & \text{otherwise} \end{cases} \)

---

\(^5\) Possibly with different reduction contexts

\(^6\) In this paper, we only give an incomplete definition and refer the reader to [10, p. 59ff].
The pattern matching semantics is formalized using these syntactic substitutions: A clause \( p \text{ when } g \rightarrow e \) matches a value \( v \) if (i) a substitution \( \sigma \) exists such that \( p\sigma = v \) holds and (ii) the guard expression \( g\sigma \) evaluates to \('true'\). Formally, this is captured by the following definition.

**Definition 4.4** Let \( p \in \text{Pat}, \ e, \ g \in \text{Exp} \) and \( v \in \text{Val} \). Then

\[
\text{match} : \text{Val} \times \text{Clause} \rightarrow \text{Exp} \cup \{\perp\} :
\]

\[
(v, p \text{ when } g \rightarrow e) \mapsto \begin{cases} e' & \text{if } \exists c \left( v = p[fv(p) \mapsto c] \land g[fv(p) \mapsto c] \rightarrow_{\sigma} 'true' \land e' = e[fv(p) \mapsto c]\right) \\ \perp & \text{otherwise} \end{cases}
\]

The \textsc{Core Erlang} \texttt{case} operator branches control according to a given value:

\[
\forall i. (\text{match}(v, cl_i) = e' \land \forall j < i. \text{match}(v, cl_j) = \perp) \rightarrow_{e} \text{case } v \text{ of } cl_1 \cdots cl_k \text{ end }
\]

\[
\text{case } v \text{ of } cl_1 \cdots cl_k \text{ end } \rightarrow_{\text{exc}} '\text{error}', '\text{case_clause}' \rightarrow_{e} \perp
\]

(Case_1)

(Case_2)

4.1.2 Message Passing

Given a mailbox \( q \) and a nonempty sequence of clauses \( cl_1, \ldots, cl_k, \ k \geq 1 \), the predicate \( q\text{match}(q, cl_1, \ldots, cl_k) \) holds iff at least one message in \( q \) matches one of the clauses \( cl_i \). Within the sequential part of the semantics, we cannot infer the contents of the process’ mailbox. Therefore we nondeterministically guess one possible prefix \( q \cdot c \). In the inference rule \( \text{Rcv}_1 \), \( c \) denotes the first message that matches at least one of the clauses \( cl_i \); the condition \( \neg q\text{match}(q, cl_1, \ldots, cl_k) \) is fulfilled iff \( q \) does not contain a matching message; in addition, we assume a successful \(^7\) evaluation of the \texttt{case} term which captures the pattern matching semantics wrt. the message \( c \) and the clauses \( cl_1, \ldots, cl_k \):

\[
\neg q\text{match}(q, cl_1, \ldots, cl_k) \rightarrow_{e} c \in \mathbb{N} \cup \{'\text{infinity}'\} \\
\text{case } c \text{ of } cl_1 \cdots cl_k \text{ end } \rightarrow_{e} e'
\]

(Rcv_1)

If no matching message is received within the time bound \( c_t, e_t \) is evaluated. According to our time–abstract model, this is formalized by nondeterminism:

\[
\neg q\text{match}(q, cl_1, \ldots, cl_k) \rightarrow_{e} c_t \in \mathbb{N} \\
\text{receive } cl_1 \cdots cl_k \text{ after } c_t \rightarrow_{e} e_t \rightarrow_{\text{timeout}(q), e} e_t
\]

(Rcv_2)

4.1.3 Higher–Order Concepts

Function abstractions are treated as values and are applied to a sequence of arguments using the \texttt{apply} operator. To capture its semantics, we replace

\(^7\) Matching failures would lead to a non–\(\tau\)–transitions.
every free occurrence of an argument variable by the corresponding value:

\[
\sigma := [x_1 \mapsto c_1, \ldots, x_n \mapsto c_n]
\]

\[
\text{apply } \text{fun}(x_1, \ldots, x_n) \rightarrow e(c_1, \ldots, c_n) \xrightarrow{\tau} e\sigma
\]  

(App1)

Note that the argument evaluation is specified implicitly by the reduction contexts defined in Figure 3.

The \texttt{letrec} operator supports on-the-fly declaration of local functions. Its semantics is formalized by the following rule:

\[
\forall i \leq m. \quad e_i' := \text{letrec} \ldots a_j/n_j = \text{fun}(x_j) \rightarrow e_j \ldots \text{in} \ e \xrightarrow{\tau} e \quad \text{(LRec)}
\]

The function names \(a_i/n_i\) are treated as variables that range over the special domain of function abstractions. Evaluation of a \texttt{letrec} expression yields a new binding whose scope reaches over \(e\) and \(e_1, \ldots, e_m\). This extended scope is reflected in the semantics by propagating the \texttt{letrec} statements into the bodies of the function abstractions (cf. the definition of \(e_i'\) in the premise).

### 4.2 Concurrent Semantics

To reason about concurrent systems implemented in \textsc{Core Erlang}, we now lift the semantics of sequential expressions to the system level where also side effects, i.e., process spawning and communication, are considered.

**Definition 4.5** The set of processes is given by \(P := \text{Exp} \uplus \{\bot\} \times \mathbb{N} \times \text{Const}^* \times 2^\mathbb{N} \times \mathbb{B}\). A process is denoted by \((e, i, q, L, t) \in P\) where \(e\) is the expression to be evaluated, \(i\) is the process identifier, \(q\) the process’ mailbox, \(L\) the set of linked processes, and \(t\) is a flag controlling exit behavior.

To describe the possible behaviors of an entire system, we extend this definition to sets of processes:

**Definition 4.6** A finite subset \(S \in 2^P\) is called a process system. \(S\) is well formed if, for every \(p, p_1, p_2 \in S\),

\[
p_1 \neq p_2 \Rightarrow \text{Pid}(p_1) \neq \text{Pid}(p_2) \quad \text{and} \quad \text{Links}(p) \subseteq \bigcup_{p' \in S, p' \neq p} \text{Pid}(p')
\]

Here, \(\text{Pid} : P \rightarrow \mathbb{N}\) and \(\text{Links} : P \rightarrow 2^\mathbb{N}\) denote the projection on the process identifier and the link component respectively.

By considering process systems as states of the modeled reactive system, we obtain the following transition-system semantics:

**Definition 4.7** Let \(p_0 \in P\) be an initial process. The corresponding transition system is defined as \(T_s = (S, S_0, Act_s, \rightarrow_s)\) where \(S := 2^P\) is the set of states with initial state \(S_0 := \{p_0\}\), and where \(\rightarrow_s \subseteq S \times Act_s \times S\) denotes the transition relation labeled by actions from the set \(Act_s\). Again, \(\tau \in Act_s\) denotes a local (unobservable) evaluation step and the other labels reflect side effects.
The inference rules which formalize local evaluation steps are directly lifted from the sequential level to the system layer semantics:

\[
\begin{align*}
S & \cup \{(e, i, q, L, t)\} \\
\rightarrow_{e} & \quad S \cup \{(e', i, q, L, t)\}
\end{align*}
\]

(Sequ)

Here, the union operator in the conclusion reflects the interleaving semantics in a set theoretic way.

In most cases, the nondeterminism that was introduced in Section 4.1 can be resolved when considering process systems, where information about the system state is available.\(^8\)

4.2.1 Creation of New Processes

New processes are created by evaluating the `spawn` built-in function. Within the sequential layer, the pid \(j\) of the newly created process cannot be inferred; therefore it is chosen nondeterministically and reflected by the transition label which indicates the side effect:

\[
\text{call 'erlang': spawn(a1, a2, c)} \quad \text{spawn(a1, a2, c)} \leadsto j
\]

The actual process creation is captured by the system layer rules where the new process term is introduced and the pid is fixed:

\[
e \quad \text{spawn(a1, a2, (c1, ..., c_k))} \quad e' \quad j \notin \text{Pids}(S) \cup \{i\}
\]

(Spawn\(_1\))

4.2.2 Message Passing

Sending messages affects the state of the sender and the receiver; this is captured on the system layer by the inference rule (Send\(_1\)):

\[
e \quad \text{send(i, j, c)} \quad e' \\
\rightarrow_{e} & \quad S \cup \{(e', i, q_i, L_i, t_i), (e_j, j, q_j, L_j, t_j)\}
\]

(Send\(_1\))

The nondeterminism that was introduced to formalize the local evaluation of a `receive` term is fully resolved on the system layer:

\[
e \quad \text{recv(q1, c)} \quad e' \\
\rightarrow_{e} & \quad S \cup \{(e', i, q_1, L, t)\}
\]

(Recv)

Here the application of the `qmatch` predicate in the premise of inference rule (Recv\(_1\)) assures that the first matching message \(c\) is chosen from the mailbox.

\(^8\) An exception is the creation of new processes, where a new identifier is chosen nondeterministically.
4.3 State–Space Reduction

The transition system $T_s$ captures the semantics of a concurrent CORE ERLANG program by considering local evaluation steps as well as those afflicted with side effects. To reason about the behavior of the whole system, we are primarily interested in inter–process communication and the creation and termination of processes. Therefore we ignore local $\tau$–evaluation steps:

**Definition 4.8** The equivalence relation $\sim \subseteq S \times S$ is defined by $\sim := \tau'_s \ast$.

By migrating to the quotient transition system $T_{\sim}$, we abstract from local $\tau$–evaluation steps and only observe the processes’ interaction:

**Definition 4.9** Let $T_{\sim} := \langle S_{\sim}, [S_0]_{\sim}, \text{Act}_{\sim}, \rightarrow_{\sim} \rangle$ denote the quotient transition system where $S_{\sim} := \{ [S]_{\sim} | S \in S \}$ denotes the set of states, $[S_0]_{\sim}$ is the initial state, $\text{Act}_{\sim} := \text{Act}_s \setminus \{ \tau \}$ is the set of actions, and where the transition relation $\rightarrow_{\sim} \subseteq S_{\sim} \times \text{Act}_{\sim} \times S_{\sim}$ is defined by

$$[S]_{\sim} \xrightarrow{\alpha} [T]_{\sim} :\iff \exists S', T' \in S. \quad S \xrightarrow{\tau}^* S' \quad \alpha_s \xrightarrow{\tau}^* T.$$

Regarding the possible state space reduction, the following lemma holds:

**Lemma 4.10** Let $S = \{ p_1, \ldots, p_n \} \in S$ a process system, $p_j = (e_j, i_j, q_j, L_j, t_j)$ for $1 \leq j \leq n$. Further, let $k_j$ denote the number of consecutive $\tau$–steps of process $p_j$ before reaching a $\tau_e$–normal form. The cardinality of $\text{Post}^*(S, \tau) := \{ S' \in S | S \xrightarrow{\tau}^* S' \}$ is then bounded by: $|\text{Post}^*(S, \tau)| \leq \prod_{1 \leq j \leq k} (k_j + 1)$.

According to Lemma 4.10, the $\prod_{1 \leq j \leq k} (k_j + 1)$ successor states of a process system $S$ are represented by one equivalence class within $T_{\sim}$. Most importantly, in $T_{\sim}$ we do no longer consider interleaving of $\tau$–evaluation steps which is natural given that those transitions do not affect other processes at all.

5 Implementation in MAUDE

The small step operational semantics introduced in Section 4 relates a given CORE ERLANG program with an initial expression $e_0$ to a quotient transition system $T_{\sim}$ thereby formalizing the possible system behaviors. In order to automatically reason about properties of such systems, in this section we use a Rewriting Logic specification of our semantics to operationalize the computation of $T_{\sim}$.

According to the Rewriting Logic framework, in Figure 4 we first define the signature of processes and process systems: to implement our semantics, we extend the representation of a process $P = \text{Exp} \uplus \{ \bot \} \times \mathbb{N} \times \text{Const}^* \times 2^\mathbb{N} \times \mathbb{B}$ (see Definition 4.5) by three additional components:

(i) a process $\text{Label}$ that summarizes the process’ state,
(ii) a $\text{SysResult}$ term indicating the result of the last side effect and
(iii) the set $\text{ModEnv}$ of known function declarations.

Accordingly, process systems are multisets of $\text{Process}$ terms, represented by the associative and commutative list constructor "$||$".

5.1 Core Erlang Signature

Apart from these basic operator declarations, a slightly restricted\(^9\) CORE ERLANG syntax (cf. Section 2) is specified as a many–kinded signature so that it can be parsed by the MAUDE interpreter. To allow arbitrarily many arguments (e.g., when considering apply expressions), we define a flattened argument list operator:

\[
\text{subsort} \text{ Expr} < \text{NeExprList}.
\]

\[
\text{op } _,_,_ : \text{NeExprList} \text{ NeExprList} \rightarrow \text{NeExprList} [\text{ctor assoc}].
\]

As a consequence, unbounded argument lists are internally replaced by a single argument of sort $\text{NeExprList}$. Note however, that due to the flattened concatenation operator "$\_,\_\$" this does still allow to parse arbitrary CORE ERLANG expressions.

In the MAUDE system, terms are built using many–kinded signatures. Here a sort denotes a semantic concept whereas kinds refer to the notion of a sort in the context of traditional many–sorted signatures.

On the syntactical level, each well–formed term is assigned a kind whereas its affiliation to a designated sort has to be inferred using membership axioms of an underlying membership equational theory (cf. [2] for details).

5.2 Specification of the Process Layer

In principle, mapping the quotient transition–system semantics to a Rewriting Logic specification is straightforward: the equivalence relation $\sim := \leftrightarrow^*$ is transformed into an equational theory that models local $\tau$–evaluation steps. Operationally, these equations are split into a set of equations $A$ that is inferred according to associativity and commutativity attributes of the operators and a set of directed equations $ER$ that constitute a terminating and confluent term rewrite system.

---

\(^9\) Additional whitespaces are required.
Deviating from the CORE ERLANG semantics, in our implementation we specify the sequential semantics already wrt. the term representation of a single process instead of considering closed CORE ERLANG expressions only\(^\text{10}\). In the underlying sequential semantics, we use nondeterminism to model side effect afflicted evaluation steps. However, the implementation requires the set \(ER\) of directed equations to converge modulo \(A\). We therefore augment the process terms with a \texttt{Label} and a \texttt{SysResult} component to avoid nondeterministic choices.

As a first example, consider the semantics of the sequencing operator \texttt{do}:

```plaintext
var ESL : StopLabel .

ceq < tau | RES | do EX1 EX2 | PID | MBOX | LINKS | TRAP | ME > =
< #filterExit(ESL) | RES1 | do EX1' EX2 | PID | MBOX | LINKS | TRAP | ME >
if not(EX1 :: Const)
\/
< ESL | RES1 | EX1' | PID | MBOX | LINKS | TRAP | ME > :=
< tau | RES | EX1 | PID | MBOX | LINKS | TRAP | ME > .
```

A necessary condition for the applicability of this directed equation is that the first subexpression \(EX1\) is not a value yet. Only then, further evaluation takes place within the second condition yielding a new expression \(EX1'\). The sort of the label variable \(ESL\) is crucial here: If the evaluation of \(EX1\) yields a side effect, the process' label changes to a term of sort \texttt{StopLabel} thereby reaching a normal form wrt. the equational rewriting. Therefore, the result of the side effect is not guessed non–deterministically within the “local” process layer, but is resolved later by the rewrite rules that operate on these normal forms.

5.2.1 Substitutions

The pattern matching semantics is based on syntactic substitutions on CORE ERLANG expressions; a single binding is represented as a term of sort \texttt{Binding}. An environment is then constructed as a comma separated associative and commutative list of such variable bindings:

```plaintext
sort Binding Env .

subsort Binding < Env .
op #empty-env : -> Env [ctor] .
op _,_ : Env Env --> Env [ctor assoc comm id: #empty-env] .
```

Given a CORE ERLANG expression, the \texttt{#subst} function specifies the substitution of free variables according to a given environment by recursively descending into the subterms. Therefore, the base cases include

```plaintext
eq #subst(V, (V --> C), ENV) = C .
eq #subst(E, ENV) = E [owise] .
```

Here, the first rule specifies the substitution of a variable that gets bound according to the environment. Note that since AC matching is involved, we

\(^\text{10}\)Therefore argument evaluation cannot be specified by MAUDE’s evaluation strategies.
can assume without loss of generality that the corresponding Binding term is the first binding in the environment. When considering Core Erlang expressions that introduce new variable bindings, the environment has to be shrunked accordingly:

\[
\text{ceq } \#\text{subst}(\text{let } VS = EX1 \text{ in } EX2, ENV) = \text{let } VS = \#\text{subst}(EX1, ENV) \text{ in } \#\text{subst}(EX2, ENV1) \\
\text{if } VSET := \#\text{projectVarSet}(VS) \land ENV1 := \#\text{restrictEnv}(VSET, ENV) .
\]

The scope of a new binding introduced by evaluating a let expression ranges over the EX2 expression only. Therefore, the substitution is applied to the EX1 subterm without any modification. According to the semantics of \texttt{let}, in EX2 free occurrences of variables of the sequence VS are bound and may not be substituted. In ENV1, such “critical” bindings are removed thereby excluding the variables bound by the \texttt{let} context. Based on the variable sequence VS, the functions \texttt{#projectVarSet} and \texttt{#restrictEnv} compute the set of newly bound variables and shrink the environment accordingly.

5.2.2 Pattern Matching

Pattern matching is formalized by the partial function \texttt{#match}. Matching failures are represented by introducing a constant \texttt{#nomatch} with a new sort \texttt{Env?} which is declared as a supersort of variable environments.

During pattern matching, we recursively descend into the subterms of the given pattern and try to construct a unifying variable environment. In case of a clash failure\(^{11}\), the \texttt{#nomatch} constant is included in the environment indicating the matching failure. Therefore we have:

\[
\begin{align*}
\text{subsort } & \texttt{Env < Env?} . \\
\text{op } & \texttt{#match : Pat Const ~> Env} . \\
\text{eq } & \texttt{#match(VAR, CONST)} = (\text{VAR --> CONST}) . \\
\text{eq } & \texttt{#match(CONST, CONST)} = \#\text{empty-env} . \\
\text{eq } & \texttt{#match([PAT1|PAT2], [C1|C2])} = \#\text{match(PAT1, C1)}, \#\text{match(PAT2, C2)} . \\
\text{eq } & \texttt{#match(PAT, CONST)} = \#\text{nomatch [owise]} .
\end{align*}
\]

According to Core Erlang’s semantics, all variables in a pattern are free; therefore we can directly construct a variable binding when matching against a constant value. In the same way, two identical values match each other without entailing a new binding.

In more complex patterns like lists, the \texttt{#match} function recursively descends into the corresponding subterms. Finally the fourth equation covers matching failures and is – according to the \texttt{owise} attribute – applicable only if none of the other \texttt{#match}–equations allows to continue the matching process.

The \texttt{#subst} and \texttt{#match} functions allow to specify the semantics of Core Erlang’s pattern matching operations. A Core Erlang \texttt{case} expression has the form \texttt{case v of cl\(_1\) \cdots cl\(_k\) end} where evaluation continues with the expression \texttt{e\(_i\)} of the first matching clause \texttt{cl\(_i\) = p\(_i\) \texttt{when } g\(_i\) \texttt{-> } e\(_i\)} in the sequence \texttt{cl\(_1\), \ldots, cl\(_k\)\texttt{. This is specified by the following directed equation:}}

\(^{11}\text{Due to Core Erlang’s pattern matching semantics occur failures cannot happen.}\)
According to the first condition, the equation is applicable only if the pattern \( \text{PAT} \) matches the value \( \text{C} \). The second condition formalizes the guard evaluation. Depending on its result, evaluation continues either with the clause’s right hand expression or – if the guard does not evaluate to ‘true’ – a modified case term is reevaluated with the failed clause removed.

Furthermore, the pattern matching itself may fail:

\[
\begin{align*}
\text{eq} \quad & \tau \text{ if not } \langle \text{match}(\text{PAT}, \text{C}) \rangle. \\
\text{env} \quad & \text{case } \text{C} \text{ of PAT when GUARD -> EX CLAUSES end } | \text{PID} | \text{MBOX} | \text{LINKS} | \text{TRAP} | \text{ME} > \\
& \text{if ENV := } \langle \text{exception}(\text{exit}, \langle \text{atom("normal")} \rangle) | \text{EX1} | \text{EX2} | \text{PID} | \text{MBOX} | \text{LINKS} | \text{TRAP} | \text{ME} > := \\
& \langle \text{tau} | \text{no-res} | \text{EX2} | \text{PID} | \text{MBOX} | \text{LINKS} | \text{TRAP} | \text{ME} > . \\
\text{else case } \text{C} \text{ of CLAUSES end fi .}
\end{align*}
\]

If the first clause in the sequence does not match the constant \( \text{C} \), the environment computed by the \#match function contains the \#nomatch constant; therefore \#match(\text{PAT}, \text{C}) is of sort \text{Env}? and the membership formula is not fulfilled. In this case, the failed clause is removed and matching is continued with the tail of the clause sequence.

### 5.3 System layer semantics

The directed equations introduced so far describe unobservable local evaluation steps. The observable transitions that we consider now model interactions between Erlang processes and are formalized by rewriting rules. From an operational point of view, these rules operate on normal forms wrt. equational rewriting.

The rules that cover process creation provide a first example. When symbolically evaluating an expression \text{call 'erlang':}'spawn'(a_1, a_2, c), the process’ label is changed to indicate the side effect. This yields a normal form wrt. equational rewriting; the system–layer rules then compute a new identifier and extend the process environment:

\[
\begin{align*}
\text{eq} \quad & <\text{call atom("erlang")}:\text{atom("spawn")}(a_1, a_2, \text{C})|\text{PID} | \text{MBOX} | \text{LINKS} | \text{TRAP} | \text{ME} > \quad = \quad <\text{pid(INT)!C}|\text{no-res}|\text{call atom("erlang")}:\text{atom("spawn")}(a_1, a_2, \text{C})|\text{PID} | \text{MBOX} | \text{LINKS} | \text{TRAP} | \text{ME} > .
\end{align*}
\]

When evaluating the expression \text{call 'erlang':}!'(\text{Pid}, \text{Msg}), the message is appended to the mailbox of the process identified by \text{Pid}. This information is passed to the corresponding rewrite rule from within the equational theory:

\[
\begin{align*}
\text{eq} \quad & \langle \text{call atom("erlang")}:\text{atom("!")}(\text{int(INT)},\text{C})|\text{PID} | \text{MBOX} | \text{LINKS} | \text{TRAP} | \text{ME} > \\
& \quad = \quad \langle \text{pid(INT)!C}|\text{no-res}|\text{call atom("erlang")}:\text{atom("!")}(\text{int(INT)},\text{C})|\text{PID} | \text{MBOX} | \text{LINKS} | \text{TRAP} | \text{ME} > .
\end{align*}
\]
Neuhaeusser and Noll

The transition rules of the system level operate on this normal form by extracting the receiver’s PID and the message from the process’ label and appending the message to the receiver’s mailbox:

\[
\text{cr1} \quad (\text{SL}, \langle \text{pid}(\text{INT})!C | \#\text{no-res} | \text{EX} | \text{PID} | \text{MBOX} | \text{LINKS} | \text{TRAP} | \text{ME} \rangle \ || \ \langle \text{EL1} | \#\text{no-res} | \text{EX1} | \text{pid}(\text{INT}) | \text{MBOX1} | \text{LINKS1} | \text{TRAP1} | \text{ME1} \rangle \ || \ \text{PRCS}, \text{ME'}, \text{PIDS}) \\
\Rightarrow (\text{sys-sendmsg}(\text{PID}, \text{pid}(\text{INT}), C), \langle \tau | \#\text{res-send}(\text{true}) | \text{EX} | \text{PID} | \text{MBOX} | \text{LINKS} | \text{TRAP} | \text{ME} \rangle \ || \ \langle \text{EL1}' | \#\text{no-res} | \text{EX1} | \text{pid}(\text{INT}) | \text{MBOX1} | \text{LINKS1} | \text{TRAP1} | \text{ME1} \rangle \ || \ \text{PRCS}, \text{ME'}, \text{PIDS})
\]

\[
\text{if EL1'} := \text{if EL1 == waiting} \text{ or } (\text{EL1 == blocked}) \text{ then } \tau \text{ else } \text{EL1 fi .}
\]

5.3.1 Completeness

Operationally, a given process environment is first normalized by equational rewriting. Then, the system layer’s transition rules are applied to these normal forms. The following result from [10] shows that this implementation is complete, i.e., by first normalizing the process environment, no transitions get lost that would otherwise be possible.

**Theorem 5.1 (Completeness)** Let \( \mathcal{X} = (\Sigma, E, R) \) be the Rewriting Logic specification of the CORE ERLANG semantics where \( E = ER \uplus A \) and \( s, t, t', t'' \) denote process environments. Then it holds:

According to Theorem 5.1, given a process environment \( s \), any applicable transition rule is also applicable to the \( ER \) normal form \( s \downarrow \) of \( s \) and the resulting terms are again in the same equivalence class (modulo \( ER \)).

6 Verifying system properties

Based on the transition–system model of a given CORE ERLANG program that is computed by the interpreter, MAUDE’s integrated LTL model checker allows to verify system properties: terms of sort ProcessEnvironment constitute the states of the computed transition system. Because we focus on inter–process communication in a distributed environment, the relevant properties refer to the system’s transitions instead of its states. Therefore, each state is augmented by the transition label of the incoming transition. This label is reflected as a term of sort SysLabel in the first component of each ProcessEnvironment.

With this approach, it is possible to define state predicates (for an in–depth discussion of the MAUDE model checker, refer to [6]). For example, the send predicate is defined as follows:

\[
\text{op send : Int Int Const -> Prop .} \\
\text{eq (sys-sendmsg(pid(P1),pid(P2),C),PRCS,ME,PIDS) |= send(P1,P2,C) = true .}
\]

It takes the identifiers of the sender and receiver and the message as arguments.
In the equation, the validity of the predicate is defined wrt. the SysLabel component of the state which reflects the action that led into this state. Therefore it determines that the parameterized predicate \texttt{send(P1, P2, C)} holds iff the incoming transition label reflects the corresponding send operation.

An essential issue when reasoning about concurrent systems is the possibility to specify fair scheduling strategies. In our approach, we impose fairness constraints as LTL premises. For an a priori given set of processes, the \texttt{scheduler} function evaluates to an LTL–premise specifying a fair scheduling strategy. It has the form

\[
\varphi_{\text{scheduler}}(i_1, \ldots, i_n) := \bigwedge_{k=1}^{n} \square \left( p_{\text{running}}(i_k) \rightarrow p_{\text{running}}(i_k) \bigcup \left( p_{\text{scheduled}}(i_k) \vee p_{\text{blocked}}(i_k) \right) \right)
\]

where \(i_1, \ldots, i_n\) denote the included processes and \(p_{\text{running}}(i_k)\) states the existence of the \(k\)th process. The predicate \(p_{\text{scheduled}}(i_k)\) holds iff the process with identifier \(i_k\) caused the last system level transition, and \(p_{\text{blocked}}(i_k)\) is valid iff the corresponding process is blocked during message reception. Intuitively, it states that whenever a process exists, it is scheduled sometime later or it becomes blocked waiting for message reception.

Considering again the mutual exclusion program for two competing processes from Figure 2, we can now successfully verify the mutual exclusion property specified by the following LTL formula:

\[
\varphi_2 = \varphi_{\text{scheduler}}(0,1,2) \rightarrow \square \left( p_{\text{send}}(0,1,'ok') \rightarrow \left( \neg p_{\text{recv}}(2,'ok') \bigcup p_{\text{recv}}(0,\{\text{rel},1\}) \right) \right) \\
\wedge \square \left( p_{\text{send}}(0,2,'ok') \rightarrow \left( \neg p_{\text{recv}}(1,'ok') \bigcup p_{\text{recv}}(0,\{\text{rel},2\}) \right) \right)
\]

Recapitulatory, in our approach we use Meseguer’s Rewriting Logic framework and the MAUDE system to automatically compute the transition–system model of a given CORE ERLANG program. On this basis, we define state predicates and use MAUDE’s integrated LTL model checker to verify certain linear–time properties.

References


URL \texttt{http://www.erlang.se/workshop/carlsson.ps}

R. Virding, Core Erlang 1.0.3 language specification (2004), http://www.it.uu.se/research/group/hipe/corerl/doc/core_erlang-1.0.3.pdf.


Abstraction and Completeness for Real-Time Maude

Peter Csaba Ölveczky\textsuperscript{a, b} and José Meseguer \textsuperscript{b}

\textsuperscript{a} Department of Informatics, University of Oslo
\textsuperscript{b} Department of Computer Science, University of Illinois at Urbana-Champaign

Abstract

This paper presents criteria that guarantee completeness of Real-Time Maude search and temporal logic model checking analyses, under the maximal time sampling strategy, for a large class of real-time systems. As a special case, we characterize simple conditions for such completeness for object-oriented real-time systems, and show that these conditions can often be easily proved even for large and complex systems, such as advanced wireless sensor network algorithms and active network multicast protocols. Our results provide completeness and decidability of time-bounded search and model checking for a large and useful class of dense-time non-Zeno real-time systems far beyond the class of automaton-based real-time systems for which well known decision procedures exist. For discrete time, our results justify abstractions that can drastically reduce the state space to make search and model checking analyses feasible.

\textit{Key words:} Rewriting logic, real-time systems, object-oriented specification, formal analysis, abstraction, completeness

1 Introduction

Real-Time Maude \cite{15, 12} occupies a useful middle ground between automaton-based real-time formal tools providing decision procedures, such as Uppaal \cite{7, 1}, Kronos \cite{18}, and HyTech \cite{6}, and general modeling and simulation tools which can be applied to much broader classes of systems, clearly beyond the pale of the above decision procedures, but that have quite limited analytic features.

In terms of expressiveness and the generality of the systems that can be specified, Real-Time Maude is clearly in the same category as modeling and simulation tools. But in terms of analytic power, it is much closer to the above-mentioned automaton-based tools, although with some limitations. The limitations in question have to do with the fact that, since we are dealing with general classes of infinite-state real-time systems for which no decision procedures are known, some of the formal analyses are \textit{incomplete}.
We shall call an analysis method (for example, Real-Time Maude’s (timed) breadth-first search, or LTL model checking) *sound* if any counterexample found using such a method is a real counterexample in the system; that is, if the method does not yield *spurious* counterexamples. In this precise sense, all the formal analysis methods supported by Real-Time Maude are indeed sound. We call an analysis method *complete* if the fact that a counterexample is never found using the method actually means that no such counterexamples exist for the analysis in question. For example, (timed) breadth-first search for a violation of an invariant property will be complete if the fact that the search never finds any such violation (assuming an idealized machine) actually means that no such violations exist. Similarly, the LTL model checking of a time-bounded property \( \varphi \) will be complete if the fact that a model checker responds with the answer *true* actually means that \( \varphi \) holds in the system, and therefore that no counterexamples violating \( \varphi \) exist.

For *discrete time* systems, completeness can be purchased, but at a very heavy price. The point is that, if time is discrete, an analysis can exhaustively visit *all* time instants. This makes breadth-first search for the violation of an invariant complete. For general real-time systems outside the scope of decision procedures, unbounded time leads to infinite state spaces that cannot be model checked with the standard algorithms. However, under very reasonable assumptions satisfied by practically all discrete-time systems of interest, *time-bounded* LTL properties do have *finite* state spaces that can indeed be model checked, yielding a complete decision procedure for such properties. The heavy price of achieving completeness this way has to do with the fact that visiting all discrete times typically leads to a state space explosion that renders many formal analyses unfeasible.

For *dense time* systems, achieving completeness by visiting all times is indeed quite hopeless. The problem, of course, is that if time advances from, say, time \( r \) to time \( r + r' \) with \( r' > 0 \) there is an *infinite* set of intermediate times \( r'' \) with \( r \leq r'' \leq r + r' \) that will not be visited if the clock ticks by the positive amount \( r' \). Real-Time Maude deals with this problem by making all analyses relative to a *time sampling strategy*. That is, only those times chosen by the strategy are used to tick the time; and only those behaviors where the states are those corresponding to the chosen times are analyzed. This has several important advantages. First, such time sampling strategies make “tick” rewrite rules that advance time *executable*, whereas if time can tick by any intermediate amount tick rules typically become nondeterministic. Second, under very reasonable assumptions about the time sampling strategy and about the system, it becomes possible for a timed breadth-first search, that only visits states at the chosen times, to examine all such states to see if an invariant is violated. Similarly, even though the state space of even time-bounded LTL properties is now infinite, the subspace obtained by restricting the times to those chosen by the strategy is typically finite, and can indeed be model checked. Of course, since only states with the chosen times are visited,
these formal analyses, though sound, are in general incomplete.

The question that this paper raises and provides practical answers to is: under what conditions on the real-time system and on the time sampling strategy can completeness be guaranteed? That is, under what conditions does breadth first search become a complete semi-decision procedure for violations of invariants, and does LTL model checking of time-bounded properties (excluding the next operator $\bigcirc$) become a complete decision procedure for such properties even when time is dense?

Our experience in specifying and analyzing a substantial collection of real-time systems has guided our search for criteria guaranteeing completeness. Indeed, as we further explain in this paper, many practical and nontrivial real-time systems, including many of the systems that we had previously analyzed, satisfy the requirements that we present. If they used dense time, this shows in hindsight that many of our analyses were in fact complete. But even for discrete time this provides new completeness guarantees, since to avoid state explosions most of our analyses were made relative to a time sampling strategy also in that case.

The key insight is that many real-time systems are “time-robust” in a sense that we make mathematically precise in Section 3.1. A typical example of a time-robust system is one where each instantaneous transition is triggered by the expiration of a timer or by the arrival of a message with a given transmission delay. The typical time sampling strategy used for such systems is a maximum time elapse (mte) strategy, that advances time as much as possible to reach the next time at which an instantaneous transition will become enabled. We give simple conditions for time robustness, and also for what we call “tick-stabilizing” state properties that do not change arbitrarily in between mte time ticks. Our main result is that for time-robust systems and tick-stabilizing properties the mte time sampling strategy is indeed complete.

We prove this property using basic concepts such as abstraction and stuttering bisimulation. The point is that there are two real-time rewrite theories (and therefore two Kripke structures) involved: the original one, and the one in which time advance is restricted by the mte strategy. The behaviors of the restricted theory are of course a subset of those of the original theory. We can view the restricted theory as providing an abstraction of the original system, similar in nature to those considered in partial order reduction methods [3]. As in partial order reduction, the key point is to show that (after excluding pathological Zeno behaviors) the restricted, more abstract system is stuttering bisimilar to the original one. This result (the proof of which is given in [13]) yields as a direct corollary the fact that both systems satisfy the same LTL properties (excluding the next operator $\bigcirc$). We also show how this result can be naturally restricted to time bounded LTL properties. This ensures all of our desired completeness guarantees.

Of course, to guarantee completeness one has to check the time robustness of the specification and the tick-stabilizing nature of the relevant state prop-
That is, a complete analysis decomposes into two tasks: (1) a standard formal analysis in Real-Time Maude under the \textit{mte} time sampling strategy; and (2) the checking of appropriate proof obligations ensuring time robustness and tick-stabilization. We address the pragmatic question of finding simple and easy-to-check proof obligations to accomplish task (2). Specifically, we show that for a very large class of systems, namely, real-time object-oriented systems made up of objects that can communicate asynchronously by message passing, if one follows the specification methodology advocated in [15], there are indeed quite simple proof obligations that, if met, discharge task (2).

We illustrate the ease of checking such proof obligations by means of several nontrivial examples. Finally, since some of the systems that we have analyzed in the past involve the use of probabilistic algorithms, we also include a discussion of how our results can be interpreted for such systems.

The paper is organized as follows: Section 2 gives some background on Real-Time Maude and stuttering simulations. Section 3 defines time-robustness, timed fair behaviors, tick-stabilization and tick-invariance of properties, and proves that unbounded and time-bounded LTL $\\{\Box\}$ model checking using the \textit{mte} time sampling strategy is complete for systems satisfying the above requirements. Section 4 shows how proving those requirements reduces to proving very simple properties for object-based Real-Time Maude specifications, and shows that these properties can easily be proved for our large Real-Time Maude applications.

