Preface

This volume contains the preliminary proceedings of the 14th International Workshop on Rewriting Logic and its Applications (WRLA 2022), held as a satellite event of the European Joint Conferences on Theory and Practice of Software (ETAPS 2022) in Munich, Germany, on April 2nd and 3rd 2022.

Rewriting logic is a natural model of computation and an expressive semantic framework for concurrency, parallelism, communication, and interaction. It can be used for specifying a wide range of systems and languages in various application fields. It also has good properties as a metalogical framework for representing logics. Over the years, several languages based on rewriting logic have been designed and implemented. The aim of the workshop is to bring together researchers with a common interest in rewriting logic and its applications and to give them the opportunity to present their recent works, discuss future research directions, and exchange ideas.

The previous meetings were held in Asilomar (USA) 1996, Pont-à-Mousson (France) 1998, Kanazawa (Japan) 2000, Pisa (Italy) 2002, Barcelona (Spain) 2004, Vienna (Austria) 2006, Budapest (Hungary) 2008, Paphos (Cyprus) 2010, Tallinn (Estonia) 2012, Grenoble (France) 2014, Eindhoven (Netherlands) 2016, Thessaloniki (Greece) 2018, and online as a virtual event in 2020.

We received 13 submissions. Each was reviewed by at least three program committee members. After extensive discussions, the program committee decided to accept 11 papers for presentation at the workshop. This volume also includes the abstracts of the invited talks, given by Gwen Salaün and Sebastian Moldersheim, of the tutorials, given by Santiago Escobar and Rubén Rubio, and of the experience report, given by Peter Csaba Olveczky.

A selection of the papers accepted for presentation, along with the full papers or extended abstracts of the invited talks, tutorials, and experience report, will appear in the proceedings published in the Springer LNCS series, following the tradition of previous meetings in this series.

We sincerely thank all the authors of papers submitted to the workshop, and the invited speakers for kindly accepting to contribute to WRLA 2022. We are grateful to the members of the program committee and the subreviewers for their careful work in the review process. We also thank the members of the WRLA steering committee for their valuable suggestions. Finally, we express our gratitude to all members of the local organization of ETAPS 2022, whose work has made the workshop possible.

March 14, 2022

Kyungmin Bae
# Table of Contents

Equational Unification and Narrowing in Maude .......................... 1  
*Santiago Escobar*

Rewriting Privacy ............................................................... 2  
*Sebastian M"odersheim*

Tutorial: The Maude strategy language ...................................... 3  
*Rubén Rubio*

Modelling and Quantitative Analysis of BPMN Processes using Maude . 4  
*Gwen Salaën*

Teaching Formal Methods to Undergraduate Students using Maude ...... 5  
*Peter Csaba ¨Olveczky*

CryptoSolve: Towards a Tool for the Symbolic Analysis of  
Cryptographic Algorithms .................................................. 6  
*Dalton Chichester, Wei Du, Raymond Kauffman, Hai Lin, Christopher  
Lynch, Andrew M. Marshall, Catherine Meadows, Paliath Narendran,  
Veena Ravishankar, Lais Roeira and Brandon Rozek*

Formal specification and model checking of lattice-based key  
encapsulation mechanisms in Maude ....................................... 26  
*Duong Dinh Tran, Kazuhiro Ogata, Santiago Escobar, Sedat Akleylek  
and Ayoub Otmani*

Business Processes Analysis with Resource-aware Machine Learning  
Scheduling in Rewriting Logic .............................................. 44  
*Francisco Durán, Daniela Martínez and Camilo Rocha*

Modeling, Algorithm Synthesis, and Instrumentation for Co-simulation  
in Maude .......................................................... 61  
*Simon Thrane Hansen and Peter Csaba ¨Olveczky*

An Efficient Canonical Narrowing Implementation for Protocol Analysis . 82  
*Raúl López-Rueda, Santiago Escobar and José Meseguer*

Checking Sufficient Completeness by Inductive Theorem Proving ....... 102  
*José Meseguer*

On Ground Convergence and Completeness of Conditional Equational  
Program Hierarchies ...................................................... 122  
*José Meseguer and Stephen Skeirik*

Parallel Maude-NPA for Cryptographic Protocol Analysis ............ 143  
*Canh Minh Do, Adrián Riesco, Santiago Escobar and Kazuhiro Ogata*
**Program Committee**

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erika Abraham</td>
<td>RWTH Aachen University</td>
</tr>
<tr>
<td>María Alpuente</td>
<td>Universitat Politècnica de València</td>
</tr>
<tr>
<td>Kyungmin Bae</td>
<td>Pohang University of Science and Technology</td>
</tr>
<tr>
<td>Roberto Bruni</td>
<td>Università di Pisa</td>
</tr>
<tr>
<td>Francisco Durán</td>
<td>University of Málaga</td>
</tr>
<tr>
<td>Santiago Escobar</td>
<td>Universitat Politècnica de València</td>
</tr>
<tr>
<td>Maribel Fernandez</td>
<td>King’s College London</td>
</tr>
<tr>
<td>Mark Hills</td>
<td>East Carolina University</td>
</tr>
<tr>
<td>Nao Hirokawa</td>
<td>JAIST</td>
</tr>
<tr>
<td>Alexander Knapp</td>
<td>Universität Augsburg</td>
</tr>
<tr>
<td>Temur Kutsia</td>
<td>Johannes Kepler University Linz</td>
</tr>
<tr>
<td>Alberto Lluch Lafuente</td>
<td>Technical University of Denmark</td>
</tr>
<tr>
<td>Dorel Lucanu</td>
<td>Alexandru Ioan Cuza University</td>
</tr>
<tr>
<td>Salvador Lucas</td>
<td>Universitat Politècnica de València</td>
</tr>
<tr>
<td>Narciso Martí-Oliet</td>
<td>Universidad Complutense de Madrid</td>
</tr>
<tr>
<td>Jose Meseguer</td>
<td>University of Illinois at Urbana-Champaign</td>
</tr>
<tr>
<td>Aart Middeldorp</td>
<td>University of Illinois at Urbana-Champaign</td>
</tr>
<tr>
<td>Vivek Nigam</td>
<td>Huawei ERC</td>
</tr>
<tr>
<td>Kazuhiro Ogata</td>
<td>JAIST</td>
</tr>
<tr>
<td>Adrian Riesco</td>
<td>Universidad Complutense de Madrid</td>
</tr>
<tr>
<td>Christophe Ringeissen</td>
<td>INRIA</td>
</tr>
<tr>
<td>Camilo Rocha</td>
<td>Pontificia Universidad Javeriana Cali</td>
</tr>
<tr>
<td>Vlad Rusu</td>
<td>INRIA</td>
</tr>
<tr>
<td>Traian Florin Serbanuta</td>
<td>University of Bucharest</td>
</tr>
<tr>
<td>Carolyn Talcott</td>
<td>SRI International</td>
</tr>
<tr>
<td>Peter Ölveczky</td>
<td>University of Oslo</td>
</tr>
</tbody>
</table>
Additional Reviewers

A
Aoto, Takahito

C
Chiang, James

F
Fernet, Laouen

S
Sapiña, Julia
Equational Unification and Narrowing in Maude

Santiago Escobar
VRAIN, Universitat Politècnica de València, Valencia, Spain.
secesobar@upv.es

Abstract. Maude is a language and a system based on rewriting logic. It is a mathematical modeling language thanks to its logical basis and its initial model semantics. Maude can be used in three, mutually reinforcing ways: as a declarative programming language, as an executable formal specification language, and as a formal verification system. Logical reasoning capabilities have been added to Maude and, in this tutorial, we give an overview of the different unification and narrowing techniques available in Maude 3.2.1, focusing on some of the programming, modeling, and verification aspects of Maude.
Rewriting Privacy

Sebastian Mödersheim
Technical University of Denmark, Lyngby, Denmark
samo@dtu.dk

Abstract. Privacy properties are very relevant for security protocols, e.g., privacy of votes in electronic voting, privacy of personal data in e-health care applications, or unlinkability of several transactions by the same user. The de-facto standard to express these properties is via bisimilarity of two processes, e.g. that an intruder cannot distinguish two processes of electronic voting that differ only by swapping the vote of two honest voters. This yields rather technical specifications of systems and their properties, so that it is hard to convince oneself that this specification truly captures all the intuitive privacy goals one wishes to achieve. Moreover verification with bisimilarity in general is hard to automate, and several approaches choose to consider very restricted notions in order to achieve an automated verification procedure.