2 Preliminaries on Real-Time Maude and Stuttering Simulations

2.1 Rewrite Theories

Membership equational logic (MEL) [9] is a typed equational logic in which data are first classified by \textit{kinds} and then further classified by \textit{sorts}, with each kind \( k \) having an associated set \( S_k \) of \textit{sorts}, so that a datum having a kind but not a sort is understood as an \textit{error} or \textit{undefined} element. Given a MEL signature \( \Sigma \), we write \( T_{\Sigma,k} \) and \( T_{\Sigma}(X)_k \) to denote respectively the set of ground \( \Sigma \)-terms of kind \( k \) and of \( \Sigma \)-terms of kind \( k \) over variables in \( X \), where \( X = \{x_1 : k_1, \ldots, x_n : k_n\} \) is a set of kinded variables. \textit{Atomic formulae} have either the form \( t = t' \) (\( \Sigma \)-equation) or \( t : s \) (\( \Sigma \)-membership) with \( t, t' \in T_{\Sigma}(X)_k \) and \( s \in S_k \); and \textit{\( \Sigma \)-sentences} are universally quantified Horn clauses on such atomic formulae. A MEL \textit{theory} is then a pair \( (\Sigma, E) \) with \( E \) a set of \( \Sigma \)-sentences. Each such theory has an initial algebra \( T_{\Sigma,E} \) whose elements are equivalence classes of ground terms modulo provable equality.

In the general version of rewrite theories over MEL theories defined in [2], a \textit{rewrite theory} is a tuple \( \mathcal{R} = (\Sigma, E, \varphi, R) \) consisting of: (i) a MEL theory \( (\Sigma, E) \); (ii) a function \( \varphi : \Sigma \rightarrow \wp(\mathbb{N}) \) assigning to each function symbol \( f : k_1 \cdots k_n \rightarrow k \) in \( \Sigma \) a set \( \varphi(f) \subseteq \{1,\ldots,n\} \) of \textit{frozen argument positions}; (iii)
a set \( R \) of (universally quantified) labeled conditional rewrite rules \( r \) having the general form

\[
(\forall X) \ r : t \rightarrow t' \text{ if } \bigwedge_{i \in I} p_i = q_i \land \bigwedge_{j \in J} w_j : s_j \land \bigwedge_{i \in L} t_i \rightarrow t'_i
\]

where, for appropriate kinds \( k \) and \( k_l \) in \( K \), \( t, t', t'_l \in \mathbb{T}_\Sigma(X)_k \) and \( t, t'_l \in \mathbb{T}_\Sigma(X)_{k_l} \) for \( l \in L \).

The function \( \varphi \) specifies which arguments of a function symbol \( f \) cannot be rewritten, which are called frozen positions. Given a rewrite theory \( \mathcal{R} = (\Sigma, E, \varphi, R) \), a sequent of \( \mathcal{R} \) is a pair of (universally quantified) terms of the same kind \( t, t' \), denoted \( (\forall X) t \rightarrow t' \) with \( X = \{ x_1 : k_1, \ldots, x_n : k_n \} \) a set of kinded variables and \( t, t' \in \mathbb{T}_\Sigma(X)_k \) for some \( k \). We say that \( \mathcal{R} \) entails the sequent \( (\forall X) t \rightarrow t' \), and write \( \mathcal{R} \vdash (\forall X) t \rightarrow t' \), if the sequent \( (\forall X) t \rightarrow t' \) can be obtained by means of the inference rules of reflexivity, transitivity, congruence, and nested replacement given in [2].

2.2 Kripke Structures and Stuttering Simulations

A transition system is a pair \( \mathcal{A} = (A, \rightarrow_{\mathcal{A}}) \) with \( A \) a set (of states) and \( \rightarrow_{\mathcal{A}} \subseteq A \times A \) the transition relation. Given a fixed set \( \Pi \) of atomic propositions, a Kripke structure is a triple \( \mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}}) \), where \( \mathcal{A} = (A, \rightarrow_{\mathcal{A}}) \) is a transition system with \( \rightarrow_{\mathcal{A}} \) a total relation, and \( L_{\mathcal{A}} : A \rightarrow \varphi(\Pi) \) is a labeling function associating to each state the set of atomic propositions that hold in it. We write \( \rightarrow^* \) for the total relation that extends a relation \( \rightarrow \) by adding a pair \( (a, a) \) for each \( a \) such that there is no \( b \) with \( a \rightarrow b \).

To a rewrite theory \( \mathcal{R} = (\Sigma, E, \varphi, R) \) we can associate a Kripke structure \( \mathcal{K}(\mathcal{R}, k)_{L_{\Pi}} = (\mathbb{T}_{\Sigma/E,k}, (\rightarrow_{\mathcal{R},k})^*, L_{\Pi}) \) in a natural way provided we: (i) specify a kind \( k \) in \( \Sigma \) so that the set of states is defined as \( \mathbb{T}_{\Sigma/E,k} \), and (ii) define a set \( \Pi \) of (possibly parametric) atomic propositions on those states; such propositions can be defined equationally in a protecting extension \( (\Sigma \cup \Pi, E \cup D) \supseteq (\Sigma, E) \), and give rise to a labeling function \( L_{\Pi} \) on the set of states \( \mathbb{T}_{\Sigma/E,k} \) in the obvious way. The transition relation of \( \mathcal{K}(\mathcal{R}, k)_{L_{\Pi}} \) is the one-step rewriting relation of \( \mathcal{R} \), to which a self-loop is added for each deadlocked state. The semantics of linear-time temporal logic (LTL) formulas is defined for Kripke structures in the well-known way (e.g., [3,5]). In particular, for any LTL formula \( \psi \) on the atomic propositions \( \Pi \) and an initial state \( [t] \), we have a satisfaction relation \( \mathcal{K}(\mathcal{R}, k)_{L_{\Pi}} \models [t] \models \psi \) which can be model checked, provided the number of states reachable from \( [t] \) is finite. Maude 2.1 [5] provides an explicit-state LTL model checker precisely for this purpose.

In [8] the notion of stuttering simulations, which is used to relate Kripke structures, is introduced. For \( \mathcal{A} = (A, \rightarrow_{\mathcal{A}}) \) and \( \mathcal{B} = (B, \rightarrow_{\mathcal{B}}) \) transition systems and \( H \subseteq A \times B \) a relation, a path \( \rho \) in \( \mathcal{B} \) \( H \)-matches a path \( \pi \) in \( \mathcal{A} \) if there are strictly increasing functions \( \alpha, \beta : \mathbb{N} \rightarrow \mathbb{N} \) with \( \alpha(0) = \beta(0) = 0 \) such that, for all \( i, j, k \in \mathbb{N} \), if \( \alpha(i) \leq j < \alpha(i+1) \) and \( \beta(i) \leq k < \beta(i+1) \), it holds that \( \pi(j) \in H \rho(k) \). A stuttering simulation of transition systems \( H : \mathcal{A} \rightarrow \mathcal{B} \)
is a binary relation $H \subseteq A \times B$ such that if $a H b$, then for each path $\pi$ in $\mathcal{A}$ there is a path $\rho$ in $\mathcal{B}$ that $H$-matches $\pi$. Given Kripke structures $\mathcal{A} = (A, \rightarrow_A, L_A)$ and $\mathcal{B} = (B, \rightarrow_B, L_B)$ over a set of atomic propositions $\Pi$, a stuttering $\Pi$-simulation $H : \mathcal{A} \rightarrow \mathcal{B}$ is a stuttering simulation of transition systems $H : (A, \rightarrow_A) \rightarrow (B, \rightarrow_B)$ such that if $a H b$ then $L_B(b) \subseteq L_A(a)$. We call $H$ a stuttering $\Pi$-bisimulation if $H$ and $H^{-1}$ are stuttering $\Pi$-simulations, and we call $H$ strict if $a H b$ implies $L_B(b) = L_A(a)$. A strict stuttering simulation $H : \mathcal{A} \rightarrow \mathcal{B}$ reflects satisfaction of $ACTL^*$ formulas without the next operator $\bigcirc$ as explained in [8], where $ACTL^*$ is the restriction of $CTL^*$ to those formulas whose negation-normal forms do not contain any existential path quantifiers [3]. In particular, $ACTL^*$ contains $LTL$ as a special case.

2.3 Real-Time Maude

The references [15,14] describe Real-Time Maude and its semantics in detail. A real-time rewrite theory $\mathcal{R}$ is a tuple $(\Sigma, E, \varphi, R, \phi, \tau)$, where $(\Sigma, E, \varphi, R)$ is a (generalized) rewrite theory, such that

- $\phi$ is an equational theory morphism $\phi : \text{TIME} \rightarrow (\Sigma, E)$ from the theory $\text{TIME}$ [11] which defines time abstractly as an ordered commutative monoid $(\text{Time}, 0, +, \leq)$ with additional operators such as $\div$ ("monus") and $\leq$;
- $(\Sigma, E)$ contains a sort $\text{System}$ (denoting the state of the system), and a specific sort $\text{GlobalSystem}$ with no subsorts and supersorts and with an operator $\{,\} : \text{System} \rightarrow \text{GlobalSystem}$ so that each ground term $t$ of sort $\text{GlobalSystem}$ reduces to a term of the form $\{u\}$ using the equations $E$; furthermore, the sort $\text{GlobalSystem}$ does not appear in the arity of any function symbol in $\Sigma$;
- $\tau$ is an assignment of a term $\tau_l$ of sort $\phi(\text{Time})$ to every rewrite rule $l : \{t\} \rightarrow \{t'\}$ if $\text{cond}$ involving terms of sort $\text{GlobalSystem}$\footnote{All rules involving terms of sort $\text{GlobalSystem}$ are assumed to have different labels.}; if $\tau_l \neq \phi(0)$ we call the rule a tick rule and write $r : \{t\} \xrightarrow{\tau_l, \tau} \{t'\}$ if $\text{cond}$. Rewrite rules that are not tick rules are called instantaneous rules and are supposed to take zero time.

The state of the system should have the form $\{u\}$, in which case the form of the tick rules ensures that time advances uniformly in all parts of the system. The total time elapse $\tau(\alpha)$ of a rewrite $\alpha : \{t\} \rightarrow \{t'\}$ of sort $\text{GlobalSystem}$ is the sum of the times elapsed in each tick rule application [11]. We write $\mathcal{R} \vdash \{t\} \xrightarrow{r} \{t'\}$ if there is proof $\alpha : \{t\} \rightarrow \{t'\}$ in $\mathcal{R}$ with $\tau(\alpha) = r$. We write $\text{Time}$, 0, $\ldots$, for $\phi(\text{Time})$, $\phi(0)$, etc.

Real-Time Maude extends Full Maude [5] to support the specification of real-time rewrite theories as timed modules and object-oriented timed modules. Although Real-Time Maude is parametric in the time domain, which may be discrete or dense, it provides a "skeleton" sort $\text{Time}$ and a supersort $\text{TimeInf}$
which adds an infinity element \( \text{INF} \). Tick rules should (in particular for dense time) have one of the forms

\[
\text{cr}_l \{l\} : \{t\} \rightarrow \{t'\} \text{ in time } x \text{ if } \text{cond} \land x \leq u \land \text{cond}' \text{ [nonexec]} .
\]

(†),

\[
\text{cr}_l \{l\} : \{t\} \rightarrow \{t'\} \text{ in time } x \text{ if } \text{cond} \text{ [nonexec]} .
\]

(‡), or

\[
\text{rl}_l \{l\} : \{t\} \rightarrow \{t'\} \text{ in time } x \text{ [nonexec]} .
\]

(§),

where \( x \) is a variable of sort \text{Time} which does not occur in \( \{t\} \) and which is not initialized in the condition. The term \( u \) denotes the maximum amount by which time can advance in one tick step. The (possibly empty) conditions \text{cond} and \text{cond}' should not further constrain \( x \) (except possibly by adding the condition \( x \neq 0 \)). Rules of these forms are in general not executable since it is not clear what value to assign to \( x \) in a rewrite step; our tool deals with such rules by offering a choice of different “time sampling” strategies for setting the value of \( x \). For example, the \textit{maximal} time sampling strategy advances time by the maximum possible time elapse \( u \) in rules of the form (†) (unless \( u \) equals \text{INF}), and tries to advance time by a user-given time value \( r \) in tick rules having other forms. All applications of time-nondeterministic tick rules—be it for rewriting, search, or model checking—are performed using the selected time sampling strategy. This means that some behaviors in the system, namely those obtained by applying the tick rules differently, are not analyzed.

The real-time rewrite theory \( \mathcal{R}^{\text{maxDef}(r),\text{nz}} \) denotes the real-time rewrite theory \( \mathcal{R} \) where the tick rules are applied according to the maximal time sampling strategy, and where tick steps which advance time by 0 are not applied. The “untimed” rewrite theory \( \mathcal{R}^C \) is obtained from \( \mathcal{R} \) by adding a “clock” component to the state, so that the global state is a term of the form \( \text{\{t\} in time r} \) of sort \text{ClockedSystem}, where \( r \) is the duration of the rewrite leading to state \( \{t\} \). When the maximal time sampling strategy is chosen, Real-Time Maude executes a command by internally transforming the real-time rewrite theory \( \mathcal{R} \) and the command to the theory \( \langle \mathcal{R}^{\text{maxDef}(r),\text{nz}} \rangle^C \) (or to an extension of this, depending on the command to be executed) and executes the corresponding command in Maude [15].

In this paper, we focus on Real-Time Maude’s \textit{unbounded} and \textit{time-bounded} search and \textit{LTL} model checking commands. In unbounded model checking under the maximal time sampling strategy, we check the \textit{LTL} formula w.r.t. each path in \( \langle \mathcal{R}^{\text{maxDef}(r),\text{nz}} \rangle^C \) starting with the state \( t_0 \text{ in time } 0 \). For \textit{time-bounded} model checking with time bound \( \Delta \), we only consider the set of paths \( \text{Paths}(\mathcal{R}^{\text{maxDef}(r),\text{nz}})_{t_0}^{\leq \Delta} \) of \( \mathcal{R}^C \)-states “chopped off” at the time limit \( \Delta \) as explained in [15], where we also define the unbounded and time-bounded satisfaction \( \langle \mathcal{R}, L, t_0 \models \Phi \rangle \) and \( \langle \mathcal{R}, L, t_0 \models \leq \Delta \Phi \rangle \), respectively, in the expected way.
3 Soundness and Completeness of the Maximal Time Sampling Strategy in Time-Robust Systems

In this section, we define time-robust real-time rewrite theories and show that unbounded and time-bounded search and model checking analyses using the maximal time sampling strategy are sound and complete with respect to the timed fair paths of a time-robust theory, given that the atomic propositions satisfy certain “stability” requirements with respect to tick steps.

A time-robust system is one where:

(i) From any given state time can advance either by (i) any amount, by (ii) any amount up to and including a specific instant in time, or (iii) not at all.

(ii) Advancing time does not affect the above property, unless time is advanced all the way to the specific bound in time in case 1-(ii) above.

(iii) An instantaneous rewrite rule can only be applied at specific times, namely, when the system has advanced time by the maximal possible amount.

A typical example of such time-robust systems is one where each instantaneous transition is triggered by the expiration of a timer or by the arrival of a message with a given transmission delay. Our experience indicates that many large systems are indeed time-robust.

A time-robust system may have Zeno paths, where the sum of the durations of an infinite number of tick steps is bounded. We differentiate between Zeno paths forced on the system by the specification (this could indicate a flaw in the system design) and Zeno paths that are due to bad “choices” in the tick increments. Intuitively, the second type of Zeno behavior does not reflect realistic behaviors in the system and therefore is not simulated by the maximal time sampling strategy. We therefore call timed fair paths those paths of the system that do not exhibit this second, unrealistic kind of Zeno behavior.

In Section 3.4 we state that there is a stuttering bisimulation, as defined in [8] \(^2\), between the set of timed fair paths in a time-robust real-time rewrite theory \(^3\) \(\mathcal{R}\) and the theory \(\mathcal{R}_{\text{maxDef}(r),\text{nz}}\), which defines the system where the tick rules are applied according to the maximal time sampling strategy. The full proof of this theorem, together with other proofs, is given in [13]. This main result is proved by proving that for each timed fair path \(\pi\) in (the Kripke structure associated with) \(\mathcal{R}\), there is a corresponding path \(\rho\) in (the Kripke structure associated with) \(\mathcal{R}_{\text{maxDef}(r),\text{nz}}\) which matches \(\pi\) as explained in [8], and vice versa. Such a stuttering bisimulation means that each infinite path can be appropriately simulated to analyze unbounded properties. Section 3.5

\(^2\) Although with the slight difference that we work on sets of paths to treat timed fair paths.

\(^3\) In addition, the set of propositions considered must satisfy some properties with respect to tick steps.
gives some requirements which are sufficient to prove that each time-bounded prefix of a timed fair path can be simulated by a time-bounded path obtained by using the maximal time sampling strategy, so that we get completeness also for the analysis of time-bounded formulas.

3.1 Time-Robust Real-Time Rewrite Theories

There are two different kinds of tick rule applications that the maximal strategy can treat:

- ticks from states from which time can only advance up to a certain maximal time, and
- ticks from states from which time can advance by any amount.

The maximal time sampling strategy handles the first kind of tick steps by advancing time as much as possible, and handles the second kind by advancing time by a user-given time value $r$.

**Definition 3.1** A one-step rewrite $t \xrightarrow{r} t'$ using a tick rule and having duration $r$ is:

- a maximal tick step, written $t \xrightarrow{r_{\text{max}}} t'$, if there is no time value $r' > r$ such that $t \xrightarrow{r'} t''$ for some $t''$;
- an $\infty$ tick step, written $t \xrightarrow{r_{\infty}} t'$, if for each time value $r' > 0$, there is a tick rewrite step $t \xrightarrow{r'} t''$; and
- a non-maximal tick step if there is a maximal tick step $t \xrightarrow{r_{\text{max}}} t''$ for $r' > r$.

**Example 3.2** The following specification models a dense-time retrograde clock, where the term $\{\text{clock}(24)\}$ should always be explicitly reset to $\{\text{clock}(0)\}$. Just before or after this reset operation, the clock may check whether it has sufficient battery capacity left or whether it must stop immediately by going to a state $\{\text{clock-dead}(\ldots)\}$.

(tmod SIMPLIFIED-DENSE-CLOCK is protecting POSRAT-TIME-DOMAIN .


vars R R' : Time .

crl [tickWhenRunning] : $\{\text{clock}(R)\} \Rightarrow \{\text{clock}(R + R')\}$ in time $R'$
    if $R' \leq 24$ monus $R$ [nonexec] .

rl [tickWhenStopped] : $\{\text{stopped-clock}(R)\} \Rightarrow \{\text{stopped-clock}(R)\}$
    in time $R'$ [nonexec] .

rl [reset] : $\text{clock}(24) \Rightarrow \text{clock}(0)$ .

rl [batteryDies1] : $\text{clock}(0) \Rightarrow \text{stopped-clock}(0)$ .

rl [batteryDies2] : $\text{clock}(24) \Rightarrow \text{stopped-clock}(24)$ .

endtm)

From a state $\{\text{clock}(5)\}$ there is an infinity of non-maximal tick steps, e.g., to $\{\text{clock}(10 + 2/7)\}$, as well as a maximal tick step to $\{\text{clock}(24)\}$. From $\{\text{stopped-clock}(24)\}$ there are $\infty$ tick steps from the term to itself in any
amount of time.

The first requirement for a theory $R$ to be time-robust is that each tick step is either a maximal, a non-maximal, or an $\infty$ tick step. This excludes specifications with dense time domains where time can advance by any amount strictly smaller than some value $\Delta$ from some state, but where time cannot advance by time $\Delta$ or more from the same state. Specifications with tick rules of the form $\dagger$, $\ast$, and $\S$ where the conditions $\text{cond}$ and $\text{cond'}$ do not further constrain $x$ will satisfy this requirement.

The next set of assumptions concerns the ability to cover all behaviors without performing non-maximal ticks. We have to make sure that a sequence of non-maximal ticks followed by a maximal tick can be simulated by just one maximal tick step, and that performing a non-maximal tick cannot lead to behaviors (deadlocks, other tick possibilities, taking instantaneous rules, etc.) different from those that can be covered with a maximal tick. Therefore, we add the following requirements: If $t \xrightarrow{r+\epsilon} t''$ is a maximal tick step and $t \xrightarrow{r} t'$ is a non-maximal tick step, then there should be a maximal tick step $t' \xrightarrow{r'} t'''$ for some $t'''$. Likewise, if $t \xrightarrow{r} t'$ is a non-maximal tick step and $t' \xrightarrow{r'} t''$ is a maximal tick step, then there must be a maximal tick step $t \xrightarrow{r'} t''$. Finally, to ensure that no instantaneous rule is ignored by ticking maximally, if $t \xrightarrow{r} t'$ is a non-maximal tick step with $r > 0$, then there is no instantaneous one-step rewrite $t' \xrightarrow{\text{inst}} t''$.

We next consider $\infty$ tick steps. For the system to be time-robust, performing an $\infty$ tick step should not lead to the possibility of applying an instantaneous rule, and should not preclude the possibility of taking further $\infty$ steps. Therefore, if $t \xrightarrow{r} t'$ is an $\infty$ tick step with $r > 0$, then there is also an $\infty$ tick step from $t'$, and there can be no instantaneous step $t' \xrightarrow{\text{inst}} t''$. These criteria by themselves only ensure that, once we have performed an $\infty$ tick step, all the remaining steps will be $\infty$ tick steps, and that we can have a sequence of $\infty$ tick steps where time is advanced by $r$ in each step. For our desired bisimulation result, the key idea is that each infinite sequence of $\infty$ tick steps is simulated by another such sequence. However, there need not be any relationship between the states and the durations of these two sequences of $\infty$ steps. To have a proper stuttering simulation, we must also take the propositions into account.

Real-Time Maude assumes that zero-time tick steps do not change the state of the system; therefore the tool does not apply a tick rule to advance time by 0. Consequently, we require that if $t \xrightarrow{0} t'$ is a tick step that advances time by 0 time units for ground terms $t$ and $t'$, then $t$ and $t'$ must be equivalent in the underlying equational theory of the system.

We summarize the requirements for time-robustness as follows:

**Definition 3.3** A real-time rewrite theory $R$ is time-robust if the following requirements TR1 to TR6 hold for all ground terms $t$, $t'$, $t''$ of sort `GlobalSystem`,
and ground terms $r, r', r''$ of sort $\text{Time}$:

\begin{enumerate}
  \item Each one-step rewrite using a tick rule is either a maximal, a non-
  maximal, or an $\infty$ tick step.
  \item $t \xrightarrow{r \text{ max}} t'$ and a non-maximal tick step $t \xrightarrow{r \text{ max}} t'$ imply that there is a maximal tick step $t' \xrightarrow{r' \text{ max}} t''$ for some $t''$.
  \item For $t \xrightarrow{r} t'$ a non-maximal tick step, $t' \xrightarrow{r' \text{ max}} t''$ implies that there is a maximal step $t \xrightarrow{r + r' \text{ max}} t''$.
  \item If $t \xrightarrow{r} t'$ is a tick step with $r > 0$, and $t' \xrightarrow{\text{inst}} t''$ is an instantaneous one-step rewrite, then $t \xrightarrow{r} t'$ is a maximal tick step.
  \item $t \xrightarrow{r \text{ max}} t'$ implies that there are $t', r'$ such that $t' \xrightarrow{r' \text{ max}} t''$.
  \item $t = t'$ holds in the underlying equational theory for any 0-time tick step $t \xrightarrow{0} t'$.
\end{enumerate}

3.2 Zeno Behaviors and Timed Fairness

For dense time, the form of typical tick rules makes it possible to have “Zeno” behaviors in the system in which an infinite number of tick applications only advances the total time in the system by a finite amount. For analysis purposes it seems important to differentiate between Zeno behaviors caused by “bad choices” about how much to increase time, and Zeno behaviors forced upon the system by the specification (which indicates a design error in the model). In the latter category we also include the case where there is an infinite sequence of applications of instantaneous rules. For example, the rewrite sequence

$$
t_0 \xrightarrow{1} t_1 \xrightarrow{1/2} t_2 \xrightarrow{1/4} t_3 \xrightarrow{1/8} \cdots
$$

is a Zeno behavior caused by bad choices of advancing time in a system which has a tick rule like

$$\text{rl [tick]} : \{t\} \Rightarrow \{g(t, x)\} \text{ in time } x \text{ .}
$$

However, a Zeno rewrite sequence

$$\{f(1)\} \xrightarrow{1} \{f(2)\} \xrightarrow{1/2} \{f(4)\} \xrightarrow{1/4} \{f(8)\} \xrightarrow{1/8} \cdots
$$

is “forced” by the tick rule

$$\text{crl [tick]} : \{f(N)\} \Rightarrow \{f(2 \times N)\} \text{ in time } x \text{ if } x \leq 1/N \text{ .}
$$

We will ignore, as timed unfair, all paths with an infinite sequence of tick steps where, at each step, time could have advanced to time $r_0$ or beyond, but where the total duration of a path never reaches time $r_0$.

Another timed unfairness issue deals with the fact that a system could continuously advance time by 0 in a tick rule. If this is the maximum amount by which time can advance from a state, then such bad sequences are not
covered by the above non-Zeno requirement. In our clock example, a system could continuously apply the tick rule to the state \{clock(24)\} to reach the same state in the maximal possible time 0. Given that 0-time ticking should not change the state of the system, we require that a “fair” path does not contain an infinite sequence consisting only of 0-time ticks as long as an instantaneous rule can be applied.

We define \textit{timedFairPaths}(\mathcal{R}) to be the set of paths of a theory \(\mathcal{R}\) that satisfy the above two requirements, and define unbounded satisfaction of LTL formulas with respect to such timed fair paths as follows:

**Definition 3.4** Given a real-time rewrite theory \(\mathcal{R}\), a term \(t_0\) of sort \textit{GlobalSystem}, and a ground term \(r\) of sort \textit{Time}, the set \textit{Paths}(\mathcal{R}),\:t_0 is the set of all infinite sequences

\[
π = (t_0 \text{ in time } r_0 \rightarrow t_1 \text{ in time } r_1 \rightarrow \cdots \rightarrow t_i \text{ in time } r_i \rightarrow \cdots)
\]

of \(\mathcal{R}^C\)-states, with \(r_0 = 0\), such that either

- for each \(i\), \(t_i \rightarrow t_{i+1}\) is a one-step rewrite having duration \(r\) (which is 0 when an instantaneous rule is applied) with \(r_i + r = r_{i+1}\); or

- there is a \(k\) such that there is no one-step rewrite from \(t_k\) in \(\mathcal{R}\), and such that \(t_k = t_j\) and \(r_k = r_j\) for each \(j > k\), and for all \(i < k\), \(t_i \rightarrow t_{i+1}\) is a one-step rewrite having duration \(r\) with \(r_i + r = r_{i+1}\).

The set \textit{timedFairPaths}(\mathcal{R}),\:t_0 is the subset of the paths \(π\) in \textit{Paths}(\mathcal{R}),\:t_0 that satisfy the following conditions:

- for any ground term \(Δ\) of sort \textit{Time}, if there is a \(k\) such that for each \(j > k\) there is a one-step tick rewrite \(t_j \xrightarrow{\text{tick}} t'\) with \(Δ \leq r_j + r\), then there must be an \(l\) with \(Δ \leq r_l\);

- 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 it must be the case that \(t_l \xrightarrow{\text{inst}} t_{l+1}\) is a one-step rewrite applying an instantaneous rule for some \(l > k\).

We extend the satisfaction relation in [15] to timed fair paths as follows:

**Definition 3.5** Given a real-time rewrite theory \(\mathcal{R}\), a protecting extension \(L_Π\) of \(\mathcal{R}^C\) defining the atomic state and clocked propositions \(Π\), a term \(t_0\) of sort \textit{GlobalSystem}, and an LTL formula \(Φ\), we define satisfaction, without time bound, with respect to timed fair paths as follows:

\[
\mathcal{R}, L_Π, t_0 \models_\text{tf} \Phi \iff \pi, L_Π^C \models Φ \text{ for all paths } π ∈ \text{timedFairPaths}(\mathcal{R}),\:t_0.
\]

### 3.3 Temporal Logic Propositions and Tick Steps

For model checking purposes, atomic propositions should satisfy certain “stability” requirements with respect to tick steps to enable the maximal time
sampling strategy to simulate all timed fair paths. For *unbounded* model checking, we may allow the valuation of a set of temporal logic properties to change *once* in a sequence of tick applications. As explained in Section 3.5, for *time-bounded* model checking, properties should not change by tick steps that are not maximal tick steps. The following example shows that it is not necessary to require that a proposition is unchanged by a tick step for unbounded analysis:

**Example 3.6 (Example 3.2 cont.)** In our clock example, we could define a proposition \( \text{ge20} \) which holds if the clock shows a value greater or equal than 20. This proposition is not invariant under ticks. Nevertheless, such a proposition should be allowed under the maximal time sampling strategy with unbounded analysis, since it changes only *once* in any tick sequence from \( \{ \text{clock}(0) \} \) to \( \{ \text{clock}(24) \} \). The term \( \{ \text{clock}(0) \} \) can therefore “simulate” all the stuttering steps from \( \{ \text{clock}(0) \} \) to the last state in the tick sequence with a clock value less than 20, and the term \( \{ \text{clock}(24) \} \) could simulate the remaining steps. However, if we add another proposition \( \text{ge22} \), the valuation of the two propositions could change *twice* in a tick sequence, and neither \( \{ \text{clock}(0) \} \) nor \( \{ \text{clock}(24) \} \) can represent, e.g., the state \( \{ \text{clock}(21) \} \). The maximal time sampling strategy would never find a behavior containing a state satisfying \( \text{ge20} \land \neg \text{ge22} \). Likewise, the proposition \( \text{clockIs20} \) (which holds only for the state \( \{ \text{clock}(20) \} \)) can neither be “simulated” by \( \{ \text{clock}(0) \} \) nor by \( \{ \text{clock}(24) \} \).

For unbounded model checking we need to assume that the set of propositions under consideration is *tick-stabilizing*, in the sense that if

\[
t_1 \xrightarrow{r_1} t_2 \xrightarrow{r_2} \cdots \xrightarrow{r_{n-1}} t_n \xrightarrow{r_{n-1} \max} t_n
\]

is a sequence of non-maximal tick steps followed by a maximal tick step, then there is an \( i < n \) such that \( t_1, \ldots, t_i \) can all be simulated by \( t_1 \), and \( t_{i+1}, \ldots, t_n \) can simulated by \( t_n \):

**Definition 3.7** Let \( P \subseteq \text{AP} \) be a set of atomic propositions (or a property corresponding to a search pattern). For ground terms \( t, t' \), we write \( t \simeq_P t' \) (or just \( t \simeq t' \) when \( P \) is implicit) if \( t \) and \( t' \) satisfy exactly the same set of propositions from \( P \); that is, \( L(t) \cap P = L(t') \cap P \) for the labeling function \( L \). Such a set of propositions \( P \) is *tick-stabilizing* if and only if:

- For each sequence \( t_1 \xrightarrow{r_1} t_2 \xrightarrow{r_2} \cdots \xrightarrow{r_{n-1}} t_{n-1} \xrightarrow{r_{n-1} \max} t_n \) of non-maximal tick steps followed by a maximal tick step, there is a \( k < n \) such that \( t_1 \simeq_P t_j \) for each \( j \leq k \), and such that \( t_l \simeq_P t_n \) for each \( l > k \).

- \( t \xrightarrow{r} t' \) and \( t' \xrightarrow{r'} t'' \) implies \( t' \simeq_P t'' \) when \( r > 0 \). In addition, \( t \xrightarrow{r \infty} t' \) and \( t \xrightarrow{r' \infty} t'' \) implies \( t' \simeq_P t'' \) for \( r, r' > 0 \).

For *clocked* propositions\(^4\), the tick steps \( t \xrightarrow{r} t' \) should be understood as

\(^4\) Whether or not a state satisfies a *clocked* proposition depends not only on the state, but
their equivalent clocked rewrites \( t \rightarrow \text{in time } r' \rightarrow t' \) in time \( r' + r \) for each \( r' \), so that by “\( t \rightarrow \text{in time } r' \rightarrow t' \)” with \( p \) a clocked proposition we mean that, for each time value \( r' \), the proposition \( p \) must hold for the clocked state \( t \) in time \( r' \) if and only it holds for the clocked state \( t' \) in time \( r' + r \).

### 3.4 Completeness of Unbounded Model Checking

In this section we prove that \( R, L, t_0 \models \Phi \) holds if and only if it is the case that \( R^{\maxDef(r),nz}, L, t_0 \models \Phi \) holds, for \( R \) a time-robust real-time rewrite theory, \( \Phi \) an \( LTL \backslash \{\Box\} \) formula, and \( L \) a labeling function extending \( R^C \) such that the set of atomic propositions occurring in \( \Phi \) satisfies the tick-stabilization requirement. This equivalence is proved by showing that \( \simeq_P \) is a strict stuttering bisimulation between the Kripke structure \( K(R^C, [\text{ClockedSystem}])_{L^C} \), restricted to its timed fair paths, and \( K((R^{\maxDef(r),nz})^C, [\text{ClockedSystem}])_{L^C} \). Furthermore, the bisimulation is time-preserving as explained in the proof, which is given in [13].

**Lemma 3.8** Let \( R \) be a time-robust real-time rewrite theory, \( r \) a time value in \( R \) greater than 0, \( L \) a protecting extension of \( R^C \) which defines the propositions \( \Pi \), and \( P \subseteq \Pi \) a set of tick-stabilizing atomic propositions (some of which could be clocked, and some unclocked). Then,

\[
\simeq_P : K(R^C, [\text{ClockedSystem}])_{L^C} \rightarrow K((R^{\maxDef(r),nz})^C, [\text{ClockedSystem}])_{L^C}
\]

is a strict stuttering \( P \)-simulation when \( K(R^C, [\text{ClockedSystem}])_{L^C} \) is restricted to timed fair paths.

The converse of the above lemma also holds (the proof is given in [13]):

**Lemma 3.9** The relation \( \simeq_P \) is a strict stuttering \( P \)-simulation

\[
\simeq_P : K((R^{\maxDef(r),nz})^C, [\text{ClockedSystem}])_{L^C} \rightarrow K(R^C, [\text{ClockedSystem}])_{L^C}
\]

when \( K(R^C, [\text{ClockedSystem}])_{L^C} \) is restricted to timed fair paths.

**Theorem 3.10** Let \( R \) be a time-robust real-time rewrite theory, \( r \) a time value in \( R \) greater than 0, \( L \) a protecting extension of \( R^C \) which defines the propositions \( \Pi \), and \( P \subseteq \Pi \) a set of tick-stabilizing atomic propositions (some of which could be clocked, and some unclocked). Then \( \simeq_P \) is a strict stuttering \( P \)-bisimulation between \( K(R^C, [\text{ClockedSystem}])_{L^C} \), restricted to its timed fair paths, and \( K((R^{\maxDef(r),nz})^C, [\text{ClockedSystem}])_{L^C} \).

**Proof.** Follows directly from Lemmas 3.8 and 3.9, since \( \simeq_P \) equals \( (\simeq_P)^{-1} \) because \( \simeq_P \) is symmetric.

\( \square \)

---

\( \text{also on the total time it has taken to reach the state.} \)

141
The following main result, whose proof is also presented in [13], shows that maximal time sampling analysis is sound and complete:

**Corollary 3.11** Let \( \mathcal{R}, t_0, r, L_\Pi, \) and \( P \) be as in Theorem 3.10, and let \( \Phi \) be an LTL \( \setminus \{\circ\} \) formula whose atomic propositions are contained in \( P \). Then

\[
\mathcal{R}, L_\Pi, t_0 \models \mathcal{U} \Phi \quad \text{if and only if} \quad \mathcal{R}^{\text{maxDef}(r),nz}, L_\Pi, t_0 \models \Phi.
\]

### 3.5 Soundness and Completeness of Time-Bounded Model Checking using the Maximal Time Sampling Strategy

Real-Time Maude's *time-bounded* model checking features have proved very useful for analyzing large applications [17,16]. Not only are time-bounded properties interesting *per se*, but model checking analyses terminate for time-bounded properties in non-Zeno specifications when using one of Real-Time Maude’s time sampling strategies. Furthermore, even when the reachable state space is finite, using the maximal time sampling strategy can drastically reduce the state space to make time-bounded model checking feasible.