Together with my colleagues, we have developed an alternative approach called Alpha-Beta-Privacy. The fundamental difference to previous approaches is that we do not describe privacy as a distinction between two possible states of the word, but rather as a single world and logically reasoning what the intruder can find out about this world. We describe every state of the world by two logical formulas in First-Order Logic with Herbrand universes: alpha is a formula that describes high-level information that is considered public (e.g., that there are N binary votes, and R of them are "yes"), and beta describes low-level information (e.g., all the cryptographic messages the intruder has observed and what he knows about their structure). Privacy then means that every model of alpha can be extended to a model of beta, i.e., the technical messages the intruder can observe do not allow him to learn anything that is not entailed by the official information alpha.

Based on such worlds (alpha, beta) we define transition systems that arise from the intruder interacting with several participants (augmenting beta), and gradually information being released (augmenting alpha). This models what the intruder can derive when knowing the processes that the honest agents are executing (but not necessarily the concrete values they work with) in a rewriting based evaluation. This makes privacy actually a reachability problem: can the system reach a state where the intruder learns a statement something he should not be able to.

While in general the underlying problems are undecidable, we show that using rewriting techniques and symbolic representations we can obtain an effective procedure for a bounded number of sessions with a standard algebraic theory of the cryptographic operators.
Tutorial: The Maude strategy language

Rubén Rubio
Universidad Complutense de Madrid, Madrid, Spain
rubenrub@ucm.es

Abstract. Computation in rewriting logic is the succession of independent rule applications anywhere within the term. This flexibility is the cornerstone of its natural representation of nondeterminism and concurrency, but it is sometimes useful to restrict or guide the evolution of rewriting. Strategies are the traditional resource to express these concerns, but specifying them in Maude was not an easy task. This has changed in Maude 3 with the inclusion of an object-level strategy language to control the application of rules. Several operators similar to the usual programming language constructs and regular expressions allow combining the basic instruction of rule application to program arbitrarily complex strategies, which can be compositionally defined in strategy modules. The language was originally designed in the mid-2000s by Narciso Martí-Oliet, José Meseguer, Alberto Verdejo, and Steven Eker based on the previous experience with internal strategies at the metalevel and earlier strategy languages like ELAN, Stratego, and Tom. While its first prototype was available as a Full Maude extension, the language is now efficiently implemented in C++ as part of the official Maude interpreter. Moreover, the new specifications with strategies need to be verified too. Together with Narciso Martí-Oliet, Isabel Pita, and Alberto Verdejo, we have extended the Maude LTL model checker to work with strategy-controlled specifications and established connections with external model checkers for evaluating CTL, CTL*, and μ-calculus properties. More recently, we have also developed a probabilistic extension of the Maude strategy language whose specifications can be analyzed using probabilistic and statistical model-checking techniques.

This tutorial explains the strategy language and these different topics, illustrated with some examples.
Modelling and Quantitative Analysis of BPMN Processes using Maude

Gwen Salaïn
Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, 38000 Grenoble, France
gwen.salaun@inria.fr

Abstract. Business process modelling and optimization is a strategic activity in organizations because of its potential to increase profit margins and reduce operating costs. The Business Process Model and Notation (BPMN) is a graphical modeling language for specifying business processes using a workflow-based notation. BPMN collaboration diagrams are particularly convenient for describing processes consisting of several participants interacting by exchanging messages. Providing automated techniques for analyzing and optimizing BPMN processes is a challenging problem. In this work, we first propose to encode the BPMN syntax and execution semantics in rewriting logic. Then, we rely on the rewriting logic framework and use the Maude system to stochastically simulate multiple concurrent executions of a process instance that compete for the shared resources. This simulation allows us to automatically compute several properties of interest such as average execution time, synchronization times for merge gateways or resource occupancy over time. We will illustrate these ideas with several examples including realistic processes with large workloads.
Abstract. I have been teaching an introductory formal methods course based on Maude—first to third- and fourth-year students, and lately to second-year students—at the University of Oslo for a number of years. The first part of the course introduces functional modules in Maude and covers the basic topics in term rewriting, whereas the second part of the course uses Maude to formally model and analyze a number of classic distributed systems, including: transport protocols such as the alternating bit and the sliding windows protocols, the two-phase commit protocol for distributed atomic commitment, distributed algorithms for mutual exclusion and leader election, and authentication protocols. In this talk I motivate the use of Maude for such an introductory formal methods course, outline the course content, and summarize student feedback and my own impressions about the course.
CryptoSolve: Towards a Tool for the Symbolic Analysis of Cryptographic Algorithms


1 University of Mary Washington, Fredericksburg, VA, USA
2 University at Albany–SUNY, Albany, NY, USA
3 Clarkson University, Potsdam, NY, USA
4 Naval Research Laboratory, Washington, DC, USA
5 Rensselaer Polytechnic Institute, Troy NY, USA
{rozekb}@rpi.edu

Abstract. We present a new tool for the automatic synthesis and verification of cryptographic algorithms. Currently the tool considers symbolic security and invertibility of recursively defined modes of operation with an xor-operation and encryption. A cryptographic mode of operation is an algorithm for encrypting a message of arbitrary length using a block cipher that only encrypts messages of a single fixed length. The system can both automatically generate modes and accept user-defined ones. These modes can then be checked for properties such as security and invertibility. In order to analyze the modes, the tool utilizes term rewriting and unification methods which are implemented in a core supporting library. The state of the tool and underlying library are in an initial iteration. The goal is to continue expanding the tool to consider additional security questions and cryptosystems.

1 Introduction

Although security properties of cryptographic algorithms are generally proved using a computational model in which probabilities of events are explicitly quantified, there are often advantages to using a more abstract symbolic model, as long as we can prove the symbolic model computationally sound: that is, if the symbolic analogue of a particular property holds, then the computational version also holds. Symbolic methods are generally simpler to apply and more amenable to automation than computational ones, and they have been proven particularly useful in the automatic generation and verification of cryptosystems, e.g. [5, 6, 11], in which many candidates must be evaluated at once. Even when not proven computationally sound, they can be useful for weeding out insecure cryptosystems [3, 16]. However, the symbolic problems we encounter often come with constraints tied to the properties of the cryptosystem such as requiring that
any substitutions be constructable from terms and function symbols available to an adversary, so specialized algorithms or tools may be necessary.

In this paper we present a technical overview of such a tool,\(^6\) that has been designed to manipulate and analyze specifications of cryptosystems. This in turn allows for the automatic generation and symbolic analysis of certain cryptographic algorithms. The goal of this new tool is broad, to develop not only a usable analysis tool for an extensive family of cryptographic algorithms but to also develop the underlying libraries which could be used in analysis of additional algorithms and properties, and also in other symbolic analysis tools. At this point the tool supports methods for generating and verifying cryptographic modes of operations (MOOs); we use this application to illustrate the design and behavior of the tool in this paper.

We are starting with a set of base libraries for critical symbolic capabilities such as term representation, term rewriting, unification, and more. Building on these libraries, we have developed a tool that can generate modes of operations and prove or disprove security properties of both automatically generated and user-input modes. The goal of the MOO security component of the tool is to automatically generate and test cryptographic algorithms with the ultimate aim of weeding out insecure algorithms and identifying algorithms that are symbolically secure. If it is known that symbolic security implies computational security, then we conclude that the algorithms are computationally secure. If not, we can consider the algorithm as at least a candidate for a proof of computational security.

Currently the modules available in the tool and a simplified representation of their relation to each other is described in Figure 1.