We defined in [15] the semantics of time-bounded properties. Essentially, a time-bounded formula is interpreted over all possible paths “chopped off” at the time limit, and with self-loops added to states from which time could advance beyond the time limit in one tick step. In this semantics, the time-bounded property “\( \langle \langle \mathcal{R} \rangle_{t_0} \geq 20 \) in time \( \leq 22 \)” would *not* hold from \( \{\text{clock}(0)\} \) in our clock example, since there is, e.g., a tick step from the clocked state \( \{\text{clock}(0)\} \) in time \( 0 \) to the state \( \{\text{clock}(24)\} \) in time \( 24 \) in one step.

However, the property *does* hold with time bound \( 24 \) according to our semantics.

We denote by \( \text{timedFairPaths}(\mathcal{R})_{t_0}^{\leq \Delta} \) the subset of \( \text{Paths}(\mathcal{R})_{t_0}^{\leq \Delta} \) (see [15]) which contains all the timed fair paths starting in state \( t_0 \) that are chopped off at time \( \Delta \) as explained in [15]. The satisfaction relation \( \models \mathcal{U} \) over timed fair paths is extended to time-bounded satisfaction in the obvious way:

**Definition 3.12** \( \mathcal{R}, L_\Pi, t_0 \models \mathcal{U}_{\leq \Delta} \Phi \) holds if and only if \( \pi, L_\Pi \models \Phi \) holds for all \( \pi \in \text{timedFairPaths}(\mathcal{R})_{t_0}^{\leq \Delta} \).

Since when we have time bounds a sequence of non-maximal tick steps followed by a maximal tick step can be chopped off before the maximal tick is taken, it is no longer sufficient to require tick stabilization of the atomic propositions. Instead, we now must require *tick-invariance*, which means that only a *maximal* tick step may change the valuation of the propositions:

**Definition 3.13** A time-robust specification is tick-invariant with respect to a set \( P \) of propositions if and only if it is the case that \( t \approx_P t' \) holds for each non-maximal or \( \infty \) tick step \( t \xrightarrow{r} t' \).

**Fact 3.14** Tick-invariance implies tick-stabilization.

Our main result, proved in [13], is that the maximal time sampling strategy...
is sound and complete for time-robust systems when the atomic propositions
are tick-invariant:

**Theorem 3.15** Given a time-robust real-time rewrite theory $R$, a protecting
extension $L_{\Pi}$ of $R^C$ defining the atomic state and clocked propositions $\Pi$, a
tick-invariant subset $P \subseteq \Pi$, an initial state $t_0$ of sort $\text{GlobalSystem}$, a Time
value $\Delta$, and an LTL $\{\Box\}$ formula $\Phi$ whose atomic propositions are contained
in $P$. Then,

$$R, L_P, t_0 \models^{tf}_{\leq \Delta} \Phi \quad \text{if and only if} \quad R^{\text{maxDef}(r).nz}, L_P, t_0 \models_{\leq \Delta} \Phi.$$

The completeness results in Corollary 3.11 and Theorem 3.15 carry over
directly to unbounded and time-bounded search without lower timer bounds.
Therefore, unbounded and time-bounded search using the maximal time sam-
pling strategy become, respectively, a complete semi-decision procedure and a
complete decision procedure for the reachability problem for non-Zeno speci-
fications.

In [13] we prove that our clock specification is time-robust and that the
unbounded and time-bounded search and model checking analyses in [15] are
complete when using maximal time sampling.

## 4 Completeness for Object-Based Systems

In Real-Time Maude we usually specify “flat” object-based real-time systems
by means of functions

$$\text{op } \delta : \text{Configuration Time} \rightarrow \text{Configuration [frozen (1)] \ .}$$

$$\text{op } \text{mte} : \text{Configuration} \rightarrow \text{TimeInf [frozen (1)] \ .}$$

that define, respectively, the effect of time elapse on a configuration, and
the maximum time elapse (mte) possible from a configuration. We let these
functions distribute over the objects and messages in a configuration according
generic equations of the form:

$$\text{vars NeC NeC' : NEConfiguration . var R : Time .}$$

$$\text{eq } \delta(\text{none, R}) = \text{none .}$$

$$\text{eq } \delta(\text{NeC NeC'}, R) = \delta(\text{NeC, R}) \delta(\text{NeC'}, R) .$$

$$\text{eq } \text{mte}(\text{none}) = \text{INF .}$$

$$\text{eq } \text{mte}(\text{NeC NeC'}) = \min(\text{mte(NeC), mte(NeC')} ) .$$

together with domain-specific equations defining the functions $\delta$ and $\text{mte}$ on
individual objects and messages. There is usually only one tick rule in such
systems. That tick rule has the form

$$\text{crl \ [tick] : \{C:Configuration\}}$$

$$\Rightarrow$$

$$\{\delta(C:\text{Configuration, R:Time}) \text{ in time R:Time}$$

$$\text{if } R:\text{Time} \leq \text{mte(C:Configuration) [nonexec]} .$$
This specification technique has been used in most of the larger Real-Time Maude applications, such as in the AER/NCA protocol suite\textsuperscript{5} \cite{16}, the OGDC algorithm \cite{17}, the round trip protocols in \cite{15}, etc. Furthermore, in these examples, no instantaneous rule is enabled when a tick rule can advance time by a non-zero amount of time. In such systems, it is fairly simple to prove time-robustness:

**Theorem 4.1** Let \( \mathcal{R} \) be a flat object-oriented specification with a tick rule as defined above, let the infinity element \( \text{INF} \) be the only element in \( \text{TimeInf} \) which is not a normal time value, and let the time domain be linear (see \cite{11}). Then, \( \mathcal{R} \) is time-robust if the following conditions are satisfied for all appropriate ground terms \( t \) and \( r, r' \):

\begin{align*}
\text{OO1.} & \quad \text{mte}(\delta(t, r)) = \text{mte}(t) - r, \text{ for all } r \leq \text{mte}(t). \\
\text{OO2.} & \quad \delta(t, 0) = t. \\
\text{OO3.} & \quad \delta(\delta(t, r), r') = \delta(t, r + r'), \text{ for } r + r' \leq \text{mte}(t). \\
\text{OO4.} & \quad \text{mte}(\sigma(l)) = 0 \text{ for each ground instance } \sigma(l) \text{ of a left-hand side of an instantaneous rewrite rule.}
\end{align*}

Furthermore, it is sufficient to consider \( \text{OO1, \text{OO2, and \text{OO3}} for } t \text{ consisting of a single object or message.} \)

The proof of Theorem 4.1 is given in \cite{13}. This theorem simplifies the proof obligations for time-robustness for typical object-based systems to very simple requirements, which should be easy to check mechanically using a theorem prover such as Maude’s ITP \cite{4}. Furthermore, proving tick-invariance in such systems amounts to proving \( \{t\} \simeq_p \{\delta(t, r)\} \) for all \( t, r \) with \( r < \text{mte}(t) \).

As mentioned above, it is our experience that a large class of real-time specifications satisfy the above criteria. One such useful class is the class of systems where the precise transmission time of each message can be computed when the message is sent, and where each instantaneous rule is triggered either by the arrival of a message or by the expiration of a “timer.” In what follows we illustrate the general applicability of our results with some practical examples.

### 4.1 A Small Network Protocol Example

In \cite{15} we presented some versions of a very simple protocol for computing the *round trip time* of message transmission between pairs of nodes in a network. In one of those versions, it takes each message exactly \( \text{MIN-TRANS-TIME} \) to travel from its source to its destination. The system is specified according to the specification techniques defined above (with time specified by the built-in module \textsc{NAT-TIME-DOMAIN-WITH-INF}, with \textsc{delta} for \( \delta \), etc.) and is given in Appendix A. We prove below that the specification satisfies the simplified requirements for time-robustness:

\textsuperscript{5} Tick rules only applied to \texttt{ObjectConfigurations} in AER/NCA. This is equivalent to the above setting when the \texttt{mte} of a message is 0.
We must prove $\text{mte}(\text{delta}(C, r)) = \text{mte}(C) \monus r$, for $C$ being either a (delayed) message or an object. Using the equations in Appendix A we have:

- $\text{mte}(\text{delta}(\text{dly}(m, r'), r)) = \text{mte}(\text{dly}(m, r' \monus r)) = r' \monus r = \text{mte}(\text{dly}(m, r')) \monus r$.
- $\text{mte}(\text{delta}(< O : \text{Node} | \text{clock} : r', \text{timer} : TI >, r)) = \text{mte}(< O : \text{Node} | \text{clock} : r' + r, \text{timer} : TI \monus r >) = \text{TI} \monus r$.

$\text{mte}(< O : \text{Node} | \text{clock} : r', \text{timer} : TI >) = r'$.

$\text{OO2}$ follows trivially from the fact that $r + 0$ equals $r$ for all $r$, and that $\text{TI} \monus 0$ equals $\text{TI}$ for all $\text{TI}$ of sort $\text{TimeInf}$. $\text{OO3}$ holds since $+$ is associative and $(t \monus r) \monus r'$ equals $t \monus (r + r')$. Finally, we show that $\text{mte}$ of the left-hand side of any instance of an instantaneous rule is 0. For example, for the rule $\text{rttResponse}$, we have

$$\text{mte}(\text{rttReq}(O, O', R) < O : \text{Node} | >) = \min(\text{mte}(\text{rttReq}(O, O', R)), \text{mte}(< O : \text{Node} | >)) = \min(\text{mte}(\text{dly}(\text{rttReq}(O, O', R), 0)), \text{mte}(< O : \text{Node} | >)) = \min(0, ...) = 0 .$$

The same happens with each rule whose left-hand side contains a message. The only remaining instantaneous rule is $\text{tryagain}$, whose left-hand side has $\text{timer}$ value 0, and therefore also has $\text{mte}$ 0.

Finally, we prove tick-invariance of the propositions and search patterns in the analysis commands in Appendix A. The search patterns do not mention any attribute which is changed by $\text{delta}$, so they are tick-invariant. The proposition $\text{superfluousMsg}$ is defined as follows:

$$\text{(tomod MC-RTT is including TIMED-MODEL-CHECKER . protecting RTT-I .}
\text{op superfluousMsg : \rightarrow Prop [ctor] .
\text{vars REST : Configuration . vars O O' : Oid .
\text{vars R R' R'' : Time .
\text{ceq {REST < O : \text{Node} | rtt : R, clock : R' > rttReq(O', O, R'')} =
\text{superfluousMsg = true if R'' + MAX-DELAY > R'.
\text{ceq {REST < O : \text{Node} | rtt : R, clock : R' > rttResp(O, O', R'')} =
\text{superfluousMsg = true if R'' + MAX-DELAY > R'.
\text{endtom})$$

Satisfaction of this proposition depends on the value of the $\text{clock}$ attribute of the $\text{Node}$ object and could therefore potentially be vulnerable to change by a tick. However, if $\text{superfluousMsg}$ holds in $t$, then $t$ contains a message and hence has $\text{mte}$ 0, and therefore tick-invariance is vacuously true. If $t$ does not contain a $\text{rttReq}$ or $\text{rttResp}$ message, ticking will not create such a message, so that $\text{superfluousMsg}$ holds neither before nor after the tick step. The last option is that $t$ contains a $\text{rttReq}$ with $\text{dly} r$. But then $\text{mte}(t)$ is less than or equal to $r$, so $\text{superfluousMsg}$ will not hold for $\{\delta(t, r')\}$ for any $r' < r$, which
means that the system is tick-invariant with respect to superfluousMsg.

Therefore, all the analyses performed in [15] on this system are sound and complete when using the maximal time sampling strategy.

4.2 The AER/NCA Case Study

The AER/NCA active network protocol suite is the largest Real-Time Maude application analyzed so far [16,10]. Our specification of AER/NCA follows the above guidelines for object-based specification with one insignificant difference: The variable in the tick rule has sort ObjectConfiguration instead of Configuration, which means that the tick rule cannot be applied to states which contain messages at the “outermost level” (i.e., messages that are not inside link objects) in the configuration. This specification is equivalent to one with a tick rule of the form given in page 143 and where the mte of a message is 0.

Since delta is defined to have co-arity ObjectConfiguration, it is sufficient to prove Requirements OO1, OO2, and OO3 for ObjectConfigurations. Furthermore, since the initial states only contain objects of the “smallest” subclasses, it is sufficient to prove the requirements for objects of such subclasses. The time-dependent behavior is straightforward: each object may have a clock and a set of timers. The function delta increases the value of the clocks and decreases the values of the timers according to the elapsed time. mte is defined to be the time remaining until the next timer expires. For example, delta and mte are defined as follows for the objects in the round trip time component of AER/NCA, where sender objects have a clock and receiver objects have both a clock and a timer:

\[
\begin{align*}
\text{vars} & \ R \ R' : \text{Time} \quad \text{var} \ TI : \text{TimeInf} \quad \text{var} \ O : \text{Oid} . \\
\text{eq} \ \delta (< O : \text{RTTsenderAlone} | \text{clock} : R >, R') &= < O : \text{RTTsenderAlone} | \text{clock} : R + R' > . \\
\text{eq} \ \delta (< O : \text{RTTreceivableAlone} | \text{clock} : R, \text{getRTTResendTimer} : TI >, R') &= < O : \text{RTTreceivableAlone} | \text{clock} : R + R', \\
& \quad \text{getRTTResendTimer} : TI \ominus R' > . \\
\text{eq} \ \text{mte}(< O : \text{RTTsenderAlone} | >) &= \text{INF} . \\
\text{eq} \ \text{mte}(< O : \text{RTTreceivableAlone} | \text{getRTTResendTimer} : TI >) &= \text{TI} .
\end{align*}
\]

This is the same definition of delta and mte as for the Node class in the simpler RTT example given in Section 4.1, and satisfaction of the Requirements OO1, OO2, and OO3 can be proved in exactly the same way. When an object contains a set of timers, satisfaction of the requirements can be easily proved using straightforward induction techniques.

Regarding Requirement OO4, it is enough to prove \( \text{mte}(\sigma(l)) = 0 \) for instances of left-hand sides that do not contain messages. Instantaneous rules with messages in their left-hand sides satisfy the requirements for time-robustness, because the application of such a rule cannot follow directly after
a tick step, since a tick step takes a configuration without messages into another configuration without messages. It is easy to inspect each of the 60+ instantaneous rules in AER/NCA to see that either the left-hand side contains a message, or that each instance of a left-hand side of such a rule has \textit{mte} 0.

Finally, we must prove tick-invariance of the search patterns and the atomic propositions that occur in the analysis commands in [16]. Remarkably, \textit{none} of the search patterns or atomic propositions depend on the class attributes that are modified by \texttt{delta}, which implies that all the \textit{unclocked} search patterns and propositions are tick-invariant, since \( t \simeq \texttt{delta}(t, r) \) holds. However, the \textit{clocked} propositions \texttt{nomineeIsBefore} and \texttt{nomineeIsAfter} are \textit{not} tick-invariant.

To summarize, it is very easy to prove that our specification of the large and sophisticated AER/NCA suite, as given in [16], and all but one of the analysis commands in [16], satisfy the requirements for the maximal time sampling analysis to be sound and complete. AER/NCA is essentially parametric in the time domain. If time were dense, our analyses would have provided complete model checking procedures also for dense time. When the time domain is discrete, all behaviors can also be covered by the strategy that always advances time by one time unit. However, we gain a lot in efficiency by advancing time as much as possible. For example, using the maximal time sampling strategy, it took Real-Time Maude 1.5 seconds to find the bug in \texttt{nominee selection} component, while the same search took 160 seconds, i.e., about 100 times longer, when time was always increased by one time unit.

4.2.1 Probabilistic Behavior and Completeness

The AER/NCA suite contains some components with probabilistic behaviors. For example, one use case in the informal specification of the RTT component sets a timer as follows:

\[\ldots\text{Each receiver or repair server starts a } \text{Get-RTT resend timer} \]
\[\text{with a duration of a random variate, uniformly distributed between } 0 \text{ and } 1.0, \text{ times implosionSuppressionInterval.}\]

Although Real-Time Maude at present does not provide explicit support for probabilistic systems, it is however possible to \textit{simulate} such systems using a pseudo-random number generator \texttt{random}, and setting the timer to the value \texttt{random(N) rem 31}, for some ever-changing seed \texttt{N}:

\texttt{rl [R2rs] :}
\[\ldots \Rightarrow \]
\texttt{< O : RTTrepairserverAlone | getRTTResendTimer : random(N) rem 31, \ldots > \ldots}\n
For probabilistic systems completeness of our analyses becomes relative to the probabilistic choices. For example, in the specification of AER/NCA we can only analyze those behaviors that can be reached using the particular random number generator and initial seed value. For the purpose of specifying all possible behaviors, we should replace the above rule with a rule of the form \ldots
for a new variable \( X \) of the appropriate sort. In this way absolute completeness could be regained. Alternatively, if the time domain is discrete, we could force time to stop at each moment in time that is within the desired time interval, and have a nondeterministic choice of whether or not to let the timer expire at that moment. Both of these alternatives would lead to time-robust specifications, so that analyzing these versions with the maximal time sampling strategy would really be complete for all possible behaviors of the AER/NCA protocol. Finally, it is worth mentioning that the rate control component of AER/NCA does not exhibit any probabilistic features, so that the maximal time sampling strategy analyses really cover all possible behaviors of this component.

4.3 The OGDC Wireless Sensor Network Algorithm

We have recently modeled and analyzed the sophisticated OGDC density control algorithm [19] for wireless sensor networks in Real-Time Maude [17]. Our object-based specification of OGDC uses the specification techniques given above. Given the complexity of the specification, it is remarkable how easy it is to prove time-robustness by proving \( OO1 \) to \( OO4 \). For example, it follows directly that the \( mte \) of an instance of the left-hand side of an instantaneous rule is 0, and proving the other requirements amounts to proving properties like \((x \text{ monus } y) \text{ monus } z = x \text{ monus } (y + z)\). Tick-invariance of the propositions and search patterns used in the analyses in [17] follows trivially, since \( \delta \) does not modify the attributes that define these patterns and propositions.

Besides the fact that proving completeness of the analysis using maximal time sampling is almost trivial, we gain much by using maximal time sampling, since time is measured in milliseconds in this algorithm, while one round of the algorithm lasts for 1,000 seconds. An analysis based on visiting each moment in time would be unfeasible for such systems. Indeed, none of the analysis commands in [17] could terminate after several hours when we used the default time sampling strategy which increases time by one millisecond in each tick step.

OGDC is a probabilistic algorithm, and we have specified it for simulation purposes, so that the completeness of our analysis is again relative to the treatment of probabilistic behavior. Nevertheless, if we modify our specification by modeling probabilistic features by “pure nondeterminism” as indicated in Section 4.2, we would still have a time-robust specification whose analysis using maximal time sampling would be complete for all possible timed fair behaviors of the OGDC algorithm.
5 Conclusions

We have presented general criteria that, under the maximal time sampling strategy, guarantee completeness of Real-Time Maude formal analyses for a real-time system. We have considered large classes of systems, including many object-oriented systems of interest, to which our criteria can be applied; and we have characterized simple conditions under which our criteria can be checked for such systems.

The practical value of our results is that they apply to many systems outside the scope of the known automaton-based decision procedures; and that they greatly increases the level of assurance that can be attached to formal analyses performed in Real-Time Maude for such systems. Indeed, for systems meeting our criteria, our results yield a complete semi-decision procedure for violations of invariants; and a complete decision procedure for satisfaction of bounded LTL properties without the \( \Box \) operator.

The following research directions suggested by this work seem worth investigating:

- further generalizing our criteria to encompass an even broader class of systems for which completeness can be guaranteed;
- development of new abstraction techniques for real-time systems that extend or complement those presented here;
- mechanization of the process of checking proof obligations ensuring our correctness criteria; and
- development of additional case studies to experiment with our techniques and to further improve and simplify their foundations and their tool support.

Acknowledgments. We thank the anonymous referees for helpful comments on earlier versions of this paper. Partial support of this research by ONR Grant N00014-02-1-0715 and by NSF Grant CNS 05-24516 is gratefully acknowledged.

References


The Real-Time Maude Specification of a Simple Round Trip Time Protocol

This appendix presents the Real-Time Maude specification and the analysis commands of the simple protocol for finding the round trip time between pairs of nodes in a network that was described in [15]. The specification below treats the case when it takes a message exactly time $\text{MIN-TRANS-TIME}$ to travel from source to destination.

\[
\text{(tomod RTT is protecting NAT-TIME-DOMAIN-WITH-INF ) .}
\]
\[
\text{op MAX-DELAY : } \rightarrow \text{Time } . \quad \text{eq MAX-DELAY} = 4 .
\]
\[
\text{op MIN-TRANS-TIME : } \rightarrow \text{Time } . \quad \text{eq MIN-TRANS-TIME} = 1 .
\]

\[
\text{class Node | clock : Time, rtt : TimeInf,}
\]
\[
\text{nbr : Oid, timer : TimeInf .}
\]

\[
\text{msgs rttReq rttResp : Oid Oid Time } \rightarrow \text{Msg .}
\]

\[
\text{msg findRtt : Oid } \rightarrow \text{Msg .} \quad \text{- - start a run}
\]

\[
\text{--- Dly message wrappers:}
\]

\[
\text{sort DlyMsg .}
\]

\[
\text{subsorts Msg < DlyMsg < NEConfiguration .}
\]

\[
\text{op dly : Msg Time } \rightarrow \text{DlyMsg [ctor right id: 0] .}
\]

\[
\text{vars O O' : Oid .} \quad \text{vars R R' : Time .} \quad \text{var TI : TimeInf .}
\]

\[
\text{--- start a session, and set timer:}
\]

\[
\text{rl [startSession] :}
\]

\[
\text{findRtt(O) } < \text{ O : Node | clock : R, nbr : O' } \Rightarrow
\]

\[
< \text{ O : Node | timer : MAX-DELAY }>
\]

\[
\text{dly(rttReq(O', O, R), MIN-TRANS-TIME) .}
\]

\[
\text{--- respond to request:}
\]

\[
\text{rl [rttResponse] :}
\]

\[
\text{rttReq(O, O', R) } < \text{ O : Node | } \Rightarrow
\]

\[
< \text{ O : Node | } \quad \text{dly(rttResp(O', O, R), MIN-TRANS-TIME) .}
\]

\[
\text{--- received resp within time MAX-DELAY;}
\]

\[
\text{--- record rtt value and turn off timer:}
\]

\[
\text{crl [treatRttResp] :}
\]
rttResp(O, O', R) < O : Node | clock : R' > =>
    < O : Node | rtt : (R' monus R), timer : INF >
if (R' monus R) < MAX-DELAY .

--- ignore and discard too old message:
crl [ignoreOldResp] :
    rttResp(O, O', R) < O : Node | clock : R' > => < O : Node | >
    if (R' monus R) >= MAX-DELAY .

--- start new round and reset timer when timer expires:
rl [tryAgain] :
    < O : Node | timer : 0, clock : R, nbr : O' > =>
    < O : Node | timer : MAX-DELAY >
dly(rttReq(O', O, R), MIN-TRANS-TIME) .

--- tick rule should not advance time beyond expiration of a timer:
crl [tick] :
    {C:Configuration} => {delta(C:Configuration, R)} in time R
    if R <= mte(C:Configuration) [nonexec] .

--- the functions mte and delta:
op delta : Configuration Time -> Configuration [frozen (1)] .
eq delta(none, R) = none .
eq delta(NEC:NEConfiguration NEC':NEConfiguration, R) =
    delta(NEC:NEConfiguration, R) delta(NEC':NEConfiguration, R) .
eq delta(< O : Node | clock : R, timer : TI >, R') =
    < O : Node | clock : R + R', timer : TI monus R' > .
eq delta(dly(M:Msg, R'), R) = dly(M:Msg, R' monus R) .

endtom)

--- Define an initial state with three nodes:
(tomod RTT-I is including RTT .
    ops n1 n2 n3 : -> Oid .
    op initState : -> GlobalSystem .
eq initState =
    {findRtt(n1) findRtt(n2) findRtt(n3)
    < n1 : Node | clock : 0, timer : INF, nbr : n2, rtt : INF >
    < n2 : Node | clock : 0, timer : INF, nbr : n3, rtt : INF >
    < n3 : Node | clock : 0, timer : INF, nbr : n1, rtt : INF >} .
endtom)

(set tick max def 10 .)   --- use maximal time sampling strategy
(tsearch [1]
\begin{verbatim}
initState =>* {C:Configuration
  < O:Oid : Node | rtt : X:Time,
  ATTS:AttributeSet >}
such that X:Time >= 4
in time <= 100 .)

(tsearch [1]
  initState =>* {C:Configuration
  < n1 : Node | rtt : 2, ATTS:AttributeSet >
  < n2 : Node | rtt : 2, ATTS':AttributeSet >}
in time <= 100 .)

(tomod MC-RTT is including TIMED-MODEL-CHECKER . protecting RTT-I .
op superfluousMsg : -> Prop [ctor] .
vars REST : Configuration .
vars O O' : Oid .
vars R R' R'' R''' : Time .
ceq {REST < O : Node | rtt : R, clock : R' > dly(rttReq(O', O, R''), R'')}
  |= superfluousMsg = true if R'' + MAX-DELAY > R' .
ceq {REST < O : Node | rtt : R, clock : R' > dly(rttResp(O, O', R''), R'')}
  |= superfluousMsg = true if R'' + MAX-DELAY > R' .
endtom)

(mc initState |=t [] ~ superfluousMsg in time <= 100 .)
\end{verbatim}
Abstract
This paper presents GOM, a language for describing abstract syntax trees and generating a Java implementation for those trees. GOM includes features allowing the user to specify and modify the interface of the data structure. These features provide in particular the capability to maintain the internal representation of data in canonical form with respect to a rewrite system. This explicitly guarantees that the client program only manipulates normal forms for this rewrite system, a feature which is only implicitly used in many implementations.

1 Introduction
Rewriting and pattern-matching are of general use for describing computations and deduction. Programming with rewrite rules and strategies has been proven most useful for describing computational logics, transition systems or transformation engines, and the notions of rewriting and pattern matching are central notions in many systems, like expert systems (JRule), programming languages based on rewriting (ELAN, Maude, OBJ) or functional programming (ML, Haskell).

In this context, we are developing the Tom system [10], which consists of a language extension adding syntactic and associative pattern matching and strategic rewriting capabilities to existing languages like Java, C and OCaml. This hybrid approach is particularly well-suited when describing transformations of structured entities like trees/terms and XML documents.

One of the main originalities of this system is to be data structure independent. This means that a mapping has to be defined to connect algebraic data structures, on which pattern matching is performed, to low-level data structures, that correspond to the implementation. Thus, given an algebraic data structure definition, it is needed to implement an efficient support for this definition in the language targeted by the Tom system, as Java or C do not provide such data structures. Tools like ApiGen [13] and Vas, which is a human readable language for ApiGen input where used previously for generating
such an implementation to use with Tom.

However, experience showed that providing an efficient term data structure implementation is not enough. When implementing computational logics or transition systems with rewriting and equational matching, it is convenient to consider terms modulo a particular theory, as identity, associativity, commutativity, idempotency, or more problem specific equations [9].

Then, it becomes crucial to provide the user of the data structure a way to conveniently describe such rules, and to have the insurance that only chosen equivalence class representatives will be manipulated by the program. This need shows up in many situations. For instance when dealing with abstract syntax trees in a compiler, and requiring constant folding or unboxing operators protecting particular data structures.

GOM is a language for describing multi-sorted term algebras designed to solve this problem. Like ApiGen, Vas or ASDL [15], its goal is to allow the user of an imperative or object oriented language to describe concisely the algebra of terms he wants to use in an application, and to provide an (efficient) implementation of this algebra.

Moreover, it provides a mechanism to describe normalization functions for the operators, and it ensures that all terms manipulated by the user of the data structure are normal with respect to those rules. GOM includes the same basic functionality as ApiGen and Vas, and ensures that the data structure implementation it provides are maximally shared. Also, the generated data structure implementation supports the visitor combinator [14] pattern, as the strategy language of Tom relies on this pattern.

Even though GOM can be used in any Java environment, its features have been designed to work in synergy with Tom. Thus, it is able to generate correct Tom mappings for the data structure (i.e. being formal anchors [6]). GOM provides a way to define computationally complex constructors for a data structure. It also ensures those constructors are used, and that no raw term can be constructed. Private types [8] in the OCaml language do provide a similar functionality by hiding the type constructors in a private module, and exporting construction functions. However, using private types or normal types is made explicit to the user, while it is fully transparent in GOM. MOCA, developed by Frédéric Blanqui and Pierre Weis is a tool that implements normalization functions for theories like associativity or distributivity for OCaml types. It internally uses private types to implement those normalization functions and ensure they are used, but could also provide such an implementation for GOM.

The rest of the paper is organized as follows: in Section 2, to motivate the introduction of GOM, we describe the Tom programming environment and its facilities. Section 3 presents the GOM language, its semantics and some simple use cases. After presenting how GOM can cooperate with Tom in Section 4, we expose in Section 5 the example of a prover for the calculus of structures [3] showing how the combination of GOM and Tom can help
producing a reliable and extendable implementation for a complex system. We conclude with summary and discussions in Section 6.

2 The Tom language

Tom is a language extension which adds pattern matching primitives to existing imperative languages. Pattern-matching is directly related to the structure of objects and therefore is a very natural programming language feature, commonly found in functional languages. This is particularly well-suited when describing various transformations of structured entities like, for example, trees/terms, hierarchized objects, and XML documents.

The main originality of the Tom system is its language and data-structure independence. From an implementation point of view, it is a compiler which accepts different native languages like C or Java and whose compilation process consists in translating the matching constructs into the underlying native language.

It has been designed taking into account experience about efficient compilation of rule-based systems [7], and allows the definition of rewriting systems, rewriting rules and strategies. For an interested reader, design and implementation issues related to Tom are presented in [10].

Tom is based on the notion of formal anchor presented in [6], which defines a mapping between the algebraic terms used to express pattern matching and the actual objects the underlying language manipulates. Thus, it is data structure independent, and customizable for any term implementation.

For example, when using Java as the host language, the sum of two integers can be described in Tom as follows:

```java
Term plus(Term t1, Term t2) {
  //match(Nat t1, Nat t2) {
  x,zero  -> { return x; }
  x,suc(y) -> { return suc(plus(x,y)); }
  }
}
```

Here the definition of plus is specified functionally, but the function plus can be used as a Java function to perform addition. Nat is the algebraic sort Tom manipulates, which is mapped to Java objects of type Term. The mapping between the actual object Term and the algebraic view Nat has to be provided by the user.

The language provides support for matching modulo sophisticated theories. For example, we can specify a matching modulo associativity and neutral element (also known as list-matching) that is particularly useful to model the exploration of a search space and to perform list or XML based transformations. To illustrate the expressivity of list-matching we can define the search of a zero in a list as follows:
boolean hasZero(TermList l) {
  %match(NatList l) {
    conc(X1*,zero,X2*) -> { return true; }
  }
  return false;
}

In this example, list variables, annotated by a * should be instantiated by a (possibly empty) list. Given a list, if a solution to the matching problem exists, a zero can be found in the list and the function returns true, false otherwise, since no zero can be found.

Although this mechanism is simple and powerful, it requires a lot of work to implement an efficient data structure for a given algebraic signature, as well as to provide a formal anchor for the abstract data structure. Thus we need a tool to generate such an efficient implementation from a given signature. This is what tools like ApiGen [13] do.

However, ApiGen itself only provides a tree implementation, but does not allow to add behavior and properties to the tree data structure, like defining ordered lists, neutral element or constant propagation in the context of a compiler manipulating abstract syntax tree. Hence the idea to define a new language that would overcome those problems.

3 The GOM language

We describe here the GOM language and its syntax, and present an example data-structure description in GOM. We first show the basic functionality of GOM, which is to provide an efficient implementation in Java for a given algebraic signature. We then detail what makes GOM suitable for efficiently implement normalized rewriting [9], and how GOM allows us to write any normalization function.

3.1 Signature definitions

An algebraic signature describes how a tree-like data structure should be constructed. Such a description contains Sorts and Operators. Operators define the different node shapes for a certain sort by their name and the names and sorts of their children. Formalisms to describe such data structure definitions include ApiGen [13], XML Schema, ML types, and ASDL [15].

To this basic signature definition, we add the notion of module as a set of sorts. This allows to define new signatures by composing existing signatures, and is particularly useful when dealing with huge signatures, as can be the abstract syntax tree definition of a compiler. Figure 1 shows a simplified syntax for GOM signature definition language. In this syntax, we see that a module can import existing modules to reuse its sorts and operators definitions. Also, each module declares the sorts it defines with the sorts keyword, and declares
operators for those sorts with productions. This syntax is strongly influenced by the syntax of SDF \[12\], but simpler, since it intends to deal with abstract syntax trees, instead of parse trees. One of its peculiarities lies in the productions using the \( \ast \) symbol, defining variadic operators. The notation \( \ast \) is the same as in \[1, Section 2.1.6\] for a similar construction, and can be seen as a family of operators with arities in \([0, \infty)[\).

We will now consider a simple example of GOM signature for booleans:

\[
\text{module Boolean} \\
\text{sorts Bool} \\
\text{abstract syntax} \\
\begin{align*}
\text{True} & \to \text{Bool} \\
\text{False} & \to \text{Bool} \\
\text{not}(b: \text{Bool}) & \to \text{Bool} \\
\text{and}(\text{lhs}: \text{Bool}, \text{rhs}: \text{Bool}) & \to \text{Bool} \\
\text{or}(\text{lhs}: \text{Bool}, \text{rhs}: \text{Bool}) & \to \text{Bool}
\end{align*}
\]

From this description, GOM generates a Java class hierarchy where to each sort corresponds an abstract class, and to each operator a class extending this sort class. The generator also creates a factory class for each module (in this example, called BooleanFactory), providing the user a single entry point for creating objects corresponding to the algebraic terms.

Like ApiGen and Vas, GOM relies on the ATerm \[11\] library, which provides an efficient implementation of unsorted terms for the C and Java languages, as a basis for the generated classes. The generated data structure can then be characterized by strong typing (as provided by the Composite pattern used for generation) and maximal subterm sharing. Also, the generated class hierarchy does provide support for the visitor combinator pattern \[14\], allow-
ing the user to easily define arbitrary tree traversals over GOM data structures using high level constructs (providing congruence operators).

3.2 Canonical representatives

When using abstract data types in a program, it is useful to also define a notion of canonical representative, or ensure some invariant of the structure. This is particularly the case when considering an equational theory associated to the terms of the signature, such as associativity, commutativity or neutral element for an operator, or distributivity of one operator over another one.

Considering our previous example with boolean, we can consider the De Morgan rules as an equational theory for booleans. De Morgan’s laws state $A \lor B = \overline{A} \land \overline{B}$ and $A \land B = \overline{A} \lor \overline{B}$. We can orient those equations to get a confluent and terminating rewrite system, suitable to implement a normalization system, where only boolean atoms are negated. We can also add a rule for removing duplicate negation. We obtain the system:

$A \lor B \rightarrow \overline{A} \land \overline{B}$

$A \land B \rightarrow \overline{A} \lor \overline{B}$

$\overline{\overline{A}} \rightarrow A$

GOM’s objective is to provide a low level system for implementing such normalizing rewrite systems in an efficient way, while giving the user control on how the rules are applied. To achieve this goal, GOM provides a hook mechanism, allowing to define arbitrary code to execute before, or replacing the original construction function of an operator. This code can be any Java or Tom code, allowing to use pattern matching to specify the normalization rules. To allow hooks definitions, we add to the GOM syntax the definitions for hooks, and add $\langle \text{OpHook} \rangle$ and $\langle \text{FactoryHook} \rangle$ to the productions:

$$\langle \text{FactoryHook} \rangle ::= \text{factory} \{ \langle \text{TomCode} \rangle \}$$

$$\langle \text{OpHook} \rangle ::= \langle \text{Symbol} \rangle : \langle \text{Operation} \rangle \{ \langle \text{TomCode} \rangle \}$$

$$\langle \text{Operation} \rangle ::= \langle \text{OperationType} \rangle ( (\langle \text{Identifier} \rangle)^* )$$

$$\langle \text{OperationType} \rangle ::= \text{make} \mid \text{make_before} \mid \text{make_after} \mid \text{make_insert} \mid \text{make_after_insert} \mid \text{make_before_insert}$$

$$\langle \text{TomCode} \rangle ::= \{ \ldots \}$$

A factory hook $\langle \text{FactoryHook} \rangle$ is attached to the module, and allows to define additional Java functions. We will see in Section 5.3 an example of use for such a hook. An operator hook $\langle \text{OpHook} \rangle$ is attached to an operator definition, and allows to extend or redefine the construction function for this operator. Depending on the $\langle \text{OperationType} \rangle$, the hook redefines the construction function (make), or insert code before (make_before) or after (make_after) the construction function. Those hooks take as many arguments as the operator they modify has children. We also define operation types with an appended insert, used for variadic operators. Those hooks only take two arguments, when the
operator they apply to is variadic, and allow to modify the operation of adding one element to the list of arguments of a variadic operator.