![Fig. 1: Tool modules and dependencies](image-url)

\(^6\) The current version of the tool can be found here: [https://symcollab.github.io/CryptoSolve/](https://symcollab.github.io/CryptoSolve/).
In the remainder of the paper we cover the current state and capabilities of the tool without focusing on the theory behind it. Where necessary, we provide a brief theoretical background and indicate aspects of which the tool is based upon. The rest of the paper is organized as follows. We give a brief review of necessary background material in Section 2. A discussion of related work is covered in Section 3. An overview of the symbolic base modules is presented in Section 4. An overview of the security modules, their use, capabilities, and pointers to the theory behind these methods is given in Section 5. The invertibility checking module is covered in Section 6. We briefly cover the various interfaces for the tool in Section 7. We provide preliminary experimental results in Section 8. Finally, the conclusions and future work are discussed in Section 9.

2 Preliminaries

2.1 Terms, Substitutions, and Equational Theories

Given a first-order signature \( \Sigma \), a countable set \( N \) of variables bound by the symbol \( \nu \), and a countable set of variables \( X \) (s.t. \( X \cap N = \emptyset \)), the set of terms constructed from \( X, N, \) and \( \Sigma \) is denoted by \( T(\Sigma,N \cup X) \). Note that since \( N \) is a set of bounded variable we can often treat these as constants in the first-order theory. A substitution \( \sigma \) is an endomorphism of \( T(\Sigma,N \cup X) \) with only finitely many variables not mapped to themselves, denoted by \( \sigma = \{ x_1 \mapsto t_1, \ldots, x_m \mapsto t_m \} \). Application of a substitution \( \sigma \) to a term \( t \) is written \( t\sigma \).

Given a set \( E \) of \( \Sigma \)-axioms (i.e., pairs of \( \Sigma \)-terms, denoted by \( l = r \)), the equational theory \( =_E \) is the congruence closure of \( E \) under the law of substitutivity. For any \( \Sigma \)-term \( t \), the equivalence class of \( t \) with respect to \( =_E \) is denoted by \( [t]_E \). Since \( \Sigma \cap N = \emptyset \), the \( \Sigma \)-equalities in \( E \) do not contain any bound variables in \( N \). An \( E \)-unification problem with bound variables in \( N \) is a set of \( \Sigma \cup N \)-equations \( P = \{ s_1 =^E t_1, \ldots, s_m =^E t_m \} \). A solution to \( P \), called an \( E \)-unifier, is a substitution \( \sigma \) such that \( s_i \sigma =^E t_i \sigma \) for all \( 1 \leq i \leq m \).

The primary equational theory implemented in the tool is the theory of xor, \( E_{\text{xor}} \). This theory can be represented as a combination of a rewrite system, \( R_\oplus \), and an associative and commutative (AC) equational theory, \( E_\oplus \). \( E_{\text{xor}} = R_\oplus \cup E_\oplus \), \( R_\oplus = \{ x \oplus y \rightarrow 0, x \oplus 0 \rightarrow x \} \), \( E_\oplus = AC(\oplus) \), over the signature \( \Sigma_\oplus = \{ \oplus/2, f/1, 0/0 \} \). We will often denote this as the \( MOO_\oplus \) algebra and modes of operations defined in this algebra as \( MOOs_\oplus \).

Additional background material on equational theories, rewriting, and unification can be found here [2].

2.2 Modes of Operation and Symbolic Security

Modes and Their Security A cryptographic mode of operation can be described as a high level as follows. The plaintext message \( M \) is first broken into fixed sized blocks. Each block \( m_i \) is processed using the block encryption function \( E_k \) along with some additional operations to produce a cipher text block \( C_i \). Typically,
the previous cipher block is used in the computation of the current block, and an initialization vector $IV$ is used to add randomness to the first block. The final ciphertext is the sequence of cipher blocks thus produced. Figure 2 illustrates this process for Cipher Block Chaining (CBC) mode.

In order to model these modes so that they can be checked via symbolic methods, we use symbolic histories (defined in [12]). These describe interactions between the adversary and the oracle, in which the adversary sends blocks of plaintext to be encrypted, and the oracle sends back blocks of ciphertext according to some fixed schedule defined by the mode. E.g., in a block-wise schedule a ciphertext block is sent immediately after it is generated by the mode. In a message-wise schedule, the ciphertext blocks are not sent until after the entire message is encrypted.

The symbolic definition of security we use is based on the computational security property IND$^+$-CPA introduced by Rogaway in [14]. This is defined in terms of a game in which a challenger first chooses one of two oracles with probability 1/2. The first is an encryption oracle that returns cipher text when given plaintext, and the second is a random bits oracle that returns a string of random bits that is as long as the ciphertext would have been. The adversary interacts with the oracle by sending it plaintext and receiving the oracle’s response. At any time it can stop the game and guess which oracle is interacting with. Its advantage is defined to be $| \frac{1}{2} - p |$, where $p$ is the probability that the adversary guesses correctly. A mode is IND$^+$-CPA-secure if its advantage is negligible in some security parameter $\eta$, where a function $g$ is said to be negligible if there is no polynomial $q$ such that $g(\eta) \leq q(\eta)$ for all $\eta$. In the case of modes of encryption, the security parameter is the maximum of the block size and the key size. The motivation for a definition of this sort is that if the adversary cannot distinguish the output of the cryptosystem from random noise, then it learns nothing about the plaintext. This form of security, in which the security of a cryptosystem is quantified in terms of the adversary’s inability to distinguish between the output of an encryption oracle and the output of an oracle that does not use the content of the plaintext in its calculations, is common in cryptography.
We note that if the adversary can create plaintexts that consistently cause a set of ciphertexts to exclusive-or to zero, then it can distinguish between the real and random case with overwhelming probability. If such an equality holds for the case in which the substitution is the identity, we say that the mode is degenerate. In all other cases it is necessary but not sufficient that the adversary must be able to consistently cause at least one given pair of $f$-rooted terms to be equal, known as a collision. We describe the symbolic model below, and then describe the unification problem that is associated with it.

The Symbolic Model and Symbolic Security

The blocks sent between the adversary and the oracle are modeled by terms in the $\text{MOO}_\oplus$ algebra. These $\text{MOO}_\oplus$-terms consist of free variables representing plain-text blocks, bound variables representing random strings, and terms built up using these variables and the signature $\Sigma = \{\oplus/2,0/0,f/1\}$, with the Xor equational theory, where $f$ is the encryption function for some fixed key $K$, i.e., $\text{enc}(K,\cdot) = f(\cdot)$. Note that $f$ is not computable by the adversary.

A symbolic history of the adversary’s interaction with the oracle is modeled by a list of $\text{MOO}_\oplus$-terms of the form $[t_1, t_2, \ldots, t_n]$, where plaintext blocks are represented by free variables. All $\text{MOO}_\oplus$-terms are listed in the order that they are sent. For example, the following symbolic history models the CBC mode of operation with three cipher blocks using the block-wise schedule: $\nu IV[IV,x_1,f(IV \oplus x_1),x_2,f(x_2 \oplus f(IV \oplus x_1))]$. Here $IV$ is a bound variable representing an initialization vector. Each $x_i$ models a plain-text block sent by the adversary and each $f$-rooted term is a cipher block returned by the oracle according to the definition of the mode. For example, in CBC the $i$th cipher block $C_i$ is modeled by $f(C_{i-1} \oplus x_i)$, where $x_i$ is the $i$th plaintext.

Each symbolic history models the interleaving of one or more sessions between the adversary and oracle, where a session is a history that encrypts a single message consisting of a sequence of plaintext blocks. In this case the initial nonces, the IVs, will be fresh for each session.

The notions of computable substitutions and symbolic security are defined by Lin et al. in [10]. Let $P$ be a symbolic history. A substitution $\sigma$ is computable w.r.t. $P$ if $\sigma$ maps each variable to a term built up using the operators $0$ and $\oplus$ on terms returned by the oracle earlier than $x$ in $P$. A mode of operation $M$ is symbolically secure if for there is no symbolic history $P$ of $M$ such there is a $S$ of terms returned by the encryptor in $P$ s.t. $\bigoplus_{t \in S} t \sigma = 0$, where $\sigma$ is computable substitution w.r.t. $P$. It is shown in [10] that a mode of operation $M$ is symbolically secure if and only if $M$ is statically equivalent to random; static equivalence [1] is a symbolic definition indistinguishability commonly used in symbolic protocol analysis.