Such **hooks** can be used to define the boolean normalization system:

```plaintext
module Boolean
sorts Bool
abstract syntax
   True         -> Bool
   False        -> Bool
   not(x:Bool)  -> Bool
   and(l:Bool,r:Bool) -> Bool
   or(l:Bool,r:Bool)  -> Bool
not:make(arg) {
   %match(Bool arg) {
      not(x) -> { return 'x; }
      and(l,r) -> { return 'or(not(l),not(r)); }
      or(l,r)  -> { return 'and(not(l),not(r)); }
   }
   return 'make_not(arg);
}
```

We see in this example that it is possible to use Tom in the **hook** definition, and to use the algebraic signature being defined in GOM in the **hook** code. This lets the user define **hooks** as rewriting rules, to obtain the normalization system. The signature in the case of GOM is extended to provide access to the default construction function of an operator. This is done here with the **make_not(arg)** call.

When using the **hook** mechanism of GOM, the user has to ensure that the normalization system the hooks define is terminating and confluent, as it will not be enforced by the system. Also, combining hooks for different equational theories in the same signature definition can lead to non confluent systems, as combining rewrite systems is not a straightforward task.

However, a higher level strata providing completion to compute normalization functions from their equational definition, and allowing to combine theories and rules could take advantage of GOM’s design to focus on high level tasks, while getting maximal subterm sharing, strong typing of the generated code and **hooks** for implementing the normalization functions from the GOM strata. GOM can then be seen as a reusable component, intended to be used as a tool for implementing another language (as ApiGen was used as basis for **ASF+SDF** [2]) or as component in a more complex architecture.

### 4 The interactions between GOM and Tom

The GOM tool is best used in conjunction with the Tom compiler. GOM is used to provide an implementation for the abstract data type to be used in a
Tom program. The GOM data structure definition will also contain the description of the invariants the data structure has to preserve, by the mean of hooks, such that it is ensured the Tom program will only manipulate terms verifying those invariants. Starting from an input datatype signature definition, GOM generates an implementation in Java of this data structure (possibly using Tom internally) and also generates an anchor for this data structure implementation for Tom (See Figure 2). The users can then write code using the match construct on the generated mapping and Tom compiles this to plain Java. The dashed box represents the part handled by the GOM tool, while the grey boxes highlight the source files the user writes. The generated code is characterized by strong typing combined with a generic interface and by maximal sub-term sharing for memory efficiency and fast equality checking, as well as the insurance the hooks defined for the data structure are always applied, leading to canonical terms. Although it is possible to manually implement a data structure satisfying those constraints, it is difficult, as all those features are strongly interdependent. Nonetheless, it is then very difficult to let the data structure evolve when the program matures while keeping those properties, and keep the task of maintaining the resulting program manageable.

In the following example, we see how the use of GOM for the data structure definition and Tom for expressing both the invariants in GOM and the rewriting rules and strategy in the program leads to a robust and reliable implementation for a prover in the structure calculus.
We describe here a real world example of a program written using GOM and Tom together. We implement a prover for the calculus of structure [3] where some rules are promoted to the level of data structure invariants, allowing a simpler and more efficient implementation of the calculus rules. Those invariants and rules have been shown correct with respect to the original calculus, leading to an efficient prover that can be proven correct. Details about the correctness proofs and about the proof search strategy can be found in [5]. We concentrate here on the implementation using GOM.

5.1 The approach

When building a prover for a particular logic, and in particular for the system BV in the structure calculus, one needs to refine the strategy of applying the calculus rules. This is particularly true with the calculus of structure, because of deep inference, non confluence of the calculus and associative-commutative structures.

We describe here briefly the system BV, to show how GOM and Tom can help to provide a robust and efficient implementation of such a system.

Atoms in BV are denoted by $a, b, c, \ldots$ Structures are denoted by $R, S, T, \ldots$ and generated by

$$S ::= \circ | a | \langle S; \ldots; S \rangle | [S, \ldots, S] | (S, \ldots, S) | \overline{S}$$

where $\circ$, the unit, is not an atom. $\langle S; \ldots; S \rangle$ is called a seq structure, $[S, \ldots, S]$ is called a par structure, and $(S, \ldots, S)$ is called a copar structure, $\overline{S}$ is the negation of the structure $S$. A structure $R$ is called a proper par structure if $R = [R_1, R_2]$ where $R_1 \neq \circ$ and $R_2 \neq \circ$. A structure context, denoted as in $S\{\}$, is a structure with a hole. We use this notation to express the deduction rules for system BV, and will omit context braces when there is no ambiguity.

The rules for system BV are simple, provided some equivalence relations on BV terms. The seq, par and copar structures are associative, par and copar being commutative too. Also, $\circ$ is a neutral element for seq, par and copar structures, and a seq, par or copar structure with only one substructure is equivalent to its content. Then the deduction rules for system BV can be expressed as in Figure 3.

Because of the contexts in the rules, the corresponding rewriting rules can
be applied not only at the top of a structure, but also on each subterm of a structure, for implementing deep inference. Deep inference then, combined with associativity, commutativity and $\circ$ as a neutral element for seq, par and copar structures leads to a huge amount of non-determinism in the calculus. A structure calculus prover implementation following strictly this description will have to deal with this non-determinism, and handle a huge search space, leading to inefficiency [4].

The approach when using GOM and Tom will be to identify canonical representatives, or preferred representatives for equivalence classes, and implement the normalization for structures leading to the selection of the canonical representative by using GOM’s hooks. This process requires to define the data structure first, and then define the normalization. This normalization will make sure all units $\circ$ in seq, par and copar structures are removed, as $\circ$ is a neutral for those structures. We will also make sure the manipulated structures are flattened, which corresponds to selecting a canonical representative for the associativity of seq, par and copar, and also that subterms of par and copar structures are ordered, taking a total order on structures, to take commutativity into account.

When implementing the deduction rule, it will be necessary to take into account the fact that the prover only manipulates canonical representatives. This leads to simpler rules, and allow some new optimizations on the rules to be performed.

5.2 The data structure

We first have to give a syntactic description of the structure data-type the BV prover will use, to provide an object representation for the seq, par and copar structures ($\langle R; T \rangle$, $[R, T]$ and $(R, T)$). In our implementation, we considered these constructors as unary operators which take a list of structures as argument. Using GOM, the considered data structure can be described by the following signature:

```plaintext
module Struct
  imports
  public
    sorts Struc StrucPar StrucCop StrucSeq
  abstract syntax
    o -> Struc
    a -> Struc
    b -> Struc
    c -> Struc
    d -> Struc
    ...other atom constants
    neg(a:Struc) -> Struc
    concPar( Struc* ) -> StrucPar
```
To represent structures, we define first some constant atoms. Among them, the \( \circ \) constant will be used to represent the unit \( \circ \). The neg operator builds the negation of its argument. The grammar rule \( \text{par} (\text{StrucPar}) \rightarrow \text{Struc} \) defines a unary operator \( \text{par} \) of sort \( \text{Struc} \) which takes a \( \text{StrucPar} \) as unique argument. Similarly, the rule \( \text{concPar} (\text{Struc*}) \rightarrow \text{StrucPar} \) defines the \( \text{concPar} \) operator of sort \( \text{StrucPar} \). The syntax \( \text{Struc*} \) indicates that \( \text{concPar} \) is a variadic-operator which takes an indefinite number of \( \text{Struc} \) as arguments. Thus, by combining \( \text{par} \) and \( \text{concPar} \) it becomes possible to represent the structure \([a, [b, c]]\) by \( \text{par} (\text{concPar} (a, b, c)) \). Note that this structure is flattened, but with this description, we could also use nested \( \text{par} \) structures, as in \( \text{par} (\text{concPar} (a, \text{par} (\text{concPar} (b, c)))) \) to represent this structure. \( (R, T) \) and \( \langle R; T \rangle \) are represented in a similar way, using \( \text{cop}, \text{seq}, \text{concCop}, \) and \( \text{concSeq} \).

5.3 The invariants, and how they are enforced

So far, we can manipulate objects, like \( \text{par} (\text{concPar} ()) \), which do not necessarily correspond to intended structures. It is also possible to have several representations for the same structure. Hence, \( \text{par} (\text{concPar} (a)) \) and \( \text{cop} (\text{concCop} (a)) \) both denote the structure \( a \), as \( \langle R \rangle \approx [R] \approx (R) \approx R \).

Thus, we define the canonical (prefered) representative by ensuring that

- \([], \langle \rangle\) and () are reduced when containing only one sub-structure:
  \[ \text{par} (\text{concPar} (x)) \rightarrow x \]

- nested structures are flattened, using the rule:
  \[ \text{par} (\text{concPar} (a_1, \ldots, a_i, \text{par} (\text{concPar} (x_1, \ldots, x_n)), b_1, \ldots, b_j)) \]
  \[ \rightarrow \text{par} (\text{concPar} (a_1, \ldots, a_i, x_1, \ldots, x_n, b_1, \ldots, b_j)) \]

- subterms are sorted (according to a given total lexical order \(<\)):
  \[ \text{concPar} (\ldots, x_i, \ldots, x_j, \ldots) \rightarrow \text{concPar} (\ldots, x_j, \ldots, x_i, \ldots) \text{ if } x_j < x_i. \]

This notion of canonical form allows us to efficiently check if two terms represent the same structure with respect to commutativity of those connectors, neutral elements and reduction rules.

The first invariant we want to maintain is the reduction of singleton for \( \text{seq} \), \( \text{par} \) and \( \text{cop} \) structures. If we try to build a \( \text{cop} \), \( \text{par} \) or \( \text{seq} \) with an empty list of structures, then the creation function shall return the unit \( \circ \). Else if the list contains only one element, it has to return this element. Otherwise, it will just build the requested structure. As all manipulated terms are canonical forms, we do not have for this invariant to handle the case of a structure list.
containing the unit, as it will be enforced by the list invariants. This behavior can be implemented as a hook for the seq, par and cop operators.

```plaintext
par(par1:StrucPar) -> Struc
par:make (1) {
  %match(StrucPar l) {
    concPar() -> { return 'o(); }
    concPar(x)-> { return 'x; }
  }
  return 'make_par(l);
}
```

This simple hook implements the invariant for singletons for par, and use a call to the Tom constructor make_par(1) to call the intern constructor (without the normalization process), to avoid an infinite loop. Similar hooks are added to the GOM description for cop and seq operators. We see here how the pattern matching facilities of Tom embedded in GOM can be used to easily implement normalization strategies.

The hooks for normalizing structure lists are more complex. They first require a total order on structures. This can be easily provided as a function, defined in a factory hook. The comparison function we provide here uses the builtin translation of GOM generated data structures to text to implement a lexical total order. A more specific (and efficient) comparison function could be written, but for the price of readability.

```plaintext
factory {
  public int compareStruc(Object t1, Object t2) {
    String s1 = t1.toString();
    String s2 = t2.toString();
    int res = s1.compareTo(s2);
    return res;
  }
}
```

Once this function is provided, we can define the hooks for the variadic operators concSeq, concPar and concCop. The hook for concSeq is the simplest, since the ⟨⟩ structures are only associative, with o as neutral element. Then the corresponding hook has to remove the units, and flatten nested seq.

```plaintext
concSeq( Struc* ) -> StrucSeq
concSeq:make_insert(e,l) {
  %match(Struc e) {
    o() -> { return 1; }
    seq(concSeq(L*)) -> { return 'concSeq(L*,1*); }
  }
  return 'make_concSeq(e,l);
}
```
This hook only checks the form of the element to add to the arguments of the variadic operator, but does not use the shape of the previous arguments. The hooks for concCop and concPar are similar, but they do examine also the previous arguments, to perform sorted insertion of the new argument. This leads to a sorted list of arguments for the operator, providing a canonical representative for commutative structures.

```tom
concPar( Struc* ) -> StrucPar
concPar:make_insert(e,l) {
  %match(Struc e) {
    o() -> { return l; }
    par(concPar(L*)) -> { return 'concPar(L*,l*); }
  }
  %match(StrucPar l) {
    concPar(head,tail*) -> {
      if(!compareStruc(e, head) < 0)) {
        return 'make_concPar(head,concPar(e,tail*));
      }
    }
  }
  return 'make_concPar(e,l);
}
```

The associative matching facility of Tom is used to examine the arguments of the variadic operator, and decide whether to call the builtin construction function, or perform a recursive call to get a sorted insertion.

As the structure calculus verify the De Morgan rules for the negation, we could write a hook for the neg construction function applying the De Morgan rules as in Section 3.2 to ensure only atoms are negated. This will make implementing the deduction rules even simpler, since there is then no need to propagate negations in the rules.

5.4 The rules

Once the data structure is defined, we can implement proof search in system BV in a Tom program using the GOM defined data structure by applying rewriting rules corresponding to the calculus rules to the input structure repeatedly, until reaching the goal of the prover (usually, the unit $\circ$).

Those rules are expressed using Tom’s pattern matching over the GOM data structure. They are kept simple because the equivalence relation over structures is integrated in the data structure with invariants. In this example, $[]$ and $(\cdot)$ structures are associative and commutative, while the canonical representatives we use are sorted and flattened variadic operators.

For instance, the rule $s$ of Figure 3 can be expressed as the two rules $[(R,T),U] \rightarrow ([R,U],T)$ and $[(R,T),U] \rightarrow ([T,U],R)$, using only associative matching instead of associative commutative matching. Then, those rules are
encoded by the following match construct, which is placed into a strategy implementing rewriting in arbitrary context (congruence) to get deep inference, the c collection being used to gather multiple results:

```java
%match (Struct t) {
    par (concPar(X1*,cop(concCop(R*,T*)),X2*,U,X3*)) -> {
        if ('T*.isEmpty() || 'R*.isEmpty() ) { }
        else {
            StructPar context = 'concPar(X1*,X2*,X3*);
            if (canReact ('R*,'U)) {
                StructPar parR = cop2par ('R*);
                // transform a StructCop into a StructPar
                Struct elt1 = 'par (concPar (cop (concCop (par (concPar (parR*,U)),T*)),context*));
                c.add (elt1);
            }
            if (canReact ('T*,'U)) {
                StructPar parT = cop2par ('T*);
                Struct elt2 = 'par (concPar (cop (concCop (par (concPar (parT*,U)),R*)),context*));
                c.add (elt2);
            }
        }
    }
}
```

We ensure that we do not execute the right-hand side of the rule if either R or T are empty lists. The other tests implement restrictions on the application of the rules reducing the non-determinism. This is done by using an auxiliary predicate function `canReact(a,b)` which can be expressed using all the expressive power of both Tom and Java in a factory hook. The interested reader is referred to [5] for a detailed description of those restrictions.

Also, the search strategy can be carefully crafted using both Tom and Java constructions, to achieve a very fine grained and evolutive strategy, where usual algebraic languages only allow breadth-first or depth-first strategies, but do not let the programmer easily define a particular hybrid search strategy. While the Tom approach of search strategies may lead to more complex implementations for simple examples (as the search space has to be handled explicitly), it allows us to define fine and efficient strategies for complex cases.

The implementation of a prover for system BV with GOM and Tom leads not only to an efficient implementation, allowing to cleanly separate concerns about strategy, rules and canonical representatives of terms, but also to an implementation that can be proven correct, because most parts are expressed with the high level constructs of GOM and Tom instead of pure Java. As the data structure invariants in GOM and the deduction rules in Tom are defined algebraically, it is possible to prove that the implemented system is correct and complete with respect to the original system [5], while benefiting from the expressive power and flexibility of Java to express non algebraic concerns (like
building a web applet for the resulting program, or sending the results in a network).

6 Conclusion

We have presented the GOM language, a language for describing algebraic signatures and normalization systems for the terms in those signatures. This language is kept low level by using Java and Tom to express the normalization rules, and by using hooks for describing how to use the normalizers. This allows an efficient implementation of the resulting data structure, preserving properties important to the implementation level, such as maximal subterm sharing and a strongly typed implementation.

We have shown how this new tool interacts with the Tom language. As Tom provides pattern matching, rewrite rules and strategies in imperative languages like C or Java, GOM provides algebraic data structures and canonical representatives to Java. Even though GOM can be used simply within Java, most benefits are gained when using it with Tom, allowing to integrate formal algebraic developments into mainstream languages. This integration can allow to formally prove the implemented algorithms with high level proofs using rewriting techniques, while getting a Java implementation as result.

We have applied this approach to the example of system BV in the structure calculus, and shown how the method can lead to an efficient implementation for a complex problem (the implemented prover can tackle more problems than previous rule based implementation [5]).

As the compilation process of Tom’s pattern matching is formally verified and shown correct [6], proving the correctness of the generated data structure and normalizers with respect to the GOM description would allow to expand the trust path from the high level algorithm expressed with rewrite rules and strategies to the Java code generated by the compilation of GOM and Tom. This allows to not only prove the correctness of the implementation, but also to show that the formal parts of the implementation preserve the properties of the high level rewrite system, such as confluence or termination.

References


Abstract

A modular approach exploits features of Maude to produce specifications that are economical in state space, size of state, and rewriting performance. A cycle of rewrites is introduced as a framework for the composition of deterministic and concurrent features of a system. Examples of deterministic features are the flexible monitoring of a model, and a versatile and efficient representation of state; both of which are very well suited to model sensor networks. Some results of applying the proposed high-level techniques show a performance improvement of two orders of magnitude.

1 Introduction

Formal, executable specifications support formal prototyping and experimentation, which can be extremely useful in the design of complex systems and protocols. Much research is being devoted to improve the efficiency and expand the applicability of these methods. The work presented here aims to contribute to that effort. The basic idea underlying it is captured metaphorically in the motto *One size DOES NOT fit all*. The “size” is the computational effort, the size of the state, and the size of the state space. It should “fit” perfectly, as a stretch garment: neither too large and wasteful, nor too small and restrictive, and adjusting to movement. So this individualized optimization distinguishes not only among different specifications, or even different computations of a specification, but individual steps in a computation. This basic idea leads to a versatile representation of the state, which changes throughout the computation to provide custom-made, optimized performance. Guided by three most simple principles, and exploiting features of Maude [1][2] and its extension Real-Time Maude [11][8], this paper proposes a method for developing modular, object-oriented, real-time models of wireless networks that provide that individualized performance optimization. These models accommodate the demands of modelling sensor networks.

The techniques that underlie the proposed method are very opportunistic.
The notion of sensor network \cite{5,12} serves to dramatically motivate their need and see the opportunities. A sensor network consists of hundreds or thousands of battery-operated devices that communicate wirelessly, and are capable of limited computation and sensing. Much research is needed to make the notional sensor network a reality. The purpose of such a network is to provide distributed sensing and computation to satisfy the queries of a few observers. In a sensor network, because of the wireless nature of its communication, the exchange between an observer and masses of nodes occurs by means of propagations of messages through the network, which may cover the entire network or just a section.

To illustrate the need to adjust the computational effort to a particular computation, and even a step in a computation, consider a sensor network of \( n \) nodes. Suppose that an observer is interested in some event or phenomenon in the northeast quadrant — comprised, say, of \( n/4 \) nodes. A protocol that would obtain the required data would propagate messages mainly through that quadrant. Much of the rest of the network would be entirely uninvolved during this process. This means that throughout the entire computation, on a large part of the state there would be no local transitions. In the part of the state representing the northeast quadrant there would be substates unchanged by transitions also, but their immutability would be temporary, with different substates remaining unchanged in different steps. Maude provides mechanisms to control the rewriting process, which can be used to adjust the computational effort by effectively eliminating parts of the state where no transitions are taken. Thus, it is possible to fulfill \textbf{Principle 1}: \textit{If a part of a state cannot be rewritten, do not try.}

The same scenario suggests another opportunity. It may be possible in some applications to represent a node that never participates in the message propagation with a smaller state than a participating one. This possibility might also apply to some of the temporarily uninvolved nodes of the northeast quadrant. This would be an actualization of \textbf{Principle 2}: \textit{If a smaller size of the state suffices, have it not larger.}

Section 6 introduces the notion of \textit{lazy configuration}, which fulfills both principles as suggested by the example. Modelling a network as a lazy configuration allows the computational effort and the size of the state to fit each step of the computation.

Some applications present a greater opportunity at a coarser grain in the computation. In some protocols, or phases of protocols, the participation of a node at some point comes to an end. Thereafter, the state of the node remains effectively unchanged. Given the purpose of the network model, it may be possible to remove the representation of this node from the state of the network without losing any information of interest. Section 7 introduces the notion of \textit{shrinkable configuration}, which under these circumstances reduces the size of the state. This notion is another concretization of \textbf{Principle 2}.

The two notions just delineated — lazy configuration and shrinkable con-
figuration — are optimizations that adapt to the evolution of a computation. A more fundamental optimization is to allow no excess in the state space of the system, which is a property of the specification. Maude makes an essential distinction in the description of nondeterministic and deterministic features of a system — the first described with rewrite rules; the second described with equations. This distinction has very practical and significant consequences. The state space is minimized by following PRINCIPLE 3: If a feature is deterministic, do not treat it as nondeterministic.

PRINCIPLE 1 and PRINCIPLE 3 are embodied in the basic structure that Section 4 proposes for the specification of complex, real-time systems. This structure defines the step mentioned above. It underlies the features introduced above, which can adapt the representation of the state at every step. It also underlies another useful enhancement in a system, considered in Section 5: its monitoring. All these enhancements are examples of features that should be deterministic. As a result of complying with these two principles, this basic structure — this step — orchestrates nondeterministic and deterministic features of a system in a modular, flexible and efficient way.

The work presented here grew out of an effort to deal with two problems in modelling a sensor network: the very large state space, and the essential need to flexibly monitor the network. These, however, are problems in modelling many other systems. All techniques presented here apply to specifications in Real-Time Maude. Section 4 proposes a framework for the specification of real-time systems that have deterministic as well as nondeterministic features. As elaborated here, the techniques of Sections 6 and 7 apply to object-oriented, real-time models of wireless networks, though the essential ideas are applicable to a much wider range of models. The technique of Section 5 applies to any object-oriented, real-time model. A model of a sensor network profits from all. Section 3 introduces a running example, and Section 8 demonstrates the usefulness and effectiveness of the modular method of developing specifications advanced here, and the high-level optimization it supports.

2 Maude

Maude [1][2] is an executable language based on rewriting logic [6], a logic of concurrent change. In rewriting logic, a concurrent system is specified by a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, where $(\Sigma, E)$ is a membership equational theory, with the signature $\Sigma$ specifying kinds, sorts, and operations; $E$ is a set of equations on $\Sigma$-terms; $R$ is a set of labelled conditional rewrite rules,

$$l : t \longrightarrow t’ \text{ if } \text{cond}$$

and $\phi$ is a function that specifies for each operation the set of argument positions on which rewriting with rules is precluded.

A rewrite theory corresponds to a system module in Maude. For system modules that satisfy some admissibility requirements [2], rewriting with rules
is performed modulo the equations of the module. Thus, only rules contribute to the size of the state space.

This Core Maude is extended, in the language Full Maude, to support a module algebra and a declarative form of object orientation. An object-oriented system is represented by a term of sort \texttt{Configuration}, which has subsorts \texttt{Object} and \texttt{Msg}. A configuration is a multiset of objects and messages, which is constructed with the juxtaposition operator. It is represented by a term of the following form:

$$M_1 \ldots M_m \langle O_1 : C_1|atts_1 \rangle \ldots \langle O_n : C_n|atts_n \rangle.$$  

The $M$s are messages, and the other terms give the states of objects named $O_i$ of class $C_i$, with the values of their attributes given in $atts_i$, for $i = 1, \ldots, n$.

The transitions a configuration may take are specified by rules of the following general form:

$$r : \quad M_1 \ldots M_m \langle O_1 : C_1|atts_1 \rangle \ldots \langle O_n : C_n|atts_n \rangle \quad \rightarrow \quad \langle O_{i_1} : C'_{i_1}|atts'_{i_1} \rangle \ldots \langle O_{i_n} : C'_{i_n}|atts'_{i_n} \rangle \quad \langle Q_1 : D_1|atts''_1 \rangle \ldots \langle Q_p : D_p|atts''_p \rangle \quad M'_1 \ldots M'_q$$  

if $\text{cond}$.

If the condition $\text{cond}$ holds, the messages $M$ are consumed; some of the original objects $O$ persist with new states and possibly new classes $C'$, and new objects $Q$ and messages $M'$ emerge. Full Maude supports the convention that a rule or equation need not mention attributes of an object that are irrelevant.

Another extension of Maude supports the specification of real-time and hybrid systems. Real-Time Maude extends Full Maude with sorts and rules to model time. A real-time specification includes a specification of a sort \texttt{Time} for the time domain, and models the elapse of time with \textit{tick rules} \cite{olveczky99}, which have the form:

$$\text{crl}\{t\} \rightarrow \{t'\} \text{ in time } u \text{ if } \text{cond}.$$  

This rule represents global transitions in which, if the condition $\text{cond}$ is satisfied, the advance of time by $u$, of sort \texttt{Time}, transforms state $t$ — the state of the entire system, which is signified by the curly brackets — into state $t'$.

In \cite{olveczky99}, Olveczky and Meseguer introduce a scheme to model time for complex systems. It describes tick rules in terms of two functions:

$$\text{crl}\{t\} \rightarrow \{\delta(t,u)\} \text{ in time } u \text{ if } u \leq \text{mte}(t) \text{ and } \text{cond}'.$$  

Function \texttt{mte} defines the maximum time elapse that will ensure time constraints; and function \texttt{delta} defines the effect of the elapse of time on the state of the system.

\begin{verbatim}
op mte : Configuration -> TimeInf [frozen (1)] .
op delta : Configuration Time -> Configuration [frozen (1)] .
\end{verbatim}
As long as the value of \( m_{te} \) is 0, time may not advance. On the other hand, when there is no constraint on the elapse of time the value of \( m_{te} \) is infinity, \text{INF}, which is included in the sort \text{TimeInf} that extends \text{Time} with \text{INF}.

Each language in this family — Core Maude, Full Maude, and Real-Time Maude — supports a spectrum of techniques for the analysis of specifications: symbolic simulation, search of the state space, and model checking. In Real-Time Maude, these techniques exploit time to limit the size of the reachable state space.

3 Example

The purpose of the work presented here was to develop modular and efficient specifications of complex systems, specifically, systems with deterministic as well as nondeterministic features. To be useful, a model of a sensor network requires that complexity. It should be as large a model as possible, since by definition a sensor network is huge; and it should flexibly monitor the model, since the strict resource limitations of its nodes will challenge the viability of many proposed protocols. Thus, the need for the kind of optimizations sketched in the introduction, and for collecting metrics. This section introduces as a running example a flooding protocol for a sensor network, which will be used to illustrate the proposed method and techniques.

A couple of reasons lead to this choice. First, the techniques proposed are opportunistic. Unlike the scenario described in the introduction, where messages propagated through a quadrant of the network, flooding propagates messages through the entire network. If opportunities found in flooding make a difference, the techniques should be useful for other protocols. Second, even though flooding is one of the simplest of protocols, it underlies many sophisticated protocols for sensor networks [4]. For example, it is used for exploration in directed diffusion [5], for issuing “sleep” and “wake up” commands, and for multihop time synchronization [3]. Thus, for this work, the example of flooding seems appropriate.

A wireless network constrains the communication to be local: a message transmitted by a node \( T \) reaches all nodes within its communication range — its neighbors — and no other. This is specified as follows:

\[
\text{class WirelessNode .} \\
\text{crl [transmit] : } T \text{ neNs } \Rightarrow T \xrightarrow{} \text{neNs} \\
\text{if } T \text{ ready-to-transmit and neNs the-neighbors-of } T . \\
\text{crl [transmit-to-none] : } T \Rightarrow T \xrightarrow{} \text{none} \\
\text{if } T \text{ ready-to-transmit and none the-neighbors-of } T .
\]

This specification reflects one basic decision: there are only objects in the configuration. This is to restrain the size of the configuration. The messages exchanged in the application are represented within the model of the transmitter, and the model of a neighbor. At this point, no restriction has been imposed on what constitutes a message; here it is entirely hidden and gen-
eral. Different subclasses of WirelessNode can define a message in different ways. Different corresponding definitions of the operation \( \Rightarrow \) would define the transmission of a message by specifying the changes it effects in the state of the transmitter and the states of its neighbors. This is a general specification that captures the wireless nature of the communication between nodes.

In pure flooding each node retransmits any message it receives. A class FloodingNode specifies nodes of a sensor network that are engaged in pure flooding.

```plaintext
class FloodingNode |
  neighbors : NzNatSet, clock : Time,
  messages : MessageQueue, heard : Nat.
*** others to support collection of metrics
subclass FloodingNode < WirelessNode.
```

This example assumes no mobility, a usual case for sensor networks. Objects modelling the nodes have indexed identifiers, so the attribute neighbors is a set of indices. A wireless node receives a message when it hears a single message. It hears a message when any neighbor transmits. If a node hears more than one message at the same time, the messages interfere or collide, and the node receives no message. The attribute heard tracks the number of messages the node hears at the current time, which is the value of the attribute clock. A single queue, the attribute messages, holds messages received, and messages to be transmitted. As a message is received, and enqueued, it is assigned a processing time. When this has elapsed, the node transmits the message. To determine whether a node has received a message, all concurrent transmissions must be taken into account. The step defined in Section 4 ensures this. Section 5 introduces the remaining attributes of FloodingNode, which are needed to support the collection of metrics.

Pure flooding in a dense, wireless network results in a high rate of collisions, and too much redundancy [7]. Random delays in retransmissions allay the first problem. A limit on the number of retransmissions, the second. A count-based scheme that incorporates these measures can be specified for CB-FloodingNode.

```plaintext
class CB-FloodingNode | received : Nat.
subclass CB-FloodingNode < FloodingNode.
```

A node that receives, retransmits. When the limit \#received-limit is reached, the node no longer will receive. So its participation in the protocol ends. Section 7 describes how to reduce the size of the state by removing these nodes.

## 4 Cycle of Rewrites

The cycle of rewrites defines a step in the computation of a real-time system. It elaborates the scheme Õlveczky and Meseguer introduced to model time for complex systems, and exploits Principle 3 — *If a feature is deterministic,*
do not treat it as nondeterministic. — to avoid excess in the state space; and

**Principle 1** — *If a part of a state cannot be rewritten, do not try.* — to avoid wasted computational effort. It is proposed as a basic framework for composing deterministic and nondeterministic features in the specification of real-time systems using Real-Time Maude.

Consider the following. In the flooding example, correctly defining “receive” requires that all concurrent transmissions be taken into account. In monitoring the performance of a network protocol; though mixing concurrent transitions and deterministic computations gathering data might be possible; a simpler, cleaner, modular approach separates the two. The state is observed after all concurrent transitions have been taken. Sections 6 and 7 describe how to transform the state so as to improve the rewriting performance, and possibly reduce the size of the state. Such optimizations may prepare the state for a set of concurrent transitions. So it is natural, convenient, and sometimes strictly necessary to allow for deterministic computations before any concurrent transition, or after all concurrent transitions at a given time in a computation. Thus, the following pattern emerges:

- deterministic computation before any transition at time $t$
- all concurrent transitions at time $t$ are taken
- deterministic computation after all transitions at time $t$
- advance time

A Maude specification is executed by rewriting with rules and equations. Maude provides mechanisms to control both kinds of rewrite. For each, the declaration of an operator may specify argument positions in which that kind of rewrite is precluded — with the attribute `frozen` to prevent rewrites with rules, and the attribute `strat` to prevent rewrites with equations. Any term with that operator at the top would have its subterms at those positions unchanged by rewrites of that kind. In the case the subterms have no redexes, the proscription has no effect on the result of rewriting the term, but it improves the rewriting performance by reducing the search space explored. Thus, it becomes possible to exploit Principle 1. In the above pattern, having isolated purely deterministic computations in the first and third step of the pattern, rewriting with rules can be prohibited in those steps in order to improve performance.

If Principle 1 can be exploited by adding some structure to computations, it can also be exploited by adding some structure to the state. Consider the problem of collecting metrics of a network protocol. An object-oriented model of the network represents each node with an object. Metrics, however, are not properties of an individual node. They are global properties. They shouldn’t be part of the state of any node, nor should they be subject to any local transition. Instead, at some points in the computation, in the third part of the above pattern, the state should be observed to gather relevant information from which to determine the metrics. Because the metrics are obtained
by a deterministic computation, Principle 1 suggests that they should be represented by a substate separate from the part of the state that is subject to local concurrent transitions. More generally, the part of the state that is exclusively subject to deterministic computation, denoted by \( s_d \), should be separate from the rest, \( s_{nd} \), which is subject to concurrent transitions. The state then takes the general form \( s_d :: s_{nd} \).

In Real-Time Maude deterministic computations are described with equations; concurrent transitions, with rewrite rules, which take no time; and the advance of time, with tick rules. So the above pattern on computations is described by the following cycle of rewrites on the state \( s_d : s_{nd} \).

- equational rewrites on \( s_d :: s_{nd} \)
- rule rewrites on \( s_{nd} \)
- equational rewrites on \( s_d :: s_{nd} \)
- tick rule

The tick rule is global, and allows no concurrent transitions.

What part of the state gets rewritten with what kind of rewrite changes through the cycle. As noted above, Maude allows control of the rewriting process with the optional attributes \texttt{frozen} and \texttt{strat} to disallow rewriting with rewrite rules or with equations, respectively, on specified argument positions. (See [2] for details.) Therefore, different constructors compose the state in different parts of the cycle. In the second step \( s_d \) is not rewritten, while \( s_{nd} \) may be rewritten with either kind of rewrite. The state is then represented as \( s_d :: s_{nd} \), where operator \( :: \) enforces that constraint.

\[
\text{op } \_::\_ : \text{Configuration Configuration} \rightarrow \text{Configuration} \\
\quad \quad \text{[ctor frozen (1) strat (2 0)] .}
\]

In the rest of the cycle, no part of the state may be rewritten with rewrite rules, but all of it may be rewritten with equations.

\[
\text{op } \_:::\_ : \text{Configuration Configuration} \rightarrow \text{Configuration} \\
\quad \quad \text{[ctor frozen (1 2)] .}
\]

In this part of the cycle the state is represented by \( s_d :: s_{nd} \).

The last step of the cycle advances time. The scheme of Ölveczky and Meseguer (page 173) describes the advance of time in terms of the function \texttt{mte}, which defines the maximum elapse of time possible that satisfies time constraints; and the function \texttt{delta}, which defines the effect of that advance on the state of the system. To minimize the number of states, the maximum allowed for the elapse of time is prescribed to be also the minimum. This is one of the strategies Real-Time Maude offers for nondeterministic models of time. Here time advances in the same eager way, but in a deterministic model of time.

\[
\text{crl } \{ t \} \longrightarrow \{ \text{delta}(t, \text{mte}(t)) \} \text{ in time } \text{mte}(t) \text{ if } \text{mte}(t) :: \text{Time} .
\]

Time advances only if there is some constraint that requires the advance. It
Rodríguez does not, when \texttt{mte} has value \texttt{INF}.

Throughout the first three steps of the cycle, time remains still. Functions \texttt{mte} and \texttt{delta} distribute over the structure of the state.

\[
\text{mte}(s_d ::: s_{nd}) = \text{mte}(s_{nd})
\]

\[
\text{delta}(s_d ::: s_{nd}) = \text{delta}(s_d) ::: \text{delta}(s_{nd})
\]

The deterministic features of the system — implemented by the first and third steps of the cycle — enhance the nondeterministic system in some way, and they preserve \texttt{mte}(s_n). Only the second step, the nondeterministic computation, may change \texttt{mte}(s_n).

The cycle of rewrites efficiently and modularly implements deterministic and nondeterministic features of a system. The first step

\[
\{s'_d ::: s'_{nd}\} = \{\text{Pre}(s_d ::: s_{nd})\}
\]

performs some deterministic transformation \texttt{Pre}. For example, \texttt{Pre} might reestablish a lazy or shrinkable configuration to reflect the current state of the computation. This first step prepares the state for the nondeterministic step of the cycle.

In a real-time system some events are associated with time. For example, in the flooding specification, as a message is received it is assigned a processing time. When this has elapsed the message is retransmitted. The function \texttt{mte} is defined so that as long as a node must retransmit at the current time the value of \texttt{mte} remains 0. More generally, \texttt{mte} can be defined so that when all the events constrained to take place at the current time have occurred, it becomes positive. In the example, in that case \texttt{mte} takes either the value of time (a natural) at which the next retransmission must occur; or if no node is to retransmit, the value \texttt{INF}. Thus, the nondeterministic step of the cycle

\[
\{s_d :: s_{nd}\} \rightarrow^* \{s'_d :: s'_{nd}\}
\]

continues until \texttt{mte}(s'_n) becomes positive.

In the following step,

\[
\{s'_d ::: s'_{nd}\} = \{\text{Post}(s_d ::: s_{nd})\}
\]

a function \texttt{Post} transforms state \texttt{s_d :: s_{nd}}. For example, \texttt{Post} might include observing \texttt{s_{nd}} to obtain metrics that update \texttt{s_d}. This step prepares the state for the tick rule, which transforms a state \texttt{s_d :: s_{nd}} as follows:

\[
\{s_d :: s_{nd}\} \rightarrow \{\text{delta}(s_d ::: s_{nd}, \text{mte}(s_{nd}))\}
\]

in time \texttt{mte}(s_{nd}) if \texttt{mte}(s_{nd}) > 0.

5 Observations

An object \texttt{o} of class \texttt{Observation} is part of \texttt{s_d}. It holds global information about \texttt{s_{nd}}, obtained by “observing” \texttt{s_{nd}} through the operation \texttt{o <- s_{nd}}. A subclass of \texttt{Observation} is used to obtain metrics for the flooding example.

\[
\text{class Observation .}
\]
\[
\text{op _<-_ : Object Configuration -> Object .}
\]
Fig. 1. FloodingNode attributes associated with metrics.

\[
eq (M : \text{Metrics} | \text{collisions} : C, \text{transmissions} : K, \\
\quad \text{have-received} : R, \text{latency} : \text{INF})
\]
\[
\leftarrow (N : \text{FloodingNode} | \text{clock} : T, \text{heard} : H, \\
\quad \text{sent-message} : B, \text{received-message-at} : T) \\
= (M : \text{Metrics} | \text{collisions} : C + \text{if} H > 1 \text{ then } H \text{ else } 0 \text{ fi}, \\
\quad \text{transmissions} : K + \text{if} B \text{ then } 1 \text{ else } 0 \text{ fi}, \text{have-received} : R). \\
\]
\[
\begin{align*}
\text{ceq} & (M : \text{Metrics} | \text{collisions} : C, \text{transmissions} : K, \text{latency} : \text{INF}) \\
& \leftarrow (N : \text{FloodingNode} | \text{clock} : T, \text{heard} : H, \\
& \quad \text{sent-message} : B, \text{received-message-at} : T') \\
& = (M : \text{Metrics} | \text{collisions} : C + \text{if} H > 1 \text{ then } H \text{ else } 0 \text{ fi}, \\
& \quad \text{transmissions} : K + \text{if} B \text{ then } 1 \text{ else } 0 \text{ fi}) \\
& \quad \text{if not} (T' \neq \text{INF} \text{ and } T = T').
\end{align*}
\]

Fig. 2. Observing a FloodingNode.

\eq
\begin{align*}
\text{eq} & V \leftarrow (\text{none}). \text{Configuration} = V. \\
\text{ceq} & V \leftarrow (C_1 \ C_2) = (V \leftarrow C_1) \leftarrow C_2 \text{ if } C_1 \neq \text{none} \text{ and } C_2 \neq \text{none}. \\
\end{align*}
\]

Note that Observation \( V \) contains some previously gathered data. The term \( V \leftarrow C \) denotes an Observation object that combines data gathered from Configuration \( C \) with that in \( V \). The configuration \( C \) remains unchanged by the \( \leftarrow \) operation.

In Maude, an object-oriented system is represented as a multiset of objects and messages. The “observation” function \( \leftarrow \) should respect this structure, satisfying the following conditions.

\[
\begin{align*}
\text{eq} & V \leftarrow (C_1 \ (C_2 \ C_3)) = V \leftarrow ((C_1 \ C_2) \ C_3) \text{ [nonexec]}. \\
\text{eq} & V \leftarrow (C_2 \ C_1) = V \leftarrow (C_1 \ C_2) \text{ [nonexec]}. \\
\end{align*}
\]

Consider the flooding example. The goal of flooding is that a message sent by a node be received by all other nodes in the network. It is important to determine whether this goal is achieved, and furthermore, to obtain metrics of interest. For this protocol they include latency, or the time it takes to deliver the message to all nodes; the cumulative number of transmissions this takes; and the number of collisions that occur while achieving this goal.

\[
\begin{align*}
\text{class} & \text{Metrics} | \\
& \quad \text{clock} : \text{Time}, \text{done} : \text{Bool}, \text{collisions} : \text{Nat}, \\
& \quad \text{transmissions} : \text{Nat}, \text{all-received} : \text{Bool}, \\
& \quad \text{have-received} : \text{Nat}, \text{latency} : \text{TimeInf}. \\
\text{subclass} & \text{Metrics} < \text{Observation}. \\
\end{align*}
\]

To collect these metrics each FloodingNode is observed, and then it is determined whether the message reached all. Figure 1 shows attributes of FloodingNode that support the collection of metrics. Figure 2 defines the observation function \( \leftarrow \) for a FloodingNode, and Figure 3 defines the operation collect, which is an example of a transformation Post (page 178) in the cycle of rewrites.
op collected : Object -> Object.
eq collected(< M : Metrics | clock : T, have-received : R, latency : INF >) = < M : Metrics | done : true, all-received : R == #nodes, latency : if R == #nodes then T else INF fi >.

op collect_ : Configuration -> Configuration.
eq collect_(< M : Metrics | done : false > Os) ::: Ns) = (collected(< M : Metrics | > <- Ns) Os) ::: Ns.

Fig. 3. Collecting metrics from FloodingNodes.

6 Lazy Configurations

A lazy configuration is a representation of $s_{nd}$ that fits the computational effort and the size of the state to the current step of the computation. Two basic ideas underlie it. First, in rewriting logic a state is represented by a term; a transition is represented by rewriting with a rule. A transition is enabled in a state if some subterm is an instance of the left-hand side of a rewrite rule. Thus, a specification and a state determine the set of all subterms on which transitions are enabled. As discussed in Section 4, precluding rewriting in other parts of the state improves the rewriting performance. Second, for some applications, some subterms on which no transition is enabled might be replaced by smaller terms. This would be possible if there were a bijection between the smaller and the larger terms. Then the larger terms would be recovered when needed. A lazy configuration exploits both ideas. In particular, these ideas are elaborated assuming that all transitions are enabled in subconfigurations with an object and all its “neighbors”, as would be the case in a message-passing, object-oriented model of a wireless sensor network.

Consider the flooding example. The transmit rule is enabled on a subconfiguration composed of a node ready to transmit, and all its neighbors. So any node that is neither ready to transmit, nor a neighbor of one that is, is in the part of the state that cannot be rewritten. This example suggests that the state might be represented by a term $nen \#|en$, where $nen$ is a subconfiguration on which no transition is enabled, and $en$ is the minimal subconfiguration on which all the transitions that are enabled in $nen$ are also enabled. Note that the reason for this split is to effectively remove the $nen$ substate when concurrent transitions are performed. Since no transitions are enabled in this substate, there is no need to allow equational rewrites that define the structure of this substate. Neither should there be any deterministic computation, since in $s_{nd}$ any such computation should accompany a local transition. The part of the state subject to exclusive deterministic computation is $sd$. Thus, the operator $\_\#|\_$ disallows any kind of rewrite in the first argument, and allows both kinds of rewrite in the second argument.

Now consider two FloodingNodes in $nen$: one with a nonempty messages queue, with messages scheduled to be transmitted later; and one with an empty messages queue. Consider also that the heard and sent-messages attributes represent transient values, which for a node in $nen$ are the default
values 0 and false. So the second node, which has a substate consisting of
default values, could be represented by the nondefault values. A superclass
DormantFloodingNode of FloodingNode defines the attributes corresponding
to these values. Should that node later be part of a configuration that enables
a transmission — it would have to be as a neighbor of a transmitter — it
would revert to the FloodingNode representation, getting the same value of
clock that other FloodingNodes would have.

More generally, analogous distinctions may apply to specifications in which
the state is a term of sort Configuration (page 173; [2]). A specification and
a Configuration state/term determine a minimal part of the state that pre-
serves all enabled transitions. For Configuration states this part is a single
term since the constructor for nonempty Configurations is associative and
commutative. The subconfiguration on which no transition is enabled might
itself be partitioned into two, if the application justifies a distinction. So a
new constructor for Configuration partitions the state into three subconfig-
urations.

\[
\text{op } \_\#|\_|\_ : \text{Configuration Configuration Configuration} \\
\rightarrow \text{Configuration [ctor strat (2 0) frozen (1 3)]} .
\]

The first argument is a configuration on which no transition is enabled, but
it “preenables” some. The second is a configuration on which transitions are
enabled. The third is a configuration on which no transition is either enabled
or preenabled. These three configurations might be described, respectively, as
“waiting”, “active”, and “dormant”.

It is appropriate and convenient to partition the state as just described
before the cycle takes the concurrent transitions. These, in general, will not
preserve the intent of the partitioning. It will have to be reestablished correctly
before the next concurrent transitions.

This transformation of the state begins with the tick rule.

\[
\text{eq mte(Cw #| Ca |# Cd) = mte(Cw Ca)} .
\]
\[
\text{eq delta(Cw #| Ca |# Cd, T) = none #| delta(deactivate(Cw Ca), T) |# Cd} .
\]

The deactivate operation identifies the part of the configuration that would
be likely to become dormant. For the rest of the configuration, deactivate
acts as the identity function. In the flooding example, deactivate identifies
nodes with empty messages queues as the candidates. When the tick rule is
taken, the waiting times of the messages in the queue are updated, and a new
set of enabled transitions is determined by nodes with messages ready to be
transmitted. At this point, in the first step of a new cycle, the configuration
is repartitioned. Whether a node remains dormant, or is “awakened” depends
on whether one of its neighbors is ready to transmit. More generally, the Pre
function completes the repartition according to the interpretation of waiting,
active and dormant.

In a message-passing model, processes interact by sending messages. For
Rodríguez

a model of a wireless network, the ones receiving the messages are neighbors. Thus, in an object-oriented, message-passing model of a wireless network, whenever an object/node is enabled to send a message, the subconfiguration of that node with its neighbors enables a transition. In the partitioning scheme considered here, the sending node is part of the active configuration. A configuration that includes all sending nodes; and for each, all its neighbors is defined to satisfy the predicate \textit{is-fully-awake} — the name hints that some candidates to become dormant become part of the active configuration to enable some transition.

Given a configuration \( C \), the partitioning transformation obtains the lazy configuration \( C_w \#| C_a |\# C_d \), where \( C = C_w C_a C_d \), and \( C_a \) is the minimal subconfiguration of \( C \) that satisfies \textit{is-fully-awake}. The rest of \( C \) is partitioned into \( C_w \), whose nodes satisfy \textit{is-waiting} — the node will become enabled to send later; and \( C_d \), in which no node \textit{is-waiting}. Furthermore, nodes in \( C_w C_a \) are of a subclass of that for nodes in \( C_d \). This partitioning transformation preserves the \textit{neighbors} attribute of all nodes. Predicates \textit{the-neighbors-of}, \textit{is-fully-awake}, and \textit{is-waiting}; as well as functions \texttt{deactivate} and \texttt{activate} — which maps a smaller representation to a larger one; are interpreted appropriately for each different network or protocol model.

7 Shrinkable Configurations

For some applications a computation may reach a state after which the rest of the computation preserves some substate. For example, in a network model, after a node fails, its state will not change. For some protocols the participation of a node ends at some point. When \texttt{CB-FloodingNodes} (page 175) have retransmitted the maximum allowed number of times, their participation in the flooding algorithm ceases. Thus, for some applications it may be justified to remove part of the state at some point.

This transformation of the state begins with the tick rule. A \texttt{vanish} function identifies the part of the state to be removed. For a network model, whenever a node vanishes, the \texttt{Pre} transformation of the next cycle must remove all edges to that node.

When a substate of a lazy configuration vanishes the advance of time is defined as follows:

\[
\begin{align*}
\text{eq} & \quad \delta( (O C_w \#| O C_a |\# O C_d), T ) \\
& = \text{none} \#| \delta(\text{deactivate} (\text{vanish}(O C_w O C_a)), T) |\# O C_d . \\
\text{eq} & \quad \text{mte}(\text{vanish}(O)) = \text{INF} . \\
\text{eq} & \quad \delta(\text{deactivate}(\text{vanish}(O)), T) = \text{vanish}(O) .
\end{align*}
\]

8 Experiments

The specification of flooding for \texttt{CB-FloodingNodes} (page 175) has as parameters the number of nodes in the network, \#nodes; and the maximum number
of times a node may retransmit, \#received-limit. A few configurations were studied to reveal whether the message would reach all nodes, what value of \#received-limit this would require, and what the associated metrics were. For the smallest configurations the three different representations for $s_{nd}$ were used: ordinary configurations, lazy configurations and shrinkable configurations. For larger ones, only the shrinkable configurations were used. To help determine the efficacy of the techniques, when lazy or shrinkable configurations were used, the \texttt{Metrics} objects tracked how many nodes were active at each tick. A \texttt{Metrics} object was generated at each tick, reporting the metrics until then. This was the most informative part of the state.

The timed rewrite (\texttt{trew}) and untimed model checking commands of Real-Time Maude were used \cite{8}. The timed rewriting of some initial configuration may have one of three results: the \texttt{Metrics} objects reveal that the message reached all nodes, what the associated metrics (and any meta-metrics) were; the \texttt{Metrics} objects reveal that the message has not reached all, but an examination of the node configuration shows it may still be possible; and, lastly, \texttt{Metrics} objects and node configuration show the message will not reach all. In the second case the computation is resumed; the final state of the previous try becomes the configuration to be rewritten for some specified time. In the last case the limit of retransmissions is incremented.

If the timed rewriting showed the message reached all, the latency $l$, and associated metrics — transmissions $t$ and collisions $c$ — the following untimed model checking command was used to check that all computations with initial state \texttt{init} satisfy the linear temporal logic formula $\langle\langle all-received-by \ l \ w \ t \ \text{transmissions-}c \ \text{collisions} \rangle \rangle$ \cite{10} \cite{8}.

\begin{verbatim}
(mc init
 |=#u <> all-received-by l w t transmissions-\& c collisions .)
\end{verbatim}

where

\begin{verbatim}
op all-received-by_w_transmissions-\&_collisions : Time Nat Nat -> Prop [ctor].
eq \{(OC < mets : Metrics | done : true, clock : T',latency : T',
transmissions : M', collisions : N' > ) ::: C \}
\= all-received-by T w M transmissions-\& N collisions
= T' <= T and-then (M' <= M and-then N' <= N) .
\end{verbatim}

Counterexamples provide new triples to try.

The following tables show some results. The table on the left gives the results of four simulations: the latency $l$, and number of transmissions $t$ and collisions $c$, for configurations with $n$ nodes, and retransmission limit $rL$. The table on the right shows the effectiveness of the techniques: the cpu time for
the simulation, and average number of active nodes.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$n$</th>
<th>$rL$</th>
<th>$l$</th>
<th>$t$</th>
<th>$c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>20</td>
<td>2</td>
<td>32</td>
<td>26</td>
<td>10</td>
</tr>
<tr>
<td>$c_2$</td>
<td>30</td>
<td>2</td>
<td>43</td>
<td>45</td>
<td>14</td>
</tr>
<tr>
<td>$c_3$</td>
<td>30</td>
<td>3</td>
<td>43</td>
<td>61</td>
<td>22</td>
</tr>
<tr>
<td>$c_4$</td>
<td>40</td>
<td>3</td>
<td>53</td>
<td>97</td>
<td>32</td>
</tr>
</tbody>
</table>

The table below gives some model-checking results.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$l$</th>
<th>$t$</th>
<th>$c$</th>
<th>result</th>
<th>$k$</th>
<th>type</th>
<th>cpu time</th>
<th>% active</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>1751 sec</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>134 sec</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c_2$</td>
<td>62 sec</td>
<td>43</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>127 sec</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9 Conclusion

A modular approach exploits some of the features and mechanisms Maude offers to produce specifications for complex, concurrent systems that are economical in three aspects: state space, size of state, and rewriting performance. The cycle of rewrites supports modularity and efficiency, and is applicable to any specification of a complex, real-time concurrent systems. It serves as a framework for composing concurrent and deterministic features of a system. An example of a deterministic feature is a flexible mechanism for monitoring the performance of a model, which is essential in the study of sensor networks. Another deterministic feature is the versatile representation of the state, which is implemented through lazy and shrinkable configurations. These fit the size of the state, and the rewriting effort to the state of the computation. They are specified by high-level, modular specifications in terms of predicates and functions that must be interpreted for each application. Though the ideas behind them may be applied to a variety of concurrent systems, as they are elaborated here, they are most appropriate for message-passing models of wireless networks.

Acknowledgement

My appreciation and thanks to the referees and Program Chair Carolyn Talcott for their helpful suggestions.
References


Solving Sudoku Puzzles with Rewriting Rules

Gustavo Santos-García

Universidad de Salamanca

Miguel Palomino

Dpto. de Sistemas Informáticos y Programación, Universidad Complutense

Abstract

The aim of the sudoku puzzle (also known as number place in the United States) is to enter a numeral from 1 through 9 in each cell of a grid, most frequently a 9 × 9 grid made up of 3 × 3 subgrids, starting with various numerals given in some of the cells (the “givens”). Each row, column, and region must contain only one instance of each numeral. In this paper we show how a sudoku puzzle can be solved with rewriting rules using Maude. Three processes (scanning, marking up, and analysis) are the classical techniques for solving sudokus. Elimination is the main strategy that we have employed. The strategy what-if and several contingencies are also implemented.

Key words: Rewriting logic, Maude, sudoku, puzzle.

1 Introduction

Rewriting logic [12] is a logic of concurrent change that can naturally deal with states and with highly nondeterministic concurrent computations. It has good properties as a flexible and general semantic framework for giving semantics to a wide range of languages and models of concurrency. Moreover, it allows user-definable syntax with complete freedom to choose the operators and structural properties appropriate for each problem.

The naturality of rewriting logic and of its implementation, Maude [9], for modeling and experimenting with mathematical problems, has been illustrated in a number of works. Our goal with this paper is to further contribute to that pool but in a rather more recreational context, along the lines in [14].
For that, we present a case study of how to use Maude to execute and solve a popular kind of puzzles, namely sudokus. In Maude it is very easy to support objects and distributed object interactions in a completely declarative style with rewrite rules. We describe how sudokus can be represented in an object-oriented way and give formal rules that transform an initial sudoku into one in solved form, and how this representation can be straightforwardly mapped to Maude. The application of these rules, though always leading to a solution, can do so in many different ways; some give rise to combinatorial explosions and should thus be avoided. To handle this matter we have employed strategies to guide Maude’s rewrite engine, which constitute an example of the use of the recently developed strategy language for Maude [13].

2 Sudokus

A (standard) sudoku is a $9 \times 9$ grid made up of $3 \times 3$ subgrids, also called “regions.” Initially, some cells contain “given” numbers: the goal is to fill in the empty cells, one number in each, so that each column, row, and grid contains the numbers 1 through 9 exactly once (see an example in Figure 1). Originally called number place, the first such puzzle was created by Howard Garnes, a freelance puzzle constructor, in 1979 [1]. The puzzle was first published in New York by the specialist puzzle publisher Dell Magazines in its magazine Math Puzzles and Logic Problems. The puzzle was introduced in Japan by the publishing company Nikoli in the paper Monthly Nikoli in April 1984 as “Suji wa dokushin ni kagiru” [2], which can be translated as “the numbers must occur only once” or “the numbers must be single.” At a later date, the name was abbreviated to sudoku, pronounced sue-do-koo; su = number, doku = single; it is a common practice in Japanese to take only the first kanji of compound words to form a shorter version. In 1986, Nikoli introduced two innovations which guaranteed the popularity of the puzzle: the number of givens was restricted to no more than 30 and puzzles became symmetrical (meaning the givens were distributed in rotationally symmetric cells). It is now published in mainstream Japanese periodicals, such as the Asahi Shimbun. The surge in popularity of sudoku in 2005 has led the world media to dub it as “the Rubik’s cube of the 21st century” [3].

<table>
<thead>
<tr>
<th>3 7 2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 9 5</td>
<td>8</td>
</tr>
<tr>
<td>4 9 1</td>
<td>5 2</td>
</tr>
<tr>
<td>5</td>
<td>6 1</td>
</tr>
<tr>
<td>8 4</td>
<td>2 9</td>
</tr>
<tr>
<td>7 6</td>
<td>4</td>
</tr>
<tr>
<td>2 3 5</td>
<td>1 7</td>
</tr>
<tr>
<td>1 2 3 9</td>
<td></td>
</tr>
<tr>
<td>5 8 6 4</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. A sudoku puzzle (©The Times, 2005, num. 295) and its solution.
The numerals in sudoku puzzles are used for convenience. It works just as fine if the numbers are replaced with letters, shapes, colours or some other symbols. Sudoku is not a mathematical or arithmetical puzzle; arithmetic relationships between numerals are absolutely irrelevant. Indeed, Penny Press use letters in their version called scramblets; Knight Features Syndicate also use letters in their Sudoku Word (see Figure 2).

![Sudoku Word puzzle and its solution](image)

Fig. 2. A Sudoku Word puzzle (©Knight Features Syndicate, 2005) and its solution. Complete the grid so that every row, column and 3×3 box contains a different letter: A, B, E, M, N, O, R, S & U. One row or column contains a 7-letter word. What is it?

The Guardian calls these godoku and describes them as “devilish”; others name them wordoku or sudoku word. The required letters are given beneath the puzzle: once arranged they spell out a topical word between the top left and bottom right corners. This adds an extra dimension to sudokus as it may be possible to guess what the word is.

The appealing nature of the puzzle lies in the fact that the completion rules are simple, yet the reasoning required to reach the solution may be difficult. It also rises interesting questions, some of which are open: to measure the difficulty of a puzzle, to construct new sudokus, to establish the number of possible sudokus, to determine the number of possible puzzles with a single solution, to optimize the search of the solution and so on.

2.1 An NP Problem

Constraint programming has reached the masses. When solving their daily sudoku puzzle, thousands of newspaper readers apply classic propagation schemes in constraint programming like X-wing and swordfish [4]—patterns that cover several rows and columns, seeking a candidate number that can be removed from other lists in the corresponding columns and rows—to find a solution.

The general problem of solving sudoku puzzles on \( n^2 \times n^2 \) boards of \( n \times n \) blocks is known to be NP-complete [15]. This gives some vague indication of why sudokus are hard to solve, but for boards of finite size the problem is also finite and can be solved by a deterministic finite automaton that knows the entire search tree. However, for a non-trivial starting board the search tree is
very large and so this method is not feasible.

A valid sudoku solution is also a Latin square. A Latin square is an $n \times n$ table filled using different symbols in such a way that each symbol occurs exactly once in each row and exactly once in each column. Sudoku imposes the additional regional constraint; nonetheless, the number of valid sudoku solution grids for the standard $9 \times 9$ grid is $6,670,903,752,021,072,936,960$. This number is equal to $9! \times 72^2 \times 27 \times 7,704,267,971$, the last factor of which is prime. The result was derived through logic and brute force computation; the methodology for this analysis can be found at [10].

3 Definitions and Notation

3.1 Solution methods

As described in [11], three processes (scanning, marking up, and analysis) are the classical techniques for solving sudokus. The application of these processes is not deterministic so that the real performance of a sudoku solver depends on the approaches in their combination: if one applies these processes blindly, a combinatorial blowup may happen. Ideally, then, one needs to find a combination of these techniques which finds a solution in an efficient manner.

Scanning

It is performed at the outset and periodically throughout the procedure. It may have to be performed several times between analysis periods. Scanning consists of three basic methods that can be used alternatively: cross-hatching, counting, and “looking for contingencies.”

Cross-hatching is the scanning of rows to identify which line in a particular grid may contain a certain number by a process of elimination. This process is then repeated with the columns. It is important to perform this process systematically, checking all of the digits 1–9. For fastest results, the numbers are considered in order of their frequency.

The counting of the occurrences of the numbers 1 to 9 in grids, rows, and columns tries to identify missing numbers. Counting based upon the last number discovered may speed up the search.

Advanced solvers look for “contingencies” while scanning, that is, narrowing a number’s location within a row, column, or grid to two or three cells. When those cells all lie within the same row (or column) and grid, they can be used for elimination purposes during cross-hatching and counting. More difficult sudokus, by definition, cannot be solved only by basic scanning alone and require the detection of contingencies.

Marking up

Scanning comes to a halt when no more numbers can be discovered. At this point, it is useful to mark candidate numbers in the blank cells. Subscripts
are a popular notation: the candidate numbers are written as subscripts in
the cells. It is usually difficult to use this method in the newspaper because
the cell space is small. There is a second notation with dots (a dot in the top
left corner represents a 1 and a dot in the bottom right corner represents a 9).
This notation has the advantage that it can be used on the original sudoku.

Analysis
There are two main analysis approaches: elimination and what-if. The
elimination of possible numbers from a cell allows to leave the only possible
choice. There are a number of elimination tactics. One of the most common
is unmatched candidate deletion: a collection of \( n \) cells with identical possible
numbers is said to be matched if the quantity of candidate numbers in each is
equal to \( n \). Then the numbers appearing as candidates elsewhere in the same
row, column, or grid in unmatched cells can be deleted. For instance, if there
are three cells in the same row with \( \{2, 7, 8\} \) as their set of possible numbers,
then 2, 7, and 8 can be discarded as possible numbers from the remaining cells
in that row.

Using the what-if approach (which is called reductio ad absurdum in \([11]\))
a cell with only two candidate numbers is selected and a guess is made. The
procedure then continues with the resulting sudoku and, if no solution is found,
the alternative number is tried.

3.2 Rules for solving sudokus
We will represent sudokus as a “soup” (formally, a set) of objects, where each
object corresponds to a cell.

Definition 3.1 A sudoku \( S \) of order \( n \) can be represented as a set of objects
\( C_{ij}, 1 \leq i, j \leq n \), one for each cell at row \( i \) and column \( j \), where each object
has the following attributes:

- \( G_{ij} = \sqrt{n} \cdot \text{int}((i-1)/\sqrt{n}) + \text{int}((j-1)/\sqrt{n}) + 1 \) is the grid to which the
cell belongs.

- \( P_{ij} \) is the set of possible numbers that may occur in this cell. The sudoku
will be solved when this attribute is a unitary set for all cells; if \( P \) becomes
empty for some cell, the sudoku does not have a solution.

- \( N_{ij} \) is the number of elements in \( P_{ij} \).

Note that it is not necessary to consider \( i \) and \( j \) as additional attributes since
they are part of a cell’s name. The standard sudoku is a \( 9 \times 9 \) grid, that is, a
sudoku of order \( n = 9 \).

Initially \( P_{ij} \) is equal to \( \{1, \ldots, n\} \) except when \( P_{ij} \) is the cell of a given \( g_{ij} \),
in which case \( P_{ij} = \{g_{ij}\} \). Attributes \( G_{ij} \) and \( N_{ij} \) are stored explicitly but are
computed from the rest of the elements of an object. We will apply several
reduction rules to cells to obtain a solution for a sudoku. The goal is to reach
a unique number in $P_{ij}$ for all the cells; if $P_{ij}$ becomes empty for some cell, then the sudoku has no solution.

During the procedure, and due to the what-if rule, a sudoku may be split into two and to distinguish one from the other we will enclose the corresponding objects in brackets. Thus, for example, the representation of two sudokus of order 9 will look like

$$[C_{11}C_{12} \ldots C_{99}] \ [\overline{C}_{11}\overline{C}_{12} \ldots \overline{C}_{99}].$$

Among the different processes (scanning, marking up, and analysis) for solving sudokus, the main strategy that we have considered is that of analysis for elimination. It is defined by Rule 1 below, which removes an element from the set of possible numbers in a cell. We have complemented it with Rules 2 and 3, of second and third order simplification, which respectively consider two and three elements. Several contingencies corresponding to the scanning process are covered by Rules 4 to 6. The strategy what-if is implemented with the sudoku split rules (Rules 7 and 8).

We next present in detail each rule. They are to be understood as local transition rules in a possibly concurrent system, that can concurrently be applied to different fragments of the soup: the cells in the upper part of each sequent become the ones below while the rest remain unchanged (since the attributes $N_{ij}$ and $G_{ij}$ are computed from $P_{ij}$, we do not explicitly mention their new values). We use $(\cdot)^c$ to denote the complement of a set.

**Rule 1 (First order Simplification Rule)** If only one number is possible in a cell, then we remove this number from the set of possible numbers in all the other cells in the same row, column or grid. Symbolically,

$$\frac{C_{ij}}{C_{ij}^c} \frac{C_{i'j'}}{C_{i'j'}^c} \frac{C_{i''j''}}{C_{i''j''}^c}$$

where:

(i) $i = i'$ or $j = j'$ or $G_{ij} = G_{i'j'}$,

(ii) $P_{ij} = \{p\} \subseteq P_{i'j'}^c$,

and the attribute $P_{i'j'}^c$ of $C_{i'j'}^c$ is equal to $P_{i'j'}^c - \{p\}$.

**Rule 2 (Second order Simplification Rule)** If two cells in the same row (column or grid) have the same set of possible numbers and its cardinality is 2, then those numbers can be removed from the sets of possible numbers of every other cell in the same row (column or grid):

$$\frac{C_{ij}}{C_{ij}^c} \frac{C_{i'j'}}{C_{i'j'}^c} \frac{C_{i''j''}}{C_{i''j''}^c}$$

where

(i) $i = i'$ or $j = j'$ or $G_{ij} = G_{i'j'} = G_{i''j''}$,

(ii) $N_{ij} = N_{i'j'} = 2$,

(iii) $P_{ij} = P_{i'j'}$; $P_{ij} \cap P_{i''j''} \neq \emptyset$. 

191
and the attribute $\overline{P}_{i''j''}$ of $\overline{C}_{i''j''}$ is equal to $P_{i''j''} - P_{ij}$.

**Rule 3 (Third order Simplification Rule)** If three cells in the same row (column or grid) have the same set of possible numbers and its cardinality is 3, then those numbers can be removed from the sets of possible numbers of every other cell in the same row (column or grid). Actually, the rule can be slightly generalized by allowing the cardinality of the sets to be 2 for some of the cells but one:

$$
\frac{C_{ij} C_{i'j'} C_{i''j''} C_{i'''j'''}}{C_{ij} C_{i'j'} C_{i''j''} \overline{C}_{i'''j'''}}
$$

where

(i) $i = i' = i'' = i'''$ or $j = j' = j'' = j'''$ or $G_{ij} = G_{i'j'} = G_{i''j''} = G_{i'''j'''}$,

(ii) $N_{ij} = 3; \ 2 \leq N_{i'j'}; N_{i''j''} \leq 3,$

(iii) $P_{i'j'}, P_{i''j''} \subseteq P_{ij}; \ P_{ij} \cap P_{i'''j'''} \neq \emptyset$,

and the attribute $\overline{P}_{i'''j'''}$ of $\overline{C}_{i'''j'''}$ is equal to $P_{i'''j'''} - P_{ij}$.

Further simplification rules could be added, but they are not needed and actually matching is much more expensive for them.

**Rule 4 (Only One Number Rule)** When a number is not possible in any cell of a row (column or grid) but one, and the cardinality of the set of possible numbers for this cell is greater than one, then this set can become a singleton set containing that number. Symbolically,

$$
\frac{C_{i_1j_1} \{C_{i_kj_k}\}_{2 \leq k \leq n}}{\overline{C}_{i_1j_1} \{C_{i_kj_k}\}_{2 \leq k \leq n}}
$$

where:

(i) There exists a number $N, 1 \leq N \leq n$, such that $N = i_k$ or $N = j_k$ or $N = G_{i_kj_k}$ for all $1 \leq k \leq n$,

(ii) $N_{i_1j_1} > 1,$

(iii) There exists a number $p, 1 \leq p \leq n$, such that $p \in P_{i_1j_1} \cap (\bigcup_{2 \leq l \leq n} P_{i_lj_l})^c$, and the attribute $\overline{P}_{i_1j_1}$ of $\overline{C}_{i_1j_1}$ is equal to $\{p\}$.

**Rule 5 (Only Two Numbers Rule)** When two numbers $p_1$ and $p_2$ are not possible in any cell of a row (column or grid) but two, and the sets of possible numbers for these cells have cardinality greater than two, then these sets can become $\{p_1, p_2\}$.

$$
\frac{C_{i_1j_1} C_{i_2j_2} \{C_{i_kj_k}\}_{3 \leq k \leq n}}{\overline{C}_{i_1j_1} \overline{C}_{i_2j_2} \{C_{i_kj_k}\}_{3 \leq k \leq n}}
$$

where:

(i) There exists a number $N, 1 \leq N \leq n$, such that $N = i_k$ or $N = j_k$ or $N = G_{i_kj_k}$ for all $1 \leq k \leq n$,

(ii) $N_{i_1j_1}, N_{i_2j_2} > 2,$
There exists two numbers $p_1, p_2$, $1 \leq p_1, p_2 \leq n$, such that $p_1, p_2 \in P_{i_1j_1} \cap P_{i_2j_2} \cap \left( \bigcup_{3 \leq l \leq n} P_{i_lj_l} \right)^c$, and the attributes $\overline{P}_{i_1j_1}$ and $\overline{P}_{i_2j_2}$ of $\overline{C}_{i_1j_1}$ are both equal to \{p_1, p_2\}.

**Rule 6 (Twin Rule)** If, in a given grid, a number is only possible in one row (or column), then that number can be removed from the set of possible numbers in all the cells in that same row (or column) but different grid.

\[
\frac{C_{i_0j_0} \{C_{i_kj_k}\}_{1 \leq k \leq n}}{\overline{C}_{i_0j_0} \{C_{i_kj_k}\}_{1 \leq k \leq n}}
\]

where:

(i) There exists a number $N$, $1 \leq N \leq n$, such that $N = G_{i_kj_k}$, for all $1 \leq k \leq n$,

(ii) $i_0 \neq i_k$ and $j_0 \neq j_k$ for all $4 \leq k \leq n$,

(iii) $i_0 = i_1$ or $j_0 = j_1$,

(iv) There exists a number $p$, $1 \leq p \leq n$, such that $p \in P_{i_0j_0} \cap P_{i_1j_1} \cap \left( \bigcup_{4 \leq l \leq n} P_{i_lj_l} \right)^c$, and the attribute $\overline{P}_{i_0j_0}$ of the representation of $\overline{C}_{i_0j_0}$ is equal to $P_{i_0j_0} - \{p\}$.

**Rule 7 (General Sudoku Split Rule)** This rule splits a sudoku when none of the other rules can be applied. We select a cell with a minimum number (greater than 1) of possible numbers. Then a sudoku is created with the first possible number and another one with the remaining possible numbers:

\[
\begin{array}{c}
C_{ij} \{C_{kl}\}_{k \neq i, l \neq j} \\
\overline{C}_{ij} \{C_{kl}\}_{k \neq i, l \neq j}
\end{array}
\]

if these conditions are fulfilled:

(i) $N_{ij} \geq 2$ and $N_{ij}$ is minimal (but greater than 1),

(ii) $p \in P_{ij}$,

and the attribute $\overline{P}_{ij}$ of $\overline{C}_{ij}$ is equal to $\{p\}$ and the attribute $\overline{P}_{ij}$ of $\overline{C}_{ij}$ is $P_{ij} - \{p\}$.

**Rule 8 (Sudoku Split Rule)** This rule is the particular case of the previous one when the number of possible numbers in a cell is equal to 2.

The correctness of each rule is immediate by their definition. Its application, however, does not lead to a confluent system: in case a sudoku has several solutions, the one that is reached can vary according to the order of execution of the rules; when a sudoku has a unique solution, this is obtained. The system is terminating: even though Rule 7 introduces a new sudoku, every rule decreases the cardinal of the set of possible numbers in one of the cells.

193
4 Rewriting logic and Maude

Maude [8,9] is a high performance language and system supporting both equational and rewriting logic computation for a wide range of applications. The key novelty of Maude is that besides efficiently supporting equational computation and algebraic specification it also supports rewriting logic computation.

A rewrite theory is a four-tuple $T = (\Omega, E, L, R)$, where $(\Omega, E)$ is a theory in an equational logic, $L$ is a set of labels for the rules, and $R$ is the set of labeled rewrite rules axiomatizing the local state transitions of the system. Some of the rules in $R$ may be conditional. Mathematically, a rewrite rule has the form $l : t \rightarrow t'$ if $C$, with $t$, $t'$ terms of the same kind which may contain variables. Intuitively, a rule describes a local concurrent transition in a system: anywhere where a substitution instance $\sigma(t)$ of $t$ is found, a local transition of that state fragment to the new local state $\sigma(t')$ can take place.

We can regard an equational theory as the special case of a rewrite theory in which the sets of labels $L$ and rules $R$ are both empty; in this way, equational logic appears naturally as a sublanguage of rewriting logic.

Maude is a language whose modules are theories in rewriting logic. The most general Maude modules are called system modules and are written as $\text{mod } T \text{ endm}$, with $T$ the rewrite theory in question expressed with a syntax quite close to the corresponding mathematical notation. The equations $E$ in the equational theory $(\Omega, E)$ underlying the rewrite theory $T = (\Omega, E, L, R)$ are presented as a union $E = A \cup E'$, with $A$ a set of equational axioms introduced as attributes of certain operators in the signature $\Omega$—for example, an operator $+$ can be declared associative and commutative with keywords $\text{assoc}$ and $\text{comm}$—and where $E'$ is a set of equations that are assumed to be Church-Rosser and terminating modulo the axioms $A$. Maude supports rewriting modulo different combinations of such equational attributes: operators can be declared associative, commutative, with identity, and idempotent.

4.1 Specifying sudokus in Maude

Our sudoku system module is called SUDOKU.

\begin{verbatim}
mod SUDOKU is
  pr NAT . pr QID . pr CONVERSION .

  We first import into it some predefined modules that define the natural numbers, quoted identifiers, and string and number conversion. Maude provides useful support for modularity by allowing the definition of module hierarchies; a module can import other Maude modules as submodules in different modes, in this case with the keyword protecting (which can be abbreviated to pr):

  pr NAT . pr QID . pr CONVERSION .

  The framework we have proposed for solving sudokus constitutes an example of an object-based system [9, Chapter 8]. Maude supports the specification
\end{verbatim}
of such systems in a simple and direct way through a predefined module called CONFIGURATION. However, instead of using this module, and since we want our output to be formatted in a tabular manner and to use different colours, we will explicitly declare all the operators needed for the specification of such systems; we will use them to introduce Maude syntax.

As described in the previous section, the cells in a sudoku correspond to objects and we will use messages to show the current status of the puzzle: SearchingSolution, NoSolution, FinalSolution. A “soup” of such objects and messages is called a configuration and will be used to represent sudokus. Thus, we declare sorts for objects and messages, which are subsorts of configurations, as well as for object and class identifiers (called Oid and Cid).

```plaintext
sorts Oid Cid Object Msg Configuration .
subsort Object Msg < Configuration .
```

To create objects we introduce, with the keyword op, an operator `<_:|_>` that takes an object identifier, a class identifier, and a set of attributes as arguments.

```plaintext
op `<_:|_>` : Oid Cid AttributeSet -> Object
    [ctor object format (n r! o g o tm! ot d)] .
```

The underbars allow the specification of mixfix syntax and are placeholders where the arguments should be written. (The commands enclosed in the brackets simply specify the format of the output [9].)

Configurations are declared with an operator with empty syntax (`__`) which is associative, commutative, and has an identity element.

```plaintext
op none : -> Configuration .
op `__` : Configuration Configuration -> Configuration
    [ctor config assoc comm id: none] .
```

The current state of the solving procedure will be represented as a term of sort Sudoku, which consists of a set of sudokus each one enclosed in double angles:

```plaintext
sort Sudoku .
op none : -> Sudoku .
op `<>` : Configuration -> Sudoku .
op `__` : Sudoku Sudoku -> Sudoku [ctor config assoc comm id: none] .
```

Attributes of objects belong to a new sort Attribute, which is a subsort of AttributeSet built with `__,__`.

```plaintext
sorts Attribute AttributeSet .
subsort Attribute < AttributeSet .
op none : -> AttributeSet .
op `__,__` : AttributeSet AttributeSet -> AttributeSet
    [ctor assoc comm id: none format (m! o sm! o)] .
```

For our sudoku specification we need to declare the three attributes asso-
associated to each cell: the grid to which it belongs, the set of possible numbers that may be placed in it, and its cardinality; the column and row will be given by the cell identifier. Attributes are declared as operators that return a term of sort \texttt{Attribute}.

\begin{verbatim}
  op grd :_ : Nat -> Attribute .
  op pss :_ : Set -> Attribute .
  op num :_ : Nat -> Attribute .
  op id : Nat Nat -> Oid . --- row and column
\end{verbatim}

All these declarations specify the core syntax needed to represent sudokus and our solving procedure in Maude. Now, their behaviour is specified by means of equations and rules. Equations are declared using the keyword \texttt{eq} and variables with the keyword \texttt{var}. For example, the operator that returns the grid associated to the cell at column \texttt{C} and row \texttt{R} can be specified as

\begin{verbatim}
  op grd : Nat Nat -> Nat .
  vars R C : Nat .
  eq grd(R, C) = (sd(R, 1) quo 3) * 3 + (sd(C, 1) quo 3) + 1 .
\end{verbatim}

where \texttt{sd} and \texttt{quo} the integer quotient are respectively the operators for symmetric difference and integer quotient.

An initial board for a sudoku can be specified as a constant term \texttt{sudoku}. To avoid the cumbersome task of explicitly writing the complete representation of a sudoku (9 × 9 cells with their corresponding rows, columns, numbers, grids, ..., for a sudoku of order 9), we use several auxiliary operators that will transform a much closer representation of a sudoku into the object-based format. For example, for the sudoku in Figure 1:

\begin{verbatim}
  op sudoku : -> Sudoku .
  eq sudoku =
    << msg('SearchingSolution)
    fill(1, 1,
      ( 0 ; 0 ; 3 ; 7 ; 2 ; 0 ; 1 ; 0 ; 0 ;
      0 ; 0 ; 6 ; 9 ; 0 ; 5 ; 8 ; 0 ; 0 ;
      4 ; 9 ; 0 ; 0 ; 1 ; 0 ; 0 ; 5 ; 2 ;
      0 ; 5 ; 0 ; 0 ; 0 ; 0 ; 0 ; 6 ; 1 ;
      8 ; 0 ; 4 ; 0 ; 0 ; 0 ; 2 ; 0 ; 9 ;
      7 ; 6 ; 0 ; 0 ; 0 ; 0 ; 0 ; 4 ; 0 ;
      2 ; 3 ; 0 ; 0 ; 5 ; 0 ; 0 ; 1 ; 7 ;
      0 ; 0 ; 1 ; 2 ; 0 ; 3 ; 9 ; 0 ; 0 ;
      0 ; 0 ; 5 ; 0 ; 8 ; 6 ; 4 ; 0 ; 0 )
    >> .
\end{verbatim}

Here, for example, the auxiliary operator \texttt{fill} places the givens in their corresponding cells, making thus their set of possible numbers a singleton, whereas the remaining cells will have \{1, 2, 3, 4, 5, 6, 7, 8, 9\} as their set of possible numbers. The sort \texttt{List} is an auxiliary sort over which lists of natural numbers live.
numbers are constructed using the operator \(_;_\) in the usual way \[9\].

\[
\text{op fill : Nat Nat List -> Object .}
\]
\[
\text{eq fill(R, C, (N ; LL)) =}
\]
\[
\text{if C == 9 then}
\]
\[
\text{fill(s R, 1, LL) fi .}
\]
\[
\text{eq fill(R, C, N) =}
\]
\[
\text{fill(R, s C, LL) fi .}
\]

The operators together with the equations define the static part of the system. Next we need to define its dynamics, the way the system evolves towards reaching a solution, by means of rules that capture the processes for solving sudokus explained in Section 3.1. The goal, of course, is to find a solution whenever one exists (the set of possible numbers becomes a singleton for every cell) or otherwise to return a message warning about its non-existence (some set of possible numbers becomes empty).

Each of our rewrite rules diminishes the number of elements in the set of possible numbers of some cell, in a way that faithfully mimics the presentation of the solving rules given in Section 3.2. We illustrate the naturalness with which they are written in Maude by presenting two of them; for the rest, we refer the reader to the files at http://maude.sip.ucm.es/~miguelpt/bibliography.

The Sudoku Split rule that splits a sudoku into two when the number of possible numbers in a cell is equal to two, creating a sudoku with the first possible number and another with the second, is represented by means of

\[
\text{var VConf : Configuration .}
\]
\[
\text{rl [sudokuSplit2] :}
\]
\[
\text{\langle msg('SearchingSolution) VConf < id(R, C) : cell | grd : G, pss : (P1 P2), num : 2 \rangle \Rightarrow \langle msg('SearchingSolution) VConf < id(R, C) : cell | grd : G, pss : P1, num : 1 \rangle \rangle}
\]
\[
\text{\langle msg('SearchingSolution) VConf < id(R, C) : cell | grd : G, pss : P2, num : 1 \rangle \rangle .}
\]

This rule can be applied provided that a Sudoku term has a cell with just two elements P1 P2 in its set of possible numbers. In this case the term will be rewritten to a term with two concatenated Sudoku terms as expected: one with P1 as its set of possible numbers and another with P2.

As another example, and to illustrate the use of conditional rewriting rules, let us consider the Second order Simplification rule: if two cells in the same
row (column or grid) have the same set of possible numbers and its cardinality is 2, then those numbers can be removed from the sets of possible numbers of every other cell in the same row (column or grid):

crl [simplify2nd] :
  < id(R1, C1) : cell | grd : G1, pss : (P P’), num : 2 >
  < id(R2, C2) : cell | grd : G2, pss : (P P’), num : 2 >
  < id(R3, C3) : cell | grd : G3, pss : (P LP3), num : N3 >
=>
  < id(R1, C1) : cell | grd : G1, pss : (P P’), num : 2 >
  < id(R2, C2) : cell | grd : G2, pss : (P P’), num : 2 >
  < id(R3, C3) : cell | grd : G3, pss : LP3, num : sd(N3,1) >
  if ((R1 == R2) and (R1 == R3)) or ((C1 == C2) and (C1 == C3))
  or ((G1 == G2) and (G1 == G3)) .

The remaining rules for solving sudokus are represented in a similar manner.

In addition to the rules for solving a sudoku, there are also a number of equations and rules to take care of “maintenance” issues: to finish the procedure when there is no possible solution and to stop the application of rules when the final solution has been reached. For example, we have found a solution and therefore can stop the solving procedure if both the maximum and the minimum cardinality of the sets of possible numbers (computed by the auxiliary operators maxCard and minCard) are equal to 1:

\[
\text{op val':}_2 : \text{List} \rightarrow \text{Attribute} .
\]

Then, when a solution exists, the final term has the following form:

\[
\text{op msg('FinalSolution)}
\]

Finally, to show the final term (a solved sudoku) in the desired format, a new attribute is added to the objects representing cells which is assigned the list of all the values in a given row:

\[
\text{op val'} : \text{List} \rightarrow \text{Attribute} .
\]

Then, when a solution exists, the final term has the following form:
4.2 Running the sudokus

The previous section has described how a procedure for solving sudokus can be specified in Maude. As discussed in Section 3.1, however, one cannot simply apply the rewrite rules in it to obtain a solution lest a combinatorial explosion is produced. Then, in order to avoid this it will be necessary to apply the rules according to some suitable strategy.

Maude has been extended with a powerful strategy language that allows a user to specify the ways in which the rewrite rules in a module will be applied [13]. This language is itself built on top of another extension, called Full Maude (see [9]) which adds to Maude a richer set of primitives for working with parameterization and object-oriented modules. We will not use any of the advanced features of Full Maude and for our purposes it will be enough to know that to load a module it is necessary to enclose it in parentheses. Regarding the strategy language, we will limit ourselves to a very limited subset; we refer the reader to [13] for the complete details.

A strategy module is declared with the keyword stratdef, strategy operators with sop, and the equations that define these operators are introduced with seq:

```
(stratdef STRAT is
  sop rules .
  seq rules = (simplify1st orelse
               (simplify2nd orelse
                (onlyOneNumber orelse
                 (simplify3rd orelse
                  (onlyTwoNumbers orelse
twins))))).
  sop split .
  seq split = (sudokuSplit2 orelse sudokuSplitN) .
  sop solve .
  seq solve = (rules orelse split) ! .
endsd)
```

This module defines three strategies. The first one, rules, simply tries to apply one of the first six rules, in order: it tries with simplify1st; if it is not possible, it tries with simplify2nd; and so on. The second one, split, tries to apply sudokuSplit and sudokuSplitN rules in order. The last one, solve, applies the first strategy and, only if it is not possible, tries to rewrite using splitting rules; the bang ! at the end asks to continue with the application of the strategy while possible.
To illustrate its use, after loading Maude with Full-Maude and the module with the strategies we can solve the sudoku in Figure 1 by means of:

```maude
Maude> (srew sudoku using solve .)
rewrite with strategy :
result Sudoku :
  << msg('FinalSolution')
    < id(1,0): rows | val :(5 ; 8 ; 3 ; 7 ; 2 ; 4 ; 1 ; 9 ; 6) >
    < id(2,0): rows | val :(1 ; 2 ; 6 ; 9 ; 3 ; 5 ; 8 ; 7 ; 4) >
    < id(3,0): rows | val :(4 ; 9 ; 7 ; 6 ; 1 ; 8 ; 3 ; 5 ; 2) >
    < id(4,0): rows | val :(3 ; 5 ; 9 ; 8 ; 4 ; 2 ; 7 ; 6 ; 1) >
    < id(5,0): rows | val :(8 ; 1 ; 4 ; 5 ; 6 ; 7 ; 2 ; 3 ; 9) >
    < id(6,0): rows | val :(7 ; 6 ; 2 ; 3 ; 9 ; 1 ; 5 ; 4 ; 8) >
    < id(7,0): rows | val :(2 ; 3 ; 8 ; 4 ; 5 ; 9 ; 6 ; 1 ; 7) >
    < id(8,0): rows | val :(6 ; 4 ; 1 ; 2 ; 7 ; 3 ; 9 ; 8 ; 5) >
    < id(9,0): rows | val :(9 ; 7 ; 5 ; 1 ; 8 ; 6 ; 4 ; 2 ; 3) > >>
```

5 Final remarks

We have presented in this paper a case study of how to use Maude to execute and solve sudokus. We have first shown a representation of sudokus in an object-oriented way and introduced rules for solving them, whose implementation in Maude is straightforward. Since the blind application of these rules would give rise to a combinatorial explosion, we have explained how to take advantage of Maude’s strategy language to apply the rules in a non-expensive way.

The main strength of our approach is the naturalness with which sudokus are represented in Maude and the ease with which the solving procedure is implemented. The specification can be easily modified to deal with sudokus of arbitrary order just by extending the rules with additional objects to represent the extra cells. However, it is precisely this need to add additional subterms to the rules which prevents the use of a single specification to cover all orders; such specification could be written by resorting to Maude’s metalevel [9], but that would make the specification more obscure. To illustrate how the extension works, the maude files for the specification to solve sudoku monsters (sudokus of order 4), along with the complete specification of the sudoku solver presented in the paper, are available at http://maude.sip.ucm.es/~miguelpt.

On the other hand, the weakness of our implementation lies in its efficiency. Even with the use of strategies to prune the search tree, our implementation cannot compete with the numerous solvers available in the web (e.g. [5,6,7]).

Acknowledgements.

We thank Alberto Verdejo and Narciso Martí-Oliet for helpful comments and discussions (specially about the strategy language), and the anonymous
referees for their suggestions.

References


Java+ITP: A Verification Tool Based on Hoare Logic and Algebraic Semantics

Ralf Sasse and José Meseguer

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA

Abstract

Java+ITP is an experimental tool for the verification of properties of a sequential imperative subset of the Java language. It is based on an algebraic continuation passing style (CPS) semantics of this fragment as an equational theory in Maude. It supports compositional reasoning in a Hoare logic for this Java fragment that we propose and prove correct with respect to the algebraic semantics. After being decomposed, Hoare triples are translated into semantically equivalent first-order verification conditions (VCs) which are then sent to Maude’s Inductive Theorem Prover (ITP) to be discharged. The long-term goal of this project is to use extensible and modular rewriting logic semantics of programming languages, for which CPS axiomatizations are indeed very useful, to develop similarly extensible and modular Hoare logics on which generic program verification tools can be based.

Key words: algebraic semantics; Hoare logic; program verification; Java

1 Introduction

This work is part of a broader effort, namely, the rewriting logic semantics project, to which a number of authors are contributing (see the recent surveys [19,18] and references there). The overall goal is to use rewriting logic semantic definitions of programming languages, including concurrent ones, and languages like Maude to generate efficient language implementations, including interpreters and compilers, and also sophisticated program analysis tools for those languages, including invariant checkers for infinite-state programs, model checkers, and theorem provers.
One of the appealing features of all these tools is their genericity: by exploiting a common underlying semantics and maximizing the modularity of language definitions it is often possible to generate program analysis tools for different languages in a generic way, using a common infrastructure, yet with competitive performance. In the case of interpreters, invariant checkers, and model checkers this has been convincingly demonstrated for many languages, including large subsets of Java and the JVM (see [19,18] for a detailed discussion of different such language case studies).

For the case of theorem provers the situation is less advanced. One unresolved and exciting research issue is finding generic and modular program logics in the Hoare style [12], to mathematically justify such logics on the basis of their rewriting logic semantics, and to develop generic theorem proving technology to support reasoning with such program logics in different languages. We are not there yet. In fact, we think that a sound way to approach this quite ambitious goal is to gather empirical evidence through case studies to help us find the outlines of such generic and modular program logics.

This paper makes some advances in this direction by focusing on a subset of sequential Java. Specifically we:

(i) Adapt the Maude-based continuation passing style (CPS) rewriting logic semantics for a large fragment of Java given in [7] by adding to it extra features making it suitable for theorem proving purposes. Although we focus for the moment on a modest sequential fragment, there is ample evidence, both in Java and in other languages (see the discussions in [19,18]), supporting the claim that CPS-based rewriting logic definitions are modular and extensible; therefore, we believe that our present work will naturally extend to more ambitious language fragments in Java and in other languages.

(ii) Develop a Hoare logic for this fragment and mathematically justify the correctness of our Hoare rules based on the CPS semantics. Even for this modest fragment this turns out to be nontrivial, because some of the standard Hoare rules, including the rules for conditionals and for while loops, are in fact invalid and have to be properly generalized in order to be applicable to Java programs.

(iii) Develop a mechanization of this Hoare logic supporting: (i) compositional reasoning with the Hoare rules to decompose Hoare triples into simpler ones; (ii) generation of first-order verification conditions (VCs); and (iii) discharging of such VCs by Maude’s inductive theorem prover (ITP) [4] using the underlying CPS semantics. Java+ITP has been developed as an extension of Maude’s ITP and is entirely written in Maude.

Although Java+ITP is primarily a research vehicle to help us advance the longer-term goal of developing generic logics of programs and generic program verifiers based on modular rewriting logic semantic definitions, we have also found it quite useful as a teaching tool at the University of Illinois at
Urbana-Champaign to teach graduate students and seniors the essential ideas of algebraic semantics and Hoare logic. It has been used quite extensively by students on a graduate course on Program Verification (CS 476) and will also be used this Winter on a Formal Methods graduate course (CS 477).

The conceptual basis of Java+ITP is exactly what one would expect of any theorem proving tool based on a language’s rewriting logic semantics. As already mentioned, the CPS semantics of our Java fragment is axiomatized in Maude. Since we focus for the moment on a sequential fragment, this defines an equation theory \texttt{JAVAX}. Therefore, the language’s mathematical semantics is precisely the initial algebra \( T_{\text{JAVAX}} \). We use this mathematical model \( T_{\text{JAVAX}} \) to justify the semantics of our Hoare rules. Similarly, the first-order VCs associated to Hoare triples are then \textit{inductive goals} that are claimed to be satisfied by the initial model \( T_{\text{JAVAX}} \), and that Maude’s ITP tries to discharge using the equation theory \texttt{JAVAX}. Therefore, for this fragment we are within the well known algebraic semantics framework \cite{10}; however, in future extensions including threads and concurrency, the semantics will instead be given by a rewrite theory, and the inductive reasoning will be based on the initial model of such a rewrite theory.

There is a substantial body of related work on Java logics, semantics and theorem proving tools, such as, for example, \cite{16,14,13,15,9,20,17,2,3}. We discuss this related work in Section 6; we also discuss there work closer to ours such as the Maude ITP \cite{4}, on which our tool is based, the ASIP-ITP tool \cite{6,22}, and of course the JavaFAN project \cite{7,8}, to which this work contributes at the theorem proving level. The rest of the paper is organized as follows. The CPS semantics of our Java fragment is summarized in Section 2. The first-order semantics of Hoare triples based on the initial algebra semantics of the language is explained in Section 3. Our Hoare logic and its justification are treated in Section 4. The mechanization of such a logic in the Java+ITP tool, and its use in examples are discussed in Section 5. Section 6 treats related work and conclusions. The related technical report \cite{24} contains a mathematical proof of correctness for the loop rule, and two proof scripts for Java programs.

2 Algebraic Semantics of a Sequential Java Subset

We present some of the highlights of the semantics of our chosen Java subset. We do not show the whole syntax, state infrastructure and actual semantics because of space limitations. However, the whole definition is available on the web at \cite{23}. The Java fragment we are interested in includes arithmetic expressions, assignments, sequential composition and loops. Our semantics uses a continuation passing style (CPS) approach. This has the advantage of making our semantic definitions easily extensible to accommodate additional Java features in the future. For example, exceptions, objects, multi-threading and all other Java features can be expressed using a CPS style as shown by
the prototype version in [7]. Our specification is similar in style to the prototype interpreter for a much bigger Java subset in [7], but has some differences/optimizations that take advantage of the sequential nature of our chosen subset. We illustrate our semantics by making explicit its state infrastructure and showing the syntax and semantics for a few selected features.

### 2.1 The State Infrastructure for Java

To be able to describe the semantics of Java we must specify how the execution of programs affects the state infrastructure, which contains the values for the program variables and other state information. The state infrastructure is defined by the following modules, where we separately specify the locations, environments, values, stores and continuations that make up the state.

A program variable will not be directly mapped to its value but to a location in the store. This leads to a two-level mapping, of variables to locations and of locations to values. The LOCATION module defines what a location is, an example location is \( l(17) \). It also shows how to concatenate multiple locations together, as we generally work on lists of expressions, etc.

```plaintext
fmod LOCATION is
  protecting INT .
  sorts Location LocationList .
  subsort Location < LocationList .
  op noLoc : -> LocationList .
  op _ , _ : LocationList LocationList -> LocationList [assoc id: noLoc] .
  op l : Nat -> Location .
endfm
```

The ENVIRONMENT module defines an environment as a finite map from names to locations and also gives equations which define how it can be updated. It imports the NAME module that defines names, lists of names and equality on names.

```plaintext
fmod ENVIRONMENT is protecting LOCATION .
  protecting NAME .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op _ , _ : Env Env -> Env [assoc comm id: noEnv] .
  vars X Y : Name . vars Env : Env . vars L L' : Location .
  var Xl : NameList . var Ll : LocationList .
  op [_,<-_] : Env NameList LocationList -> Env .
  op [_,<-_] : Env Name Location -> Env .
  eq Env[() <- noLoc] = Env .
  eq ([X,L] Env)[X <- L'] = ([X,L'] Env) .
    if equalName(Y, X) = false .
  eq noEnv [X <- L'] = [X,L'] .
endfm
```

205
For example, the environment

\[
(['X, 1(1)] \ ['Y, 1(2)]) \ ['X,'Y,'Z <- 1(3),1(4),1(5)]
\]
evaluates to \(['X,l(3)] \ ['Y,l(4)] \ ['Z,l(5)].

Values and stores are defined in the VALUE and STORE modules below. No equations are given for the store (unlike for the environment). This is due to our wish to stay extensible, which suggests that changes to the store should not be done here, but should instead be done in conjunction with, at least in a multi-threaded case, the currently working thread.

\[
\text{fmod VALUE is}
\text{sorts Value ValueList .}
\text{subsort Value < ValueList .}
\text{op noVal : -> ValueList .}
\text{op _,- : ValueList ValueList -> ValueList [assoc id: noVal] .}
\text{op [_] : ValueList -> Value .}
\text{endfm}
\]

\[
\text{fmod STORE is protecting LOCATION .}
\text{extending VALUE .}
\text{sort Store .}
\text{op noStore : -> Store .}
\text{op [_,_] : Location Value -> Store .}
\text{op __ : Store Store -> Store [assoc comm id: noStore] .}
\text{endfm}
\]

Environments and stores are defined in a very concrete way for this language. Using a more abstract environment/store concept would have its advantages from the point of view of program verification, as shown in [6,22] for a very simple language. But a more abstract concept of environment/stores does not work nicely with the side-effects and hiding that are possible in our language, for which the concrete variant we have chosen is preferable. Furthermore, this will make it easier to extend this subset of Java to a more complete version of Java in the future. In contrast, a more abstract definition of state would not allow more complex information, like exception, loop, or lock information, to be explicitly stored.

Within continuations, which we define next, all the execution context is stored. This can be viewed as “the rest of the program” which needs to be executed. The two operators shown here are two different ending points of an execution. Within the semantics we will define other operators with co-domain Continuation as needed. For example every expression of sort Exp can be put on the top (i.e. at the front) of a continuation.

\[
\text{fmod CONTINUATION is}
\text{sort Continuation .}
\text{op stop : -> Continuation .}
\text{op res : -> Continuation .}
\text{endfm}
\]

The state is made up of state attributes, which are the environment, store,
output and a counter for the next free memory location, each wrapped by some operator. Its structure is that of a set of such attributes obtained by the usual associative-commutative multiset union operator.

```plaintext
fmod STATE is extending ENVIRONMENT . extending STORE .
  extending CONTINUATION .
  sorts StateAttribute MyState .
  subsort StateAttribute < MyState .
  op empty : -> MyState .
  op e : Env -> StateAttribute .
  op n : Nat -> StateAttribute .
  op m : Store -> StateAttribute .
  op out : Output -> StateAttribute .
  sorts SuperState WrappedState .
  subsort WrappedState < SuperState .
  op noState : -> WrappedState .
  op state : MyState -> WrappedState .
  op k : Continuation -> SuperState .
endfm
```

The second set of sort declarations (and the operators for that) are needed because we do not want the context, i.e., the `Continuation`, to be part of the state, but only to be composable with it. So, instead of having `e(..), m(..), n(..), k(..)` we now have `state(e(..), m(..), n(..)), k(..)`.

Thanks to this structure we can check for termination of a program by simply checking the sort of the state. If it is of sort `SuperState`, there is still some continuation, and therefore code, left and the program has not yet terminated. If instead the resulting state is a `WrappedState`, we know that all code has been executed. The definition of what happens to an empty continuation needs to support this and does so.

### 2.2 Syntax and Semantics of Some Features

The Java fragment we are interested in includes arithmetic expressions, assignments, sequential composition and loops. Let us now take a look at the syntax and semantics of some features of our Java subset. We first discuss addition, then conditionals and finally loops.

#### Addition.

The syntax of addition is defined making use of the definition of generic expressions, which mainly just introduces the different possible forms of expressions.

```plaintext
fmod ARITH-EXP-SYNTAX is ex GENERIC-EXP-SYNTAX .
```
The operator \(+\) defined in \texttt{ARITH-EXP-SEMANTICS} allows us to evaluate an addition expression placed on top of a continuation. The first equation changes the evaluation of \(E + E'\) into first evaluating \((E, E')\), and then evaluating \(\rightarrow +\). The second equation evaluates \(\rightarrow +\) by adding the two integers obtained by evaluating the expressions and placing the result on the top of the continuation stack.

\fmod \texttt{ARITH-EXP-SEMANTICS} is protecting \texttt{ARITH-EXP-SYNTAX}.
\begin{align*}
&\texttt{op \(+\) : Continuation \rightarrow Continuation}. \\
&\texttt{vars \(E/E'\) : Exp. var \(K\) : Continuation. vars \(I/I'\) : Int.} \\
&\texttt{eq \(k((E + E') \rightarrow K) = k((E,E') \rightarrow + \rightarrow K)\).} \\
&\texttt{eq \(k((\text{int}(I), \text{int}(I')) \rightarrow + \rightarrow K) = k(\text{int}(I + I') \rightarrow K)\).}
\end{align*}
\end{fmulticols}

\begin{fmulticols}{2}
\textbf{If-Then-Else.}\n
In Java, the If-Then-Else construct does not actually contain a \texttt{then} but has instead the syntax specified in \texttt{IF-SYNTAX}, that imports \texttt{Statement}, a construct different from expressions since it does not create a return value. By the specified parsing precedences the dangling else problem is solved as in the Java Language Specification \cite{11}, that is, the \texttt{else} part belongs to the innermost \texttt{if}. We consider the If-Then as \emph{syntactic sugar} and therefore give one equation in \texttt{IF-SYNTAX} which translates it into our If-Then-Else, meaning that we do not need to bother with it in the semantics at all. Also, one equation is enough for this desugaring.

\fmod \texttt{IF-SYNTAX} is \texttt{ex STATEMENT-SYNTAX}.
\begin{align*}
&\texttt{ex \texttt{GENERIC-EXP-SYNTAX}.} \\
&\texttt{op if\_\_else\_ : Exp Statement Statement \rightarrow Statement \ [prec 110].} \\
&\texttt{op if\_\_ : Exp Statement \rightarrow Statement \ [prec 115].} \\
&\texttt{var \(E\) : Exp. var \(St\) : Statement.} \\
&\texttt{eq \(\text{if } E \text{ } St = \text{if } E \text{ } St \text{ else } ;\).}
\end{align*}
\end{fmulticols}

The evaluation of a conditional statement is split up into first evaluating the condition, while freezing the two code parts in the continuation, and then, once the condition is evaluated to either \texttt{true} or \texttt{false}, choosing the correct path. Note that we need to import boolean expressions here.

\fmod \texttt{IF-SEMANTICS} is \texttt{ex IF-SYNTAX} \texttt{ex GENERIC-EXP-SEMANTICS}.
\begin{align*}
&\texttt{ex STATEMENT-SEMANTICS} \texttt{ex BEXP-SEMANTICS}. \\
&\texttt{op \?(__,__) \rightarrow _: Statement Statement Continuation \rightarrow Continuation}. \\
&\texttt{var \(E\) : Exp. vars \(St/St'\) : Statement. var \(K\) : Continuation.} \\
&\texttt{eq \(k((\text{if } E \text{ } St \text{ else } St') \rightarrow K) = k(E \rightarrow ? (St, St') \rightarrow K)\).} \\
&\texttt{eq \(k(\text{bool}(\text{true}) \rightarrow ? (St, St') \rightarrow K) = k(St \rightarrow K)\).} \\
&\texttt{eq \(k(\text{bool}(\text{false}) \rightarrow ? (St, St') \rightarrow K) = k(St' \rightarrow K)\).}
\end{align*}
\end{fmulticols}
While loops.

The syntax for while loops is straightforward. Note that the second argument of a while is a statement, but one can always wrap the sequential composition of several statements into a single block by using curly braces, e.g. \{S1 S2\}, with S1 and S2 statements. A block counts as a single statement again.

```latex
fmod WHILE-SYNTAX is ex STATEMENT-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
endfm
```

Defining the semantics of loops is now very easy by using the semantics of the conditional and unrolling the loop one step at a time:

```latex
fmod WHILE-SEMANTICS is ex WHILE-SYNTAX . ex GENERIC-EXP-SEMANTICS .
  ex STATEMENT-SEMANTICS . ex IF-SEMANTICS .
  var E : Exp . var St : Statement . var K : Continuation .
  eq k((while E St) -> K) = k(E -> ?({St while E St}, ;) -> K) .
endfm
```

2.3 An Interpreter for our Java Subset

The complete functional definition gives a precise mathematic axiomatization, in fact an initial algebra semantics of our chosen subset of Java that is sufficient for reasoning and program verification purposes. But since the semantic equations are ground confluent, the above semantic equations also give an operational semantics to this Java subset.

Indeed, we can describe the execution of the language by algebraic simplification with the equations from left to right. Therefore, our language definition has in essence given us an interpreter for our language. Note that in a few minor points we do not adhere to the strict syntax of Java because of some of the built-in types of Maude. For example, program variables are modeled with Maude quoted identifiers and therefore always have a quote (‘) in front, and integers are wrapped with the operator \#i() to avoid operations in Maude’s built-in INT module to interfere with arithmetic operations in Java.

With initial we create an initial empty state. By adding ‘| CODE’ to any state, where ‘CODE’ is some code fragment, of sort BlockStatements, we can compute the state resulting from executing that code fragment in the given state. Also, with STATE[VAR] the value of the variable VAR in a state STATE is returned. The equation accomplishing this is:

```latex
op _[_] : WrappedState Name -> Value .
var MYS: MyState . var X : Name . var L : Location . var Env : Env .
var V : Value . var M : Store .
var MYS: MyState . var L : Location . var Env : Env .
var V : Value . var M : Store .
var V : Value . var M : Store .
```

Some examples are:

```latex
red (initial | (int 'x = #i(1) ; int 'y = #i(20) ;
{ 'x = #i(300) ; } 'x = 'x + 'y ;))['x] .
```
A simple swap example, where swap is just a short-hand notation for the program defined by the equation
\[ \text{swap} = (\text{int } 'T = 'X ; 'X = 'Y ; 'Y = 'T ;) \], indeed swaps the values of 'X and 'Y, the results are 5, respectively 7, as expected.

\[
\text{red (initial | (int } 'X = #i(7) ; \text{ int } 'Y = #i(5) ; \text{ swap)})['X].}
\]

\[
\text{red (initial | (int } 'X = #i(7) ; \text{ int } 'Y = #i(5) ; \text{ swap)})['Y].}
\]

A small factorial program

\[
\text{red (initial | (int } 'n = #i(5) ; \text{ int } 'c = #i(0) ; \text{ int } 'x = #i(1) ; \text{ while ('c < 'n) { 'c = 'c + #i(1) ; 'x = 'x * 'c ; } }) ['x].}
\]

computes the factorial of 5 and thus returns:

\[
\text{rewrites: 416 in 0ms cpu (0ms real) (~ rewrites/second)}
\]
\[
\text{result Value: int(120)}
\]

### 3 Hoare Triples

#### 3.1 Pre and Post Conditions

Recall the swap program we have just shown in Sect. 2.3. A correctness specification for that example program, when done in an equational setting, could look like this:

\[
(\forall I : \text{Int})(\forall J : \text{Int})(S)[['Y] = \text{int}(I) \land (S)[['X] = \text{int}(J)
\]
Here we have the implicit precondition that we are starting in a state where 'X and 'Y are declared as described above. We shall call the equation

\[(S | (\text{swap}))['X'] = \text{int}(I) \land (S | (\text{swap}))['Y'] = \text{int}(J)\]

which is assumed to hold before the execution of the program, the \textit{precondition}.

Note that this equation has a \textit{single occurrence} of the state variable S in each equation, and can be thought of as a \textit{state predicate}, having the integer variables I and J as \textit{parameters}. Consider in the above specification the equation

\[(S | (\text{swap}))['X'] = \text{int}(I) \land (S | (\text{swap}))['Y'] = \text{int}(J)\]

which is supposed to hold after the execution of a program. This can also be viewed as a state predicate, namely the state predicate

\[(\dagger) (S)['X'] = \text{int}(I) \land (S)['Y'] = \text{int}(J)\]

applied not to S, but instead to the state S | (\text{swap}) after the execution. We call \((\dagger)\) the \textit{postcondition}. Note that it also has the integer variables I and J as extra parameters.

\section*{State Predicates.}

This example suggests a general notion of \textit{state predicate}, intuitively a \textit{property} that holds or does not hold of a state, perhaps relative to some extra \textit{data parameters}. Since in our Java subset the only data are integers, such parameters must be integer variables.

Therefore, for our language we can define a \textit{state predicate} as a conjunction of equations

\[t_1 = t'_1 \land \ldots \land t_n = t'_n\]

in the module \text{JAVAX}, such that the set \(V\) of variables in all terms in the equations has \textit{at most one} variable S of sort \texttt{State}, which may possibly appear more than once, and the remaining variables are all of sort \texttt{Int}.

One can of course generalize things further, by allowing an \textit{arbitrary first-order formula} (with the same condition on its variables \(V\)) instead of just a conjunction of equations. Also, the notion extends naturally to other sequential languages which may have \textit{other data structures} besides integers. However, in practice the above notion is quite general; among other things because, using an equationally defined equality predicate, we can express \textit{arbitrary Boolean combinations} of equations (and therefore any quantifier-free formula) as a \textit{single equation}.
3.2 Hoare Triples

The above example of our specification for swap is paradigmatic of a general way of specifying properties of a sequential imperative program $p$ by means of a Hoare triple (after C.A.R. Hoare, see [12]),

$$\{A\} \ p \ \{B\}$$

where $A$ and $B$ are state predicates, called, respectively, the precondition, and postcondition of the triple.

In this notation, the specification of swap becomes rephrased as,

$$\{(S)[ \text{"Y"}] = \text{int}(I) \land (S)[ \text{"X"}] = \text{int}(J)\} \ \text{swap} \ \{(S)[ \text{"X"}] = \text{int}(I) \land (S)[ \text{"Y"}] = \text{int}(J)\}$$

Given our algebraic approach to the semantics of imperative programs, this is all just an (indeed very useful) façon de parler about an ordinary first-order property satisfied by the initial model of our language, namely the initial algebra $T\text{JAVAX}$. The module JAVAX is the module defining the semantics of our Java subset. It imports all other modules, defining the syntax, state infrastructure, and semantics.

Therefore, we define the partial correctness of a program $p$ with respect to a Hoare triple by the equivalence,

$$T\text{JAVAX} \models \{A\} \ p \ \{B\} \iff T\text{JAVAX} \models (\forall V) \ A \land ((S \ | \ p) : \ \text{WrappedState}) \Rightarrow (B(S/S \ | \ p))$$

Here the $:\$ means sort membership, which in turn means that program $p$ terminates when started in state $S$. Note that in the partial correctness interpretation the termination condition is on the lefthand side of the implication.

Our swap example thus becomes

$$T\text{JAVAX} \models (\forall I : \text{Int})(\forall J : \text{Int})(\forall S : \text{State})(S)[ \text{"Y"}] = \text{int}(I) \land (S)[ \text{"X"}] = \text{int}(J)$$

$$\land (S \ | \ \text{swap}) : \text{WrappedState}$$

$$\Rightarrow (S \ | \ (\text{swap}))[ \text{"X"}] = \text{int}(I) \land (S \ | \ (\text{swap}))[ \text{"Y"}] = \text{int}(J).$$

which is just our original correctness condition with the addition of the termination condition by the sort requirement. Of course, since swap was a terminating program, this was superfluous in that case, but it is not superfluous when loops are involved.
4 A Hoare Logic for our Java Subset and its Justification

An important contribution of Hoare was to propose his triples as a *compositional logic of programs*, by giving a collection of *inference rules* based on the *structure of the program text* to decompose the proof of correctness of more complex programs into proofs for simpler subprograms.

Hoare logic, however, is language-dependent: a Hoare rule valid for a construct in a given language may be invalid in another. For example, the classical Hoare rules for conditionals and for loops are both *invalid* even in our simple Java fragment and have to be suitably modified. It becomes therefore important to: (i) choose Hoare rules that adequately capture a given feature in a specific language and (ii) to mathematically *justify* the correctness of such a rule. For this second purpose, having a precise mathematical semantics of the language in question is an essential prerequisite. We therefore introduce a Hoare logic for our Java subset and justify the correctness of its rules based on our JAVAX formal semantics.

For example, to prove the correctness of a sequential composition \( p \; q \) he gave the rule,

\[
\{ A \} \; p \; \{ B \} \quad \{ B \} \; q \; \{ C \}
\]

which can be easily justified for our Java subset by analyzing both the semantic equations and the semantics of the triples.

Another rule of easy justification in our Java semantics is the rule for the skip program `;`, which takes the form,

\[
\{ A \} \; \{ A \}
\]

For *conditionals* we need to work a little harder, because the classical Hoare rule for conditional is *invalid*. The key difficulty is that evaluating a conditional’s boolean expression may side-effect the state. Here, `evalTst(S, TE)` gives the boolean which the evaluation of the test expression \( TE \) in state \( S \) returns. Using the operator `|` we can separate the “execution” of a test expression from the rest of the program. So we have now overloaded `|`’s meaning to both combine a state and a program fragment and also to combine a state and an expression, but because of the different typings involved no ambiguity arises and this is not a problem. Furthermore, since in the Hoare triples we sometimes need to consider the execution of an expression for side effect purposes only in conjunction with the execution of a statement, our Hoare triples allow not only statements, but also expressions. Any “sequential composition” of them using the `|` symbol in an “ad-hoc” overloaded way is allowed. Note that such “compositions”, though meaningful in terms of their effects on state, do not correspond to legal Java programs; however, they are needed in the Hoare rules. Of course, both uses of `|` are closely related, since, for example,
given a state $s$, an expression $e$, and a statement $p$ we semantically interpret
the effect of $e \mid p$ on $s$ by the equation

$$s \mid (e \mid p) = (s \mid e) \mid p.$$  

It is not possible to use the usual Java program concatenation here, because
the test expression is not of the same sort as the other statements. But using
the $\mid$ operator it can be evaluated first, so that its side effects change the state,
and then the result gets thrown away and the execution continues as usual.
This is our way to allow the expression to be used as if it were a statement,
just for its side-effects.

The function $\text{evalTst}$ evaluates a test expression in a given state to a
boolean value. As it is a bit cumbersome to write this out multiple times in
some of the rules, we overload our notation a little and use a test $t$ in two
different ways in the following. In the property part of a Hoare triple, $t$
will stand for the equality $\text{evalTst}(S, t) = \text{true}$, with $S$ the variable for the
distinguished state for which that property has to hold, and similarly, $\neg t$
will stand for $\text{evalTst}(S, t) = \text{false}$. That use of $t$ (respectively its negation)
only gives us the boolean value and does not change the state. Whenever the
state $S$ is not obvious, we will fall back on the $\text{evalTst}$ notation. The other
way we use $t$ in is in the code part as usual (within $\text{if}$ or $\text{while}$ constructs)
or just for its possible state change as described above. The different uses of
$t$ are illustrated in our Hoare rule for conditionals,

$$\{A \land t\} t \mid p \{B\} \quad \{A \land \neg t\} t \mid q \{B\}$$

$$\{A\} \text{if } t \text{ p else q } \{B\}$$

This captures the usual semantics of $\text{if}$, just as in the simpler languages,
but in contrast here, since $t$ can have side effects, we have $t \mid$ in front of
the execution of the two branches of the conditional in the respective cases. It
is not enough to know that $t$ evaluates to either true or false, which is what
the two properties assure, but $t$ needs to be also executed for its possible side
effects. This Hoare rule still simplifies things, since we now do not have to
take a decision based on the test value anymore, but we just have to have the
test expression executed. One could also give a sequential composition rule
for $\mid$ additionally. The extra effort here is necessary because of side effects!

Another very useful rule, of easy justification based on the semantics of
Hoare Triples, is the consequence rule,

$$A \Rightarrow A_1 \quad \{A_1\} p \quad \{B_1\} \quad B_1 \Rightarrow B$$

$$\{A\} p \quad \{B\}$$

The most important rule in our language subset is the proof rule for the
partial correctness of $\text{while}$ loops. Here we face the same problem as with
conditionals, because the loop condition can also have side effects. It takes
the form,
This rule requires a somewhat more involved justification, which is done in the proof given in the technical report \cite{24}. The state predicate $A$ is called an invariant of the loop. This rule needs the additional Hoare triple for the test:

$$(i) \quad \{ A \land \neg t \} \ t \{ A \land \neg t \}$$

because of the way side effects can propagate with the loop unrolling. A loop works like this:

$$\text{while } t \ \text{p} \rightarrow t \mid p \mid \text{while } t \ \text{p} \rightarrow ... \rightarrow \ t \mid p \mid \text{while } t \ \text{p} \rightarrow ... \mid t \mid p \mid t$$

In the final state that is thus attained, the test $t$ does not necessarily evaluate to \texttt{false}. In the state before the final state it did indeed evaluate to \texttt{false}, but its side effect could cause its next evaluation to be \texttt{true} again. To prevent this, the Hoare triple $(i)$ has to be added to the proof obligation of the loop rule.

An example Java program where this problem appears is the following:

```java
int 'i = #i(0) ; while ( ! ( ('i = 'i + #i(1) ) == #i(1) ) ) ...
```

Here in the condition check '$i$ is increased to 1, so the equality holds and therefore the negation is false and the loop is never entered. But if the condition were evaluated in this final state, '$i$ would get the value 2, the equality would not hold and therefore the negation would hold. So here the condition is not false in the final state.

A Factorial Example.

Consider the factorial program in Section 2.3. To prove its correctness, intuitively that it correctly computes the factorial function, we first need to define mathematically such a function, by defining an operator $\text{facValue}$ and its defining equations,

$$\text{op facValue : Int -> Int .}$$

$$\text{var } I : \text{Int} .$$

$$\text{ceq facValue}(I) = 1 \text{ if } 0 < I \text{ is false .}$$

$$\text{ceq facValue}(I) = I \times \text{facValue}(I - 1) \text{ if } 0 < I \text{ is true .}$$

To avoid complications with non-termination we have defined the factorial of a negative number to be 1.

We are only interested in the meaningful results of the factorial function. Therefore, we should give the requirement that the input variable '$N$ is non-negative as a precondition, yielding the specification,
\{(S[\text{'N}] = \text{int}(I)) \land (0 \leq I = \text{true})\} \text{ facx } \{S[\text{'X}] = \text{int}(\text{facValue}(I))\}.

The above specification takes the point of view of a customer who specifies properties of the desired program. An implementer may then give to the customer the following facx program:

'\text{C}' = \#i(0) ; 'X' = \#i(1) ; \text{while } ('\text{C} < '\text{N}) \{ '\text{C}' = '\text{C}' + \#i(1) ; 'X' = 'X' \times '\text{C}' ; \}

The question, then, is how to prove this program correct. To do so we can:

- use the Hoare logic rules, which we have justified, and
- use inductive reasoning, since the correctness of Hoare triples reduces to satisfaction of first-order formulas in the initial model $T_{JAVAX}$.

A proof script of this program in our Java+ITP Tool is given in the technical report [24].

5 The Java+ITP Tool

The latest version (extended by us with support for this Java subset) of the ITP is downloadable from [23] together with the semantics of the Java fragment. It has an extension of the list of commands of the ITP specifically designed to support Hoare logic reasoning in our programming language. How the Java+ITP tool works in detail and is interfaced with Maude’s ITP is also explained on the above-mentioned web-page.

5.1 Proving Hoare Triples in the ITP with the \texttt{javax} Command

In Java+ITP the \texttt{javax} command translates a Hoare triple into its semantically equivalent inductive theorem proving goal. For example, a goal consisting of the Hoare triple

\{P\} C \{Q\}

is translated into the (universally quantified) ITP goal

$P \Rightarrow Q(S/(S\mid C))$

where $S$ is the distinguished variable of sort \texttt{WrappedState}.

5.2 Proving While Loops with \texttt{javax-inv}

The \texttt{javax} command allows a user to enter a Hoare triple goal into the ITP to prove the correctness of the program mentioned in the triple. However, the compositional approach favored by Hoare logic suggests that we should first decompose the original Hoare triple into simpler ones by using the Hoare logic inference system. For this reason, the Java+ITP tool not only does
automate the entering of Hoare triples into the ITP. It also automates the application of some Hoare rules. For while loop programs, this is accomplished by means of the `javax-inv` command. We consider while loop programs of the general form \(walp = \text{init} \; \text{loop} \), with \(\text{loop} = \text{while} \; t \; p\). That is, we allow the subprogram `init` to be executed before the while loop proper, since this is a very common situation.

The `javax-inv` command allows the specification of the following information about a while loop program \(walp\) of the form just described:

- the precondition \(P\) and postcondition \(Q\) against which one wants to prove \(walp\) correct.
- the invariant \(A\) that should be used to decompose the original Hoare triple into simpler ones using the Hoare rules.

The `javax-inv` command then does the following things:

- it applies the *composition* rule to:

\[
\frac{\{P\} \; \text{init} \; \{A\} \quad \{A\} \; \text{loop} \; \{Q\}}{\{P\} \; \text{init} \; \text{loop} \; \{Q\}}
\]

- it then applies the *consequence* rule to further decompose the second subgoal

\[
\frac{A \Rightarrow A \; \{A\} \; \text{loop} \; \{A \land \neg t\} \quad (A \land \neg t) \Rightarrow Q}{\{A\} \; \text{loop} \; \{Q\}}
\]

- it finally applies the *loop* rule:

\[
\frac{\{A \land t\} \; t \; | \; p \; \{A\} \quad \{A \land \neg t\} \; t \; \{A \land \neg t\}}{\{A\} \; \text{while} \; t \; p \; \{A \land \neg t\}}
\]

As a consequence, the following four subgoals are generated:

(i) \(\{P\} \; \text{init} \; \{A\}\)
(ii) \(\{A \land t\} \; t \; | \; p \; \{A\}\)
(iii) \((A \land \neg t) \Rightarrow Q\)
(iv) \(\{A \land \neg t\} \; t \; \{A \land \neg t\}\).

The implementation of the `javax-inv` command in the ITP then implicitly applies the `javax` command to the Hoare triples (1), (2) and (4), so that we end up with the following four ITP goals:

(i) \(P \Rightarrow A(S/S \mid \text{init})\)
(ii) \((A \land t) \Rightarrow A(S/S \mid t \mid p)\)
(iii) \((A \land \neg t) \Rightarrow Q\)
(iv) \((A \land \neg t) \Rightarrow A(S/S \mid t) \land (\text{evalTst}(S/S \mid t, t) = \text{false})\)
5.3 Supporting Compositionality

From a user’s perspective, the commands `javax` and `javax-inv` directly create the first order goals for their corresponding Hoare triples. However, by the way these commands are structured they do not allow application to more complex programs such as, for example, a program of the form `init1 loop1 init2 loop2`. But obviously, by applying the composition rule, with a suitable middle condition, a program like this could be split up into two parts, so that each could be treated by the commands that we have already discussed.

To allow this kind of compositional reasoning, Java+ITP provides several commands. First of all, to enter a Hoare triple into the tool *without* translating it into its corresponding first-order goal the `add-hoare-triple` command can be used. Furthermore, Java+ITP also offers a `decompose` command, which decomposes a Hoare triple and its code into two Hoare triples with a suitable middle condition (provided by the user). That is, given `{A} P {B}` with `A` and `B` state predicates and `P` a program we can decompose this into the two Hoare triples `{A} P1 {C}` and `{C} P2 {B}` with `C` a state predicate and `P1` and `P2` two programs, all three provided by the user giving the `decompose` command, where the two programs need to make up `P`, i.e. `P = P1 P2`.

After having decomposed in this way the original Hoare triple for a program into several simpler Hoare triples, Java+ITP then supports translating such simpler triples into first-order goals. This is accomplished with the commands `create-FO-goal-hoare`, and `create-FO-goal-hoare-inv`, which are the respective analogues of the `javax` and `javax-inv` commands.

This support for compositionality allows us to tackle more complicated programs, for example programs involving multiple loops. Using the above commands we can create a number of Hoare triple goals from just one starting goal and then can generate the respective first order goals for all of them and discharge them with the ITP.

5.4 An Example: A Binomial Coefficient Program

We show the usefulness of introducing Hoare triples, decomposing them and then proving the separate subgoals with an example of the binomial coefficient function \( \binom{n}{k} \). The details of this decomposition and proof can be found in the technical report [24]. The main facts are the program:

```plaintext
op choose-program : -> BlockStatements .

eq choose-program = (  
  int 'N ; int 'Nfac ; int 'K ; int 'Kfac ;  
  int 'N-Kfac ; int 'BC ; int 'I ;  

  'I = #i(0) ; 'Nfac = #i(1) ;  
  while ('I < 'N) { 'I = 'I + #i(1) ; 'Nfac = 'Nfac * 'I ; }  

  'I = #i(0) ; 'Kfac = #i(1) ;  
  while ('I < 'K) { 'I = 'I + #i(1) ; 'Kfac = 'Kfac * 'I ; }  
```

218
\begin{verbatim}
'I = #i(0) ; 'N-Kfac = #i(1) ;
while ('I < ('N - 'K)) { 'I = 'I + #i(1) ; 'N-Kfac = 'N-Kfac * 'I ; }

'BC = 'Nfac / ('Kfac * 'N-Kfac) ;
\end{verbatim}

and the property to be verified:

\[
\begin{align*}
\{ \text{int-val(S:WrappedState['N])} &= (N: Int) \land \text{int-val(S:WrappedState['K])} = K: Int \\
& \land 0 \leq N: Int = \text{true} \land 0 \leq K: Int = \text{true} \\
\} \\
\text{choose-program} \\
\{ \text{int-val(S:WrappedState['Nfac])} &= (N: Int) ! \\
& \land \text{int-val(S:WrappedState['Kfac])} = (K: Int) ! \\
& \land \text{int-val(S:WrappedState['N-Kfac])} = (N: Int - K: Int) ! \\
& \land \text{int-val(S:WrappedState['BC])} = \text{choose(N: Int, K: Int)} \}
\end{align*}
\]

The basic idea is then to give suitable middle conditions to split the Hoare triples apart. The empty lines of the program indicate the split positions within the program.

## 6 Related Work and Conclusions

We first discuss related work using rewriting logic and the Maude system [5]. The CPS style has been found to be quite expressive and extensible in several experiments in the rewriting semantics project [19,18]; it has in particular been used for Java in the JavaFAN project [7,8]. We have adopted this semantics in Java+ITP for extensibility reasons; but, as discussed in Section 2, we structured the state and added extra functionality to suit theorem proving uses. Java+ITP is an extension of Maude’s ITP [4]. A project similar to ours, namely the ASIP+ITP tool [6,22], has been carried out by M. Clavel and J. Santa-Cruz at UCM in Madrid. While benefitting from their experience, we had to address and solve new research issues. ASIP+ITP is based on a considerably simpler programming language used by Goguen and Malcolm [10]; one whose expressions do not have any side-effects, whose variables can be directly mapped to values in memory, and where the whole semantics cannot be extended to accommodate new features. We are primarily interested in modularity and extensibility of programming languages and Hoare logics, and view Java+ITP as a research vehicle to advance those goals. Another difference is Java+ITP’s support for compositional reasoning: in ASIP+ITP VCs for Hoare triples, including those for loops, can be generated, but triples cannot be decomposed into simpler ones. Still partially in this framework, W. Ahrendt, A. Roth and the first author report in [2] on a cross-validation of a Java semantics given in the rewriting semantics framework against the Java program transformation rules of the KeY prover [1].
Commenting more broadly on Java verification work, the Java Modeling Language, JML [3], is a good way to specify the relevant properties of programs. We have not yet made use of JML in this work, but extending Java+ITP in this direction seems worthwhile. In [15], B. Jacobs, C. Marché and N. Rauch give an overview of the capabilities of different tools by comparing how they can deal with a real-world example program. They look at ESC/Java [9], Jive [20], Krakatoa [17] and the LOOP project [16]. ESC/Java [9] is only a checker which is neither sound nor complete. Jive [20] is based on Hoare logic, but it has no side-effects, at least not in the expressions which are used to take decisions, like the if and while test expressions. Therefore the resulting Hoare logic is much simpler. Krakatoa [17] uses a modeling of the Java heap and it makes use of several sub-tools which create the proof obligations for it. They work with Java but do not have a Hoare logic approach.

In the LOOP project [16], a denotational semantics of Java is formalized as a PVS theory. Java programs are compiled into semantical objects, and proofs are performed in the PVS theory directly. On top of that, a Hoare-style and a weakest precondition (wp) style calculus are formalized as a PVS theory, and are verified against the semantics within PVS. As opposed to ‘usual’ Hoare-style or wp calculi, these ones work on the semantical objects, not on the Java syntax. In his weakest-precondition reasoning work [14], B. Jacobs also works only on the semantical object level. Similarly, M. Huismann, in her thesis [13], also works on this semantic translation of the source code into the type theory (of PVS or Isabelle). The Hoare logic is given on that level only, not on the Java source code, which is the difference to our work.

In conclusion, we view Java+ITP as a research vehicle to investigate modularity and extensibility of programming languages and of Hoare logics. It has served us well for this purpose, by uncovering subtleties in the Hoare logic needed for Java not present in toy languages, and not even present in the Hoare logics of Java tools like Jive. Keeping the compositional Hoare logic reasoning at the source code level is also one of the goals that, in contrast to other approaches, we have advanced. But of course this is just a snapshot of work in progress. Our Java fragment is still quite modest, so we should soon add new features to it such as exceptions and objects; we expect this to be easy thanks to the CPS semantics. After this, threads and concurrency should also be added, and Hoare rules for these new features should also be investigated. Our goal is of course modularity, so that our Hoare rules will be applicable not just to Java, but to any other languages using some of the same features in a modular way, but this still remains an exciting goal for the future.
Acknowledgement

We cordially thank Manuel Clavel for his generous help with details of the ITP implementation. This help has been crucial for us to extend the ITP into Java+ITP.

References


URL http://maude.sip.ucm.es/~juansc/


A Rewriting Logic Framework for Soft Constraints

Martin Wirsing\textsuperscript{1,2}  

\emph{Ludwig-Maximilians-Universität München, Institut für Informatik, 80538 München}

Grit Denker, Carolyn Talcott, Andy Poggio, Linda Briesemeister\textsuperscript{3}

\emph{SRI International, 333 Ravenswood Ave, Menlo Park, California 94025}

\begin{abstract}
Soft constraints extend classical constraints to deal with non-functional requirements, over-constrained problems and preferences. Bistarelli, Montanari and Rossi have developed a very elegant and abstract semiring based theory of soft constraints where many different kinds of soft constraints can be represented and combined in a uniform way over so-called constraint semirings. In this paper we present a framework for prototyping of soft constraints à la Bistarelli, Montanari and Rossi in Rewriting Logic. As a case study we present an application of soft constraints to the new area of software-defined radio networks. We model the problem of “optimal” parameter assignments for software-defined radios as a soft constraint solving problem, prove the correctness of the constraint solving algorithm, implement the solution in our prototypical Rewriting Logic framework for soft constraints, and embed our soft constraint solver in SRI’s Policy-Aware, Goal-Oriented Distributed Architecture (PAGODA) for modelling radio networks.

\textbf{Key words:} Rewriting logic, soft constraint, software-defined radio.
\end{abstract}

\section{Introduction}

Soft constraints are an extension of classical constraints to deal with non-functional requirements, over-constrained problems and preferences. Instead of determining just a subset of admissible domain elements, a soft constraint assigns a grade - to be chosen from a set of finite or infinitely many “preference” values - to each element of the application domain. Bistarelli, Montanari and Rossi\textsuperscript{4} have developed a very elegant and abstract semiring based theory of soft constraints where many

\textsuperscript{1} This work has been partially sponsored by the project SENSORIA, IST-2005-016004.
\textsuperscript{2} wirsing@lmu.de
\textsuperscript{3} firstname.lastname@sri.com
\textsuperscript{4} bistarelli@unifi.it, montanari@unifi.it, rossi@unifi.it
different kinds of soft constraints can be represented and combined in a uniform way over so-called constraint semirings.

In this paper, we present a framework for prototyping of soft constraints à la Bistarelli, Montanari and Rossi in Rewriting Logic [7]. To our knowledge this is the first rewriting realisation of soft constraints.

Other implementations are based on constraint logic programming and concurrent constraint programming: \texttt{clp(FD, S)} [13] and \texttt{softclp(FD)} [17] are extensions of \texttt{clp(FD)} by constraint semirings. The former is based a new abstract machine resulting in a good efficiency; the latter is an extension of the \texttt{clp(FD)} library and can in this way reuse a broad class of constraint propagation algorithms and search methods from any parent \texttt{clp(FD)} solver. The approach of [3] uses Frühwirth’s Constraint Handling Rules for soft constraint propagation, whereas [9] extend the concurrent constraint language Mozart by soft constraints.

Our rewriting logic framework consists of a package of Maude theories and parameterized functional modules which can be integrated easily in any other Maude application by instantiating the parameter theories with the particular settings of the application. The axiomatic theory of constraint semi-rings is modelled as Maude functional theory; special constraint semi-rings such as weighted sum and fuzzy natural numbers are modelled as functional Maude modules; all other modules of the framework (such as cartesian products of constraint semirings, implicit and explicit soft constraints as well as the constraint solving algorithms) are parameterized by the choice of the constraint domain and semiring. In order to improve the efficiency of constraint solving, all recursive specifications are written in tail-recursive form and domain elements of individual constraints are ordered according to their grade in the constraint semiring. We also prove the correctness of our search algorithm.

As a case study, we present an application of soft constraints to the new area of software-defined radio networks (see e.g. \url{http://www.sdrforum.org}). A software-defined radio is a radio, in which most or all frequency control, modulation/demodulation formats, bandwidth, and other parameters are realized by software and thus can be changed during radio operation. This offers a tremendous new flexibility to commercial and amateur radios but controlling such radios in different environments requires subtle fine-tuning and adaptation of their parameter settings. We model the problem of “optimal” parameter assignments for software-defined radios as a soft constraint solving problem using a cartesian product of constraint semirings, implement the solution in our prototypical Rewriting Logic framework for soft constraints, and embed our soft constraint solver in SRI’s Policy-Aware, Goal-Oriented Distributed Architecture (PAGODA) for modelling radio networks. Note that although other implementations such as [3] and [9] can also cope with cartesian products of semirings, none of the other published case studies has exploited this feature. Test runs show that most optimal parameter assignments for software-defined radio can be computed in a few seconds; only the most difficult constraint sets need more than one minute for the construction of the solutions.
2 Example Application

As the driving application for developing constraint solving mechanisms in Maude, we focus on communicating wireless devices or radios that optimize resource usage such as bandwidth and power with respect to given goals. This application is set in the broader context of policy and goal-based applications, in which devices cooperatively strive to achieve goals while satisfying policies that constrain possible solutions.

PAGODA (see http://pagoda.csl.sri.com) is a modular architecture for design of (partially) autonomous systems. A PAGODA node interacts with its environment by sensing and affecting, driven by goals to achieve and constrained by policies. A PAGODA system is a collection of PAGODA nodes cooperating to achieve some mutual goal. The PAGODA architecture was inspired by the study of architectures developed for autonomous space systems, especially the MDS architecture [12] and its precursors [15].

![Fig. 1. PAGODA node architecture](image)

Figure 1 shows the principal components of a PAGODA node: a knowledge base, a coordinator, a reasoner, a monitor, and a hardware abstraction layer. There is also a component for communication with a system operator (headquarters), and a component responsible for distributed coordination with other PAGODA nodes (distributed coordinator).

The knowledge base contains knowledge that is shared and updated by the remaining components. This knowledge covers a wide range of information including (1) goals a node or system is trying to achieve, (2) policies that constrain the actions or interactions a node or system is allowed to do, thus reducing the number of choices for setting parameters, and (3) a device model that specifies the parameters that can be set (knobs) and read (sensors) and their relationships.

The effects of knob settings can be observed in terms of improved connectivity, higher bandwidth and other observable results that are related to goals. For example, increasing the transmission power or choosing lower frequencies results in better connectivity between the radios. These relationships are defined in tables such as the one given in Figure 2 (left). Note that a certain knob setting, such as high frequencies for transmission, can have contradicting effects (decreased con-
The reasoner component determines proper knob settings so that goals are achieved or desired effects take place. The reasoner uses information from the knowledge base as a basis for its deductions: the device model—that is the relationships between knobs, effects, and sensor readings; the goals; the policies; and the current state. When new parameter settings are determined, the reasoner also provides justifications such as what sensor values and/or what relationships from the device model were used to infer the new settings. This can be used for diagnostics if things do not go as expected. The reasoner also specifies sensors that should be monitored and conditions of sensor readings that do not fulfill goals, so that the reasoner can take corrective action.

We developed a formal executable specification of the PAGODA architecture in the Maude language [7] and instantiated it with an abstract device model of a radio to test the ideas.

Goals are treated as soft constraints on subsets of sensor readings. The relationships between affectors (knobs) and sensor readings and between sensor readings and goals are formalized as constraint semi-rings, which provides a clean mathematical basis for solving soft constraints [4,10]. In particular, the sensor tables (see Figure 2 right) assign weights to knob settings. The sum of these weights indicates the benefit of a specific knob setting for a given sensor: the higher the sum of its weights, the better. The goal is to compute for a list of interesting sensors the valuations of the knobs that optimize the sums of these weights. We model this by the constraint semiring of fuzzy natural numbers for grading the weights. The effect constraints indicate the impact of a specific knob setting against a given effect (see Figure 2 left). The goal is to find knob settings that satisfy a maximum of required effects or equivalently violate a minimum of required effects. The best situation occurs if there are no violations of the effects [2]. We model this using the constraint

Fig. 2. Partial knob-effect (left) and partial knob-sensor (right) tables

nectivity but higher bandwidth). Similarly, the dependencies between knob settings and measurable sensor reading are also defined. Figure 2 (right) shows an excerpt defining the impact of choosing of a certain knob setting on a sensor reading. By definition, if the sum of weights within a column is greater than 50 (which is the case in the full table), then the combination of knob settings means that the corresponding sensor in the column reads “good.” Thus, high transmission power and low transmission frequency promotes an increased signal strength reading as well as good sensor readings for connectivity loss, whereas high transmission power and high transmission frequency promotes better readings for throughput.

We developed a formal executable specification of the PAGODA architecture in the Maude language [7] and instantiated it with an abstract device model of a radio to test the ideas.

Goals are treated as soft constraints on subsets of sensor readings. The relationships between affectors (knobs) and sensor readings and between sensor readings and goals are formalized as constraint semi-rings, which provides a clean mathematical basis for solving soft constraints [4,10]. In particular, the sensor tables (see Figure 2 right) assign weights to knob settings. The sum of these weights indicates the benefit of a specific knob setting for a given sensor: the higher the sum of its weights, the better. The goal is to compute for a list of interesting sensors the valuations of the knobs that optimize the sums of these weights. We model this by the constraint semiring of fuzzy natural numbers for grading the weights. The effect constraints indicate the impact of a specific knob setting against a given effect (see Figure 2 left). The goal is to find knob settings that satisfy a maximum of required effects or equivalently violate a minimum of required effects. The best situation occurs if there are no violations of the effects [2]. We model this using the constraint
We instantiated the PAGODA abstract device specification with a specification of a concrete radio, MadRad, that simulates actual radio hardware/software including random, unusual and faulty behavior. Test scenarios allow us then to explore some possible system behaviors.

## 3 C-Semirings and Soft Constraints in Maude

### 3.1 Brief Overview of Maude

Maude [7] is a multiparadigm executable specification language based on rewriting logic [14]. Maude sources, executables for several platforms, the manual, a primer, cases studies, and papers are available from the Maude Web site at [http://maude.cs.uiuc.edu](http://maude.cs.uiuc.edu).

We use *functional theories*, *functional modules* and *parameterized modules* in our framework. Functional modules are equational theories used to specify algebraic data types such as particular constraint semirings; they are declared with the syntax \texttt{fmod ... endfm}. Functional modules can have sorts, subsort relationships, operators, variables, membership axioms, and equations, and can import other theories or modules. Functional theories (syntax \texttt{fth ... endfth}) are similar to functional modules, they are used to declare module interfaces such as the axiomatic theory of constraint semirings; but opposed to functional modules they have a loose interpretation (as opposed to an initial algebra semantics of functional modules) and do not need to be executable (expressed by the attribute \texttt{nonexec}). Parameterized modules (available in Maude 2.2) are used to represent our generic approach to soft constraints. Such a module \texttt{P} has the syntax \texttt{fmod P\{X1 :: T1, ..., Xn :: Tn\} ... endfm} where \(X_i\) are formal parameters and \(T_i\) functional theories representing the type of the parameters \((i = 1, \ldots, n)\). A \texttt{view} \(V\) (written \texttt{view V from T to M is ... endv}) specifies how a particular target module \(M\) is claimed to satisfy a source theory \(T\). An instantiation \(P(V1, \ldots, V_n)\) of \texttt{P} requires a \texttt{view Vi} from the type \(T_i\) of each formal parameter \(X_i\) to a corresponding actual parameter \(M_i\) such that each \(M_i\) satisfies the axioms of \(T_i\) modulo the renaming specified by \(V_i\) (i.e. each \(V_i\) is a theory morphism).

### 3.2 Constraint Semirings

A semiring \(S\) is an algebra \(\langle G; +; *; 0; 1 \rangle\) with carrier set \(G\), two constants \(0, 1 \in G\) and two binary operations \(+, *\) having the following properties: \(+\) is commutative and associative and \(0\) is its neutral element; \(*\) is associative and distributes over \(+\), \(1\) is its neutral element and \(0\) is its absorbing element. A constraint semiring \(S\) (c-semiring for short) is a semiring \(\langle G; +; *; 0; 1 \rangle\) with the following two additional properties: \(*\) is commutative, and \(1\) is the absorbing element of \(+\). Then - as one of the anonymous referees observed - \(+\) is also idempotent. The addition induces a partial ordering relation \(\leq\) between the elements of \(G\) defined by \(a \leq b\) iff \(a + b = b\). This partial ordering will be used to compare solutions of constraints.
and to determine the best solution. If \( a \leq b \) we say also that \( b \) is better than \( a \). \( + \) and \( \ast \) are monotonic w.r.t. \( \leq \), 0 is the least element and 1 is the greatest element of \( G \); \( \langle G ; \leq \rangle \) forms a complete lattice with \( + \) as least upper bound.

Constraint semirings have important closure properties: they are closed under cartesian products and the formation of power sets. An ic-semiring is a constraint semiring where \( \ast \) is also idempotent. Then \( + \) distributes over \( \ast \) and \( \langle G ; \leq \rangle \) is a complete distributive lattice with \( \ast \) as its greatest lower bound.

We formalize semirings, constraint semirings, and ic-semirings as Maude theories in a straight-forward way. The sort is called Grade since its elements will be used to grade soft constraints. Note that the last axiom induces transitivity and symmetry of the equivalence relation.

\[
\begin{align*}
\text{fth SEMIRING is} & \quad \text{pr BOOL .} \\
\text{sort Grade .} & \quad \text{op zero : } \rightarrow \text{Grade . \ op one : } \rightarrow \text{Grade .} \\
\text{op } \_+\_ & \quad : \text{Grade Grade } \rightarrow \text{Grade [assoc comm id: zero prec 33] .} \\
\text{op } \_\ast\_ & \quad : \text{Grade Grade } \rightarrow \text{Grade [assoc id: one prec 31] .} \\
\text{op } \_\leq\_ & \quad : \text{Grade Grade } \rightarrow \text{Bool [prec 37] .} \\
\text{op } \_\text{equiv}_\_ & \quad : \text{Grade Grade } \rightarrow \text{Bool [prec 37] .} \\
\text{vars X Y Z : Grade .} & \quad \text{eq X } \ast (Y + Z) = (X \ast Y) + (X \ast Z) \ [\text{nonexec}] . \\
\text{eq X } \ast \text{ zero } = \text{ zero } & \quad \ [\text{nonexec}] . \\
\text{eq X } \leq Y = (X + Y) == Y & \quad \ [\text{nonexec}] . \\
\text{eq X equiv X = true} & \quad \ [\text{nonexec}] . \\
\text{ceq X equiv Z = true} & \quad \text{if X equiv Y = true } \setminus \text{ Z equiv Y = true } \ [\text{nonexec}] . \\
\end{align*}
\]

The cartesian product of two c-semirings forms a c-semiring whose operations are defined in the obvious componentwise way. The induced ordering relation \( \leq_x \) is a partial ordering. We also introduce the lexicographic ordering \( \leq_x \times \) for applications with a preferred component. The lexicographic ordering is a total ordering which extends \( \leq_x \times \); it is also well-suited for the radio application where the optimization of the effects is preferred to the optimization of the minimal sensor value. The cartesian product is modelled as a parameterized module with two constraint semirings as parameters.

\[
\begin{align*}
\text{fth C-SEMIRING is} & \quad \text{inc SEMIRING .} \\
\text{vars X : Grade .} & \quad \text{eq X + one = one \ [nonexec] .} \\
\end{align*}
\]

\[
\begin{align*}
\text{fmod PAIR{X :: C-SEMIRING, Y :: C-SEMIRING} is} & \quad \text{protecting BOOL .} \\
\text{sort Pair(X, Y) .} & \quad \text{op pair : X$Grade Y$Grade } \rightarrow \text{Pair(X, Y) [ctor] .} \\
\text{op zero : } \rightarrow \text{Pair(X, Y) . eq zero = pair(X$zero, Y$zero) .} \\
\text{op one : } \rightarrow \text{Pair(X, Y) . eq one = pair(X$one, Y$one) .} \\
\end{align*}
\]
3.3 Boolean, Fuzzy, and Weighted Sum Semirings

Examples of constraint semirings are boolean algebras, and in particular, \( \text{Bool} = \langle \{ \text{false}; \text{true} \}; \lor; \land; \text{false}; \text{true} \rangle \), “fuzzy algebras” such as “fuzzy natural numbers” \( F_N = \langle \mathbb{N} \cup \{ \infty \}; \max; \min; 0; \infty \rangle \), and weighted sum algebras, e.g. over natural numbers \( W_N = \langle \mathbb{N} \cup \{ \infty \}; \min; +; \infty; 0 \rangle \). Except for weighted sums, all of these algebras are ic-semirings. In Boolean algebras, \( \text{true} \) is better than \( \text{false} \); for fuzzy natural numbers, \( 0 \) is the least and \( \infty \) the greatest element, whereas the ordering for weighted sum is the converse to the usual ordering: \( \infty \) is the least element and \( 0 \) the greatest element.

In Maude, fuzzy natural numbers are specified as functional module in the following way. We introduce the sort \( \text{NatFN} \) of fuzzy natural numbers with infinity and two subsorts \( \text{IftyFN} \) and \( \text{NiNatFN} \) for representing the \( \infty \) element and the “non-infinity” natural numbers, which are constructed by an embedding of natural numbers.

```plaintext
fmod FUZZYNAT is
  pr NAT .
  sorts NiNatFN IftyFN NatFN .
  subsort NiNatFN IftyFN < NatFN .
  op iftyFN : -> IftyFN [ctor] .
  op _+_ : NatFN NatFN [prec 33].
  eq fn(M) + fn(N) = fn(max(M, N)) .
  eq fn(M) + iftyFN = iftyFN .
  op _*_ : NatFN NatFN [prec 33].
  eq fn(M) * fn(N) = fn(min(M, N)) .
  eq fn(N) * iftyFN = fn(N) .
  eq iftyFN * fn(N) = fn(N) .
  eq iftyFN * iftyFN = iftyFN .
  vars N M : Nat .
  var U : NatFN .
  op _<=_ : NatFN NatFN -> Bool .
endfm
```

Then, we define a view from \( \text{C-SEMIRING} \) to \( \text{FUZZYNAT} \) and can easily prove by structural induction that all axioms are satisfied, i.e., the view forms a theory morphism.

```plaintext
view FuzzyNat from C-SEMIRING to FUZZYNAT is
  sort Grade to NatFN .
endv
```
The constraint *semiring of weighted sums* $WSUM$ over natural numbers with infinity and the theory morphism $WSum$ from $C\text{-SEMIRING}$ to the weighted sum algebra are specified in an analogous way.

In our application to software-defined radios we use the cartesian product of the above constraint semirings on natural numbers with infinity. It is defined by instantiating the parameterized module $PAIR$ using the views $WSum$ and $FuzzyNat$. The cartesian product forms also a c-semiring which is expressed by the view $WSumFuzzyPair$.

```maude
fmod WSUMFUZZYPAIR is
  pr PAIR{WSum, FuzzyNat} .
endfm
view WSumFuzzyPair from C\text{-SEMIRING} to WSUMFUZZYPAIR is
  sort Grade to Pair{WSum, FuzzyNat} .
endv
```

### 3.4 Soft Constraints

A soft constraint assigns grades to different valuations of a set of problem variables. Let $S$ be a constraint semiring with carrier set $G$, $D$ a finite problem domain, and $Var$ the set of all problem variables. A *valuation* $\nu : Var \longrightarrow D$ is a partial map from $Var$ to $D$ which has a finite support. Given an ordered list $al \in Var$ of variables, a soft constraint assigns a grade in $G$ to each possible valuation of variables in $al$; more formally, a *soft constraint* is a pair $\langle al; cst \rangle$ where $cst \in (Var \longrightarrow D) \longrightarrow G$ is a mapping from valuations to elements of $G$ such that every valuation of the domain of $cst$ has finite support $al$. As the set of all such valuations is finite, any soft constraint has a finite domain of definition and can therefore be represented either in an *explicit* way by a finite set of pairs $(\nu \mapsto g)$ with $\nu \in (al \longrightarrow D)$ and $g \in G$ or in a more *implicit* way by particular constructors or function declarations as in a programming language. In our framework we support both possibilities; here we present only the explicit form and a particular implicit form we need for our application to software-defined radios.

For representing valuations in Maude, we define a total ordering on the set $Var$ of all variables. The module $VALUATION$ is parameterized by theories for an ordered set $X$ of variables and an ordered domain $D$ (using the theory $TAO\text{-SET}$ provided in the Maude prelude); each valuation $\nu = (a_1 \mapsto d_1, \ldots, a_k \mapsto d_k)$ is represented by a pair $[al \mapsto dl]$ consisting of the ordered list $al = a_1, \ldots, a_k$ of variables and the list $dl = d_1, \ldots, d_k$ of corresponding elements of $D$. For fixed $al$, $(al \longrightarrow D) \longrightarrow G$ is isomorphic to $D^k \longrightarrow G$ and thus any constraint definition $cst$ can be represented as a map from $D^k$ to $G$. In Maude we specify such maps by a parameterized module $LC\text{-MAP}$ similar to the $MAP$ module of the Maude prelude, except that the range is a constraint semiring $S$ and, for efficiency reasons, the domain of any map consists of a *list of elements* of $D$. As in $MAP$, we introduce a parameterized sort $LC\text{-Map}$ for maps and a subsort for simple entries of the form $(dl \mapsto g)$; for reasons of space efficiency, we omit entries $(dl \mapsto 0)$ from the $LC\text{-Map}$ terms. Moreover, the specification contains efficient operations for sorting maps according to the values of different lexicographic orderings of the domain and according to the values of the codomain. $SORTABLE\text{-LIST1}$ is an extension of $SORTABLE\text{-LIST}$ (see the Maude 2.2 prelude) by an operation $\text{noDupmerge}$ for merging two ordered lists without duplicating elements.
fmod Valuation{X :: TAO-SET, D :: TAO-SET} is
pr EXT-BOOL . pr SORTABLE-LIST1(X) . pr SORTABLE-LIST1(D) .
sort Valuation(X, D) .
op [ _ |-> _ ] : List(X) List(D) -> Valuation(X, D) [ctor] .
op varsibs : Valuation(X, D) -> List(X) .
op values : Valuation(X, D) -> List(D) .
op consistent : Valuation(X, D) Valuation(X, D) -> Bool .
*** checks the equality of the values of the common variables
*** of both valuations.
op mergeEntry : Valuation(X, D) Valuation(X, D) -> List(D) .
*** merges the values of two consistent valuations.
...
endfm

fmod LC-MAP{D :: TAO-SET, S :: C-SEMIRING} is
  protecting SORTABLE-LIST{D} .
sorts LC-Entry{D,S} LC-Map{D, S} .
subsort LC-Entry{D,S} < LC-Map{D, S} .
op empty : -> LC-Map{D,S} [ctor] .
op ( _|->_ ) : List(D) S$Grade -> LC-Entry{D,S} [ctor] .
op __ : LC-Map{D, S} LC-Map{D, S} -> LC-Map{D, S} .
  [ctor assoc id: empty] .
...
op sortCoDomain : LC-Map{X, Y} -> LC-Map{X, Y} .
...
endfm

The specification CONSTRAINT is parameterized by theories X for Variables, D for the problem domain, and S for the c-semiring. It has parameterized sorts for constraints and list of constraints, and subsorts EConstraint and IConstraint for explicit and implicit constraints as well as the sort ZeroConstraint for the constraint with constant grade 0 and the sort NoConstraint for the situation with no constraint at all. The explicit representation of a soft constraint has the form

\[ [al \mid (val_1 \mapsto g_1), ..., (val_m \mapsto g_m)] \]

where all grades \(g_1, ..., g_m\) are different from 0 (as in LC-MAP). The operations of c-semirings and the operations of LC-MAP are lifted to constraints as follows. The application of a constraint \(c = [al \mid cst]\) to a valuation \(v\) is defined as map application: \(c[v] = cst[v]\); zeroConstraint and noConstraint define the constraints with constant values zero and one; constraint multiplication is defined to satisfy the equation \(c_1 \cdot c_2[v] = c_1[v] \cdot c_2[v]\), and constraint addition is defined analogously. Then it is easy to prove that the set of constraints forms again a c-semiring [5].

Moreover, we define a projection operator \(\text{project}\) and (for constraint propagation) a partial evaluation operator \(\text{peval}\). For any constraint \(c = [al \mid cst]\) and any valuation \(v = [bl \mapsto dl]\) with \(bl \subseteq al\), \(\text{peval}(c, v)\) restricts \(c\) to the valuations consistent with \(v\); in particular, for \(bl = al\) we have \(\text{peval}(c, v) = [al \mid (dl \mapsto cst[v])]\). For a constraint \(c\) and an ordered list of variables \(xl\), \(\text{project}(c, al)\) computes the sum \(\sum_{dl \in Dk} \text{peval}(c, [al \mapsto dl])\) of all possible partial evaluations of \(al\). For efficiency, all operations on constraints are specified in a tail recursive way. As an example, we show the specification of constraint multiplication.

fmod CONSTRAINT{X :: TAO-SET, D :: TAO-SET, S :: C-SEMIRING} is
Wirsing et al.

protecting EXT-BOOL . protecting NAT . protecting SORTABLE-LIST1(X) .
protecting VALUATION(X, D) . protecting LC-MAP(D, S) .
sorts NoConstraint(X, D, S) ZeroConstraint(X, D, S)
  SimpleConstraint(X, D, S) EConstraint(X, D, S)
  Constraint(X, D, S) ListConstraint(X, D, S) .
subsort NoConstraint(X, D, S) ZeroConstraint(X, D, S)
  SimpleConstraint(X, D, S) Constraint(X, D, S) .
  Constraint(X, D, S) ListConstraint(X, D, S) .

op noConstraint : -> NoConstraint(X, D, S) [ctor] .
op zeroConstraint : -> ZeroConstraint(X, D, S) [ctor] .
op [____] : List(X) LC-Entry(D, S) -> SimpleConstraint(X, D, S) [ctor] .
op [____] : List(X) LC-Map(D, S) -> EConstraint(X, D, S) [ctor] .
op _+_ : Constraint(X, D, S) Constraint(X, D, S) -> Constraint(X, D, S) .
op _*_ : Constraint(X, D, S) Constrain(X, D, S) -> Constraint(X, D, S) .
op |_| : Constrain(X, D, S) Valuation(X, D) -> S$Grade .
op peval : EConstraint(X, D, S) Valuation(X, D) -> EConstraint(X, D, S) .
op project : EConstraint(X, D, S) List(X) -> EConstraint(X, D, S) .

... vars AL BL : List(X) . vars VL WL L : List(D) .
vars EN1 EN2 : LC-Entry(D, S) . vars M M1 M2 Result : LC-Map(D, S) .
vars P Q : S$Grade . var C : Constraint(X, D, S) .

eq noConstraint * C = C . eq [AL | M] * noConstraint = [AL | M] .
eq [AL | M] * zeroConstraint = zeroConstraint .
eq [AL | M] * zeroConstraint = zeroConstraint .
eq [AL | empty] * [BL | M] = [noDupMerge(AL, BL) | empty] .
eq [AL | (VL |-> P) M1] * [BL | M2] =
  if P =/= zero
  then [noDupMerge(AL, BL) | [AL | (VL |-> P) M1] *Map [BL | M2]]
  else [noDupMerge(AL, BL) | [AL | M1] *Map [BL | M2]]

op _*Map_ : SimpleConstraint(X, D, S) -> LC-Map(D, S) .
eq [AL | (VL |-> P) ] *Map [BL | (WL |-> Q)] =
  if (P * Q =/= zero) and-then consistent([AL |-> VL], [BL |-> WL])
  then (mergeEntry([AL |-> VL], [BL |-> WL]) |-> P * Q)
  else empty

op _$*Map_with_is_ : EConstraint(X, D, S) LC-Map(D, S) -> LC-Map(D, S) .
eq [AL | M1] *Map [BL | M2] =
  [AL | M1] $*Map [BL | M2] with M2 is empty .
eq [AL | EN1 M1] $*Map [BL | EN2 M2] with M is Result =
  [AL | EN1 M1] $*Map [BL | M2] with M is Result =
  [AL | EN1 M1] $*Map [BL | M] with M is Result = Result .
eq [AL | empty] $*Map [BL | M2] with M is Result = Result .

... endfm

3.5 Hard Constraints and Implicit Constraints

Hard constraints can be considered as a special class of soft constraints: those over the
two-valued boolean semiring $\text{Bool} = \{\text{false; true}\}; \lor; \land; \text{false}; \text{true}\$ where $\text{true}$
indicates the satisfaction of a hard constraint and $\text{false}$ its violation [4]. Often hard and soft
constraints occur together in the same problem. Let $S$ be the c-semiring of the soft con-
constraints. Then there are several possibilities for encoding hard constraints:

- Choose $S$ as c-semiring and represent satisfaction of a hard constraint by $1_S$ and its violation by $0_S$. This yields non-zero grades for consistent combinations of soft and hard constraints that satisfy all hard constraints; if one of the hard constraints is violated the resulting weight is $0_S$.

- Build the cartesian product $\text{Bool} \times S$ with the lexicographic ordering. Then satisfaction of a hard constraint is expressed by $\langle 1_{\text{Bool}}, g \rangle$ and violation by $\langle 0_{\text{Bool}}, g \rangle$ for some $g \in S$. This gives us a finer grained analysis of hard constraints violation: one can not only distinguish the grades of those combinations of constraints that satisfy all hard constraints, but also the grades of those soft constraints that violate one of the hard constraints.

For our radio application, we choose a refinement of the latter solution: The c-semiring is the lexicographically ordered cartesian product $\text{WSUMFUZZYPAIR}$ of weighted sum and fuzzy naturals (see Section 3.3). Hard constraints of the form $n \geq \text{constant}$ occur for the sensor constraints, which have values in the c-semiring of “fuzzy natural number constraints”. We represent satisfaction of such a hard constraint by $\langle 1_{\text{WSum}}, n \rangle$ and violation by $\langle 0_{\text{WSum}}, n \rangle$, thus giving also information about the grades of the sensor constraint.

In general, implicit constraints are defined by structural recursive expressions and by particular application-dependent constructors. As operations, we provide partial evaluation and transformation into explicit constraints.

For the radio application, we use the sensor tables as a space-efficient representation of the sensor constraints. The naive representation of the sensor values by summation leads to constraints with many variables and therefore a potentially exponential number of entries; e.g., the explicit constraint for the “signal strength” has the form (with $\text{TP}$ for transmission power, $\text{TF}$ for transmission frequency, and $\text{CP}$ for compression):

\[
\begin{align*}
\text{TP} \text{TF} \text{CP} \ldots | (\text{Hi} \mapsto 50)(\text{Lo} \mapsto 0)\ldots
\end{align*}
\]

Instead we use an implicit constraint (with constructor $\text{sm}$ for ”sensor map”) for a list representation which requires only linear space:

\[
\begin{align*}
\text{sm}([\text{TP} | (\text{Hi} \mapsto 50)(\text{Lo} \mapsto 0)] [\text{TF} | (\text{Hi} \mapsto 0)(\text{Lo} \mapsto 50)] \ldots)
\end{align*}
\]

During the process of constraint solving we always try to apply partial evaluation as much as possible in order to reduce the size of implicit constraints before transforming them into explicit constraints.

### 4 Solving Soft Constraints

The solution space of a list of soft constraints $cl = cl_1 \ldots cl_n$ is given by the multiplication $cl_1 \ast \ldots \ast cl_n$ of all constraints; the result is again a soft constraint $c$ which can be represented as a sum $sc_1 + \ldots + sc_m$ of simple constraints (where $m \leq |D|^{\text{variabs}(C)}$). Each such simple constraint is called a possible solution of $C$. Not all of these possible solutions are of interest for applications. In the following we write $\prod cl$ as a shorthand for $cl_1 \ast \ldots \ast cl_n$ and $\sum cl$ for $cl_1 + \ldots + cl_n$

In many cases we search for the set $\text{maxSol}(cl)$ of all best solutions, i.e., all elements of $D^{\text{variabs}(C)}$ with maximal grades in the solution space, or for the solutions with a grade better than a certain threshold. One can also be more general and compute the projection
project(cl₁∗...∗clₙ, xl) to a subset xl of the variables, if only these variables are of interest and then consider the same classes of solutions.

For the radio application, we need algorithms for both: for finding one admissible solution in a short time and, in cases where enough computing power and time are available, for finding all best solutions or all admissible solutions. But we could simplify the job in that all variables (knobs) were of interest and the involved semirings were all totally ordered. Following, we make therefore these simplifying assumptions and present a Maude realisation for searching all best solutions and give a sketch of the implementation of the other search algorithms.

The implementation consists of the following three parameterized modules: a module SOLVECONSTRAINT for the two search algorithms and two modules SOLUTION and CONTINUATION for representing the data type of solutions and storing the necessary backtracking information. The module SOLUTION provides explicit informations about all solutions, their grade and their number; in particular, solution(scl, g, n) is a constructor term consisting of a list scl of simple constraints denoting solutions, the grade g of all elements of scl, and the number n = |scl| of solutions. Continuations are defined by the module CONTINUATION; a continuation cl(0, sc) consists of a partial solution sc and a list cl of unsolved constraints with the intended property that the combination sc * [cl] of sc with all constraints in cl forms again a set of possible solutions of the original constraint problem cl.

The module SOLVECONSTRAINT defines depth-first search algorithms for finding all best solutions, for finding a first solution better than a certain threshold, and for finding all such solutions (see [3] for a similar approach). For any list cl, search(cl) computes the set maxSol(cl) of all best solutions. For efficiency, it assumes for all constraints [al | (vl₁ ↦ g₁), ..., (vlₙ ↦ gₙ)] in cl that all grades g₁, ..., gₙ are in descending order. search uses an auxiliary tail-recursive operation $search(cl, sc, cont, csol)$ where cl denotes the list of constraints to be solved, sc the actual partial solution, cont the continuation, and csol = solution(scl, b, n) the constraint solution obtained so far.

The most interesting case of the recursive definition is cl = [al | (vl ↦ p)erest] rest and sc = [bl | (vl ↦ q)] where vl has grade p, erest denotes the map consisting of the remaining assignments of the first constraint of cl, and rest denotes the tail of the list of constraints cl. (All other cases are simpler with erest, rest or cl being empty, or sc corresponding to noConstraint.)

If the grade p * q of the new solution is not smaller than b and different from zero and if the first subconstraint c₀ = [al | (vl ↦ p)] of cl is consistent with sc then $search$ computes a new partial solution [al | (vl ↦ p)] * sc and saves the rest of the constraint in the continuation for later backtracking. Otherwise, if p * q is large enough but c₀ not consistent with sc, the search continues with erest; finally, if p * q < b, we can use the fact that every constraint is sorted in a descending order of grades which implies that for all grades g of the entries of erest we have also g * q < b; therefore the partial solution sc cannot be completed to a best solution and the algorithm backtracks with the continuation.

\[
\text{fmod SOLVECONSTRAINT} \{X :: \text{TAO-SET}, D :: \text{TAO-SET}, S :: \text{C-SEMIRING}\} \text{ is}
\]

\[
\text{pr CONSTRAINT}(X, D, S) . \text{pr CONTINUATION}(X, D, S) .
\]

\[
\text{pr SOLUTION}(X, D, S) .
\]

\[
\text{var N : Nat} . \text{vars AL BL : List}(X) . \text{vars VL WL : List}(D) .
\]

\[
\text{var ERest : LC-Map(D, S)} . \text{vars P Q B : S$Grade} .
\]

\[
\text{var SC : SimpleConstraint}(X, D, S) .
\]

234
It is easy to see that the search algorithm is terminating. The following lemma is the basis for the correctness proof of the search algorithm.

**Lemma (search invariant).**

For all lists of explicit constraints \( cl \), all partial constraints \( sc \), all lists of continuations \( cont \) of the form \( ct(cont_1, scont_1), \ldots, ct(cont_k, scont_k) \), and all solutions \( csol \) of the form \( solution(scl, b, n) \) where \( scl \) is a list of \( n \) of possible solutions with grade \( b \), i.e., \( scl \) is a list of simple constraints of the form \( scl_i = [x_l \mid (w_{il} \mapsto b)] \), the following holds:

1. \( \$backtrack(cont, csol) = maxSol(\sum_{j=1}^{k}(scont_j \cdot \prod contl_j) + \sum scl) \);
2. \( \$search(cl, sc, cont, csol) = maxSol(sc \cdot \prod cl + \sum_{j=1}^{k}(scont_j \cdot \prod contl_j) + \sum scl) \).

**Proof.** By simultaneous computational induction on both operations.

The correctness of the search operation follows directly from termination and the invariant lemma:

**Theorem (total correctness of search)**

For all lists of explicit constraints \( cl \), the following holds:

\( search(cl) = maxSol(cl) \).
Proof.

\[ \text{search}(cl) = \]
\[ \maxSol(\text{noConstraint} \cdot \prod cl + \text{zeroConstraint} + \text{zeroConstraint}) = \]
\[ \maxSol(\prod cl), \text{where the second equality uses the lemma.} \]

5 Experimentation Results

One objective of the radio application is to determine radio parameters for a networked radio team such that the mission goals are satisfied. We formalize this problem as a soft constraint problem, implement it in our Maude soft constraint framework and integrate it with the PAGODA system for experiments.

The sensor tables (see Figure 2 right) assign weights to knob settings. The goal is to compute for a list of interesting sensors the valuations of the knobs that optimize the sums of these weights (see Section 2). Thus, we choose the c-semiring of fuzzy natural numbers for grading the weights. Given a sensor \( s \), a list of knobs \( knl = k_{n1}, ..., k_{nm} \), a valuation \( v = (k_{n1} \mapsto d_{1}, ..., k_{nm} \mapsto d_{m}) \) into a domain \( \text{KnobVal} \) of knob values and associated weights \( g_{s,1}, ..., g_{s,m} \), a sensor constraint \( c_{s} \) has the form \( (knl; \text{cst}_{s} : \text{KnobVal} \rightarrow \text{NatFN}) \) where \( \text{cst}_{s}(d_{1}, ..., d_{m}) = \sum_{i=1}^{m} g_{s,i} \). For a list \( sl \) of sensors we maximize the minimal value of these sums, i.e., we compute \( \max \{ d | \min_{s \in sl} \text{cst}_{s}(d) \} \) for all \( d \in \text{KnobVal} \).

The effect constraints indicate the impact of a specific knob setting against a given effect (see Figure 2 left). The more “+”s are in the column of an effect, the more likely it is that the knob setting will achieve the effect. The goal is to find knob settings that violate a minimum of required effects (see Section 2). We model this using the c-semiring of weighted sums by assigning grade one to “+” and grade zero to “-”. For a given effect \( e \), any knob \( k_{ni} \), value \( d_{i} \) and associated grade \( g_{e,i} \), we define a soft constraint \( (k_{ni}; \text{cst}_{e,i} : \text{KnobVal} \rightarrow \text{NatWS}) \) where \( \text{cst}_{e,i}(d_{i}) = g_{e,i} \). For a list \( sl \) of effects we minimize the sums of these values, i.e., we compute \( \min \{ d | \sum_{e \in el} \sum_{i \in \{1,...,m\}} \text{cst}_{e,i}(d_{i}) \} \).

For the radio application, we perform a multicriteria optimization by minimizing the violations of the effect goals and maximizing the benefits of the knob settings where we give preference to the optimization of the effect goals. Formally, this means to build the lexicographically ordered cartesian product of the semirings of weighted sums and of fuzzy natural numbers (modelled by \( \text{WSUMFUZZYPAIR} \)). Note that by exchanging the order of the two semirings we could give also priority to the benefits of the knob settings.

To model this application in Maude, we instantiate the constraint framework with the data types of the radio application and integrate it with PAGODA.

For representing knobs, we use the existing sorts \( \text{Knob} \) and \( \text{KnobVal} \) of the PAGODA system. \( \text{Knob} \) defines a list of 14 constants such as \( \text{TP}, \text{TF}, \text{Cp}, \text{ECC} \) representing transmission power, transmission frequency, compression, and error correction code. \( \text{KnobVal} \) has in the current implementation only two values \( \text{Hi} \) and \( \text{Lo} \). Effect constraints are defined as combinations of simple one-variable constraints with values in the left pair component; e.g., the effect constraint \( \text{ImprovedConnectivityCstr} \) for the goal "Improved Connectivity" which requires sensor values \( \text{TP} = \text{Hi}, \text{TF} = \text{Lo}, ... \) is represented by the product of one-variable constraints:

\[ [\text{TP} | (\text{Hi} \mapsto 0) (\text{Lo} \mapsto 1)] \ast [\text{TF} | (\text{Hi} \mapsto 1) (\text{Lo} \mapsto 0)] \ast ... \]
Sensor constraints are represented by implicit constraints as described in section 3.5. As a hard constraint, we require that sensor weights have a value \( n \) greater than a certain threshold \( \alpha \). It is represented by \((1_{WS_{\text{sum}}}, n)\) for \( n \geq \alpha \) and by \((0_{WS_{\text{sum}}}, n)\) for \( n < \alpha \).

For integrating this instantiation of the soft constraint framework with PAGODA it suffices to call one of the search functions within the reasoner component. E.g., a call 

\[
\text{search(ImprovedConnectivityCstr HighBWCstr ...)}
\]

returns two best solutions each of which has four effect inconsistencies and a value 60 for the minimal sum of the weights of the sensors:

\[
\text{rewrites: 652800 in 4640438906ms cpu (2142ms real) (0 rewrites/second)}
\]

\[
\text{result Solution{Knob,KnobVal,WSumFuzzyPair}: solution{}
\]

\[
\text{[TP TF PktSize Cp TW ECC RU Encr Cach SS DT CS}
\]

\[
\text{| Lo Lo Hi Hi Lo Lo Hi Lo Lo Lo Lo Lo Lo Lo |-> pair(ws(4), fn(60))}
\]

\[
\text{[TP TF PktSize Cp TW ECC RU Encr Cach SS DT CS}
\]

\[
\text{| Lo Hi Hi Hi Hi Lo Lo Lo Lo Lo Lo Lo Lo |-> pair(ws(4), fn(60))},
\]

\[
\text{pair(ws(4), fn(60)), 2)}
\]

Searching just one solution with the same threshold (using \text{searchOne}()) is typically at least 10 times faster. The following table gives an overview on the experimentation results which were performed using an Intel Pentium M 713 CPU 1.1 GHz, 512 MB RAM notebook running Core Maude Version 86a over Windows XP. The first column lists the number of sensors and effects under consideration, the second and third column indicate the CPU time consumption for finding all best solutions and the first best solution.

The table shows that most optimal parameter assignments for software-defined radio can be computed in a few seconds; only the most difficult constraint sets need more than one minute for the construction of the solutions.

<table>
<thead>
<tr>
<th>no. constraints</th>
<th>duration \text{search(...)}</th>
<th>durations \text{searchOne}(..., \alpha)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0,22 sec.</td>
<td>0,02 sec</td>
</tr>
<tr>
<td>6</td>
<td>0,26 sec.</td>
<td>0,03 sec</td>
</tr>
<tr>
<td>8</td>
<td>0,58 sec.</td>
<td>0,03 sec</td>
</tr>
<tr>
<td>12</td>
<td>3,31 sec.</td>
<td>0,14 sec</td>
</tr>
<tr>
<td>17</td>
<td>82,61 sec.</td>
<td>1,49 sec</td>
</tr>
</tbody>
</table>

Table 1: experimentation results

6 Future Directions and Concluding Remarks

In this paper, we presented a Maude framework for prototyping of soft constraints à la Bistarelli, Montanari and Rossi. The framework is written in a modular and parameterized way and easily instantiable to different applications. As a case study, we integrated our framework with SRI’s PAGODA architecture for autonomous systems and instantiated it by an application to software-defined radios. Experimentation shows that the constraint solver is well-suited for prototyping and has acceptable performance for (rather simple) constraint problems.
For more complex (and realistic) models, we need to extend our framework in several directions. In order to increase performance, we plan to use precomputed solutions for standard scenarios and to do heavy computation of optimal solutions only in non-standard situations. In order to get more flexibility for selecting and evaluating constraints, it would be useful to develop a strategy language similar to [6] for designing specialized constraint solvers. Moreover, simple soft constraints as in this paper will not be enough. More complex policies for tasks such as resource allocation and providing quality of service have to be respected and solutions have to be compared using preferences. Based on the successful c-semiring-based approaches to preferences [11], quality of service [10] and soft concurrent constrained programming [5], we are currently developing an expressive modelling language for policies involving preferences and hard/soft constraints which will also be based on the c-semiring approach.

Acknowledgment
Thanks to the anonymous referees for their helpful suggestions and remarks.

References


Maude: What’s new in 2.2?

Steven Eker

SRI International, Menlo Park, CA, USA

Abstract

We will demonstrate new features introduced with the recent 2.2 release including Core Maude support for parameterized modules, predefined container data types, random number generation, counters, and the builtin linear Diophantine equation solver. We will also preview features planned for Maude 2.3.
How to make easy, trustable and modular implementation of calculi/rewriting systems.
Presentation of Tom illustrated on an implementation of the explicit rewriting calculus.

Germain Faure and Antoine Reilles

LORIA, FR

Abstract
Following the experience of ELAN [2], the TOM language was developed to provide rewrite tools for implementation of calculi, for compilation and for XML-transformations. We will focus here on the former. TOM provides a language to define a syntax (a signature) embedded into Java. Then, we can perform pattern matching with support of associative matching modulo neutral element (also known as listmatching). Finally, we can guide the application of rules with a strategy language defining term traversals (namely evaluation/rewriting strategies). The originality of TOM is the combination of formal aspects with a general purpose language (such as Java). This combination leads to an agile language. At the same time, the strategy language inspired by Elan and Stratego [3] gives the opportunity to reduce the code written in the general purpose language (and thus increase the formal parts). We will illustrate the presentation by an implementation of the explicit rewriting calculus, introduced at the last WRLA [1]. This running example will demonstrate the adequacy of TOM for such a development, offered by the integration in a general purpose language and by the strategy language.

References


This paper is electronically published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
strategies (system description). In 12th International Conference on Rewriting
Techniques and Applications, volume 2051 of Lecture Notes in Computer Science,
MOMENT-OCL:
Algebraic Specifications of OCL 2.0 within the
Eclipse Modeling Framework

Artur Boronat, Joaquín Oriente, Abel Gómez, José Carsí, Isidro Ramos
U. Politénica de Valencia, Spain

Abstract

Model-Driven Development is a field in Software Engineering that, for several
years, has been representing software artifacts as models in order to improve produc-
tivity, quality, and economic incomes. Models provide a more abstract description
of a software artifact than the final code of the application. Interest in this field has
grown in software development companies such as the Model-Driven Architecture
(MDA), supported by OMG, and the Software Factories, supported by Microsoft,
ensuring a model-driven technology stock for the near future.

Model-Driven Development has evolved to the Model-Driven Engineering field,
where not only design and code generation tasks are involved, but also traceability,
model management, meta-modeling issues, model interchange and persistence, etc.
To fulfill these tasks, model transformations and model queries are relevant tasks
that must be solved. In the MDA context, they are dealt from an open-standard
point of view. The standard Meta-Object Facilities (MOF) provides a way to de-
fine meta-models. The standard proposal Query/Views/Transformations (QVT)
indicates how to provide support for both transformations and queries. In QVT,
while new languages are provided for model transformation, the Object Constraint
Language (OCL) remains as the best choice for queries.

OCL is a textual language that is defined as a standard “add-on” to the UML
standard. It is used to define constraints and queries on UML models, allowing the
definition of more precise and more useful models. It can also be used to provide sup-
port for meta-modeling (MOF-based and Domain Specific Meta-modeling), model
transformation, Aspect-Oriented Modeling, support for model testing and simul-
ation, ontology development and validation for the Semantic Web, among others.
Despite its many advantages, while there is wide acceptance for UML design in
CASE tools, OCL lacks a well-suited technological support.

In this demo, we present the MOMENT-OCL tool, which integrates an algebraic
specification of the operational semantics of part of the OCL 2.0 standard into
the Eclipse Modeling Framework (EMF). EMF is a modeling environment that is plugged into the Eclipse platform and that provides a sort of implementation of the MOF. EMF enables the automatic importation of software artifacts from heterogeneous data sources: UML models, relational schemas, and XML schemas. In MOMENT-OCL, OCL queries and invariants can be executed over instances of EMF models in Maude. An interesting feature of this algebraic specification of the OCL 2.0 is the use of the parameterization to reuse the OCL specification for any meta-model/model and the simulation of higher-order functions for the sake of the reuse of collection operator definitions.
Abstract

In this presentation, we introduce the ITP/OCL tool, a rewriting-based tool that supports automatic validation of static class diagrams with respect to OCL constraints. The ITP/OCL tool is directly based on the equational specification of UML+OCL class diagrams. It is written entirely in Maude making extensive use of its reflective capabilities. We also give notice of the Visual ITP/OCL, a Java graphical interface that can be used as a front-end for the ITP/OCL tool.
Web ITP Tool Server:
A Web-Based Interface for the ITP Tool

Adrian Riesco
Facultad de Informática, Universidad Complutense, Madrid, Spain

Abstract

The ITP tool is an experimental inductive theorem prover for proving properties of (Diet) Maude equational specifications, i.e., specifications in membership equational logic. The Web ITP tool is a client-server application that allows a web-based interaction with the ITP tool. It includes a (Diet) Maude module editor, an ITP formula editor, and an ITP command editor. The Web ITP tool has been developed as an educational project: it aims to provide an ITP-based tool for teaching specification, validation and verification of abstract data types.