We note that if a mode satisfies $\text{IND}^S$-$\text{CPA}$, then it must be symbolically secure, because if the adversary that could make a substitution to the plaintext that would always cause the same equation to be satisfied by the ciphertext, it could easily distinguish the ciphertext from random with overwhelming probability. A stronger condition has been shown by Meadows in [12] to imply $\text{IND}^S$-$\text{CPA}$ security. It has two parts. The first is non-degeneracy, which requires that sym-
bolic security hold for the trivial case in which the computable substitution $\sigma$ is the identity. The second is the condition that no two different $f$-rooted terms have a computable unifier, whether or not it leads to a violation of symbolic security. This does not necessarily mean that symbolically secure modes that fail to satisfy the second condition are not IND$^*$-CPA secure, simply that more work may be required to prove them so.

Checking Symbolic Security: Examples Let’s consider several examples of symbolic histories and checking for symbolic security. We start with the classic example of an insecure mode: the Electronic Code Book (ECB) mode. In ECB, each block is encrypted separately, so plaintext $x_1, \ldots, x_n$ yields ciphertext $f(x_1), \ldots, f(x_n)$. Notice that after applying ECB, the image in Figure 3 is still not completely scrambled and some information from the original picture can still be deduce. This is because whenever two plaintext blocks are identical they produce the same ciphertext blocks. Thus, any substitution unifying any two free variables is computable and leads to a violation of symbolic security.

Other MOOs may be symbolically secure or insecure depending on the schedule. For example, consider a symbolic history of CBC with three ciphertext blocks: $P_2 = \nu IV[IV, x_1, f(x_1 \oplus IV), x_2, f(x_2 \oplus f(x_1 \oplus IV))]$. We consider two schedules: the block-wise schedule, where each ciphertext block is returned to the adversary as soon as it can be computed, and the message-wise schedule, where they are returned all together at the end. Note that in the block-wise schedule there is a computable unifier of $f(x_1 \oplus IV)$ and $f(x_2 \oplus f(x_1 \oplus IV))$, namely $\sigma = \{ x_1 \mapsto IV, x_2 \mapsto f(0) \}$, but this is not computable in the message-wise schedule, which can be shown to be symbolically secure and IND$^*$-CPA secure.

Finally, we consider one additional MOO, Output Feedback Mode (OFB). Consider an OFB history with three ciphertext blocks: $P_3 = \nu IV[IV, x_1, f(IV) \oplus x_1, x_2, f(f(IV)) \oplus x_2]$. Note that, in order to unify $f(IV) \oplus x_1$ and $f(f(IV)) \oplus x_2$, the adversary would have to set $\sigma x_2 = x_1 \oplus f(IV) \oplus f(f(V))$, which it cannot to no matter what schedule is used, because it does not learn $f(f(V))$ until after it has computed $x_2$. OFB is also both symbolically secure and IND$^*$-CPA secure. Notice that when generating the cipher blocks for differing MOOs such as CBC and OFB, the root symbol of the cipher blocks could differ and this will impact the unification algorithm required.

Hello world
(a) Image before ECB encryption
(b) Image after ECB encryption

Fig. 3: ECB encryption with AES 128 ECB
3 Related Work

The most closely related work to ours is other publicly available tools that have been developed for the generation and testing of security properties of crypto algorithms (e.g. [3, 5, 6, 11]). Perhaps the first of these is the work by Barthe et al. [3]. This paper describes a tool, ZooCrypt, designed for the analysis of chosen plaintext and chosen cipher-text security public-key encryption schemes built from trapdoor permutations and hash functions. A ZooCrypt analysis of a cryptosystem consists of two stages. In the first stage a symbolic analysis tool is used to search for attacks on the cryptosystem. If none are found, the analysis enters the second stage, in which an automated theorem prover is used to search for a security proof in the computational model.

Later work incorporates computational soundness results that allow one to use symbolic techniques to prove computational security. For example, Malozemoff et al. [11] provide a symbolic algorithm whose successful termination implies adaptive chosen plaintext security of cryptographic modes of operation using the message-wise schedule. These results are extended by Hoang et al. in [6] to symbolic techniques for proving adaptive chosen ciphertext security of modes. Both papers also include software that implements symbolic algorithms for generating cryptosystems and proving their security. Other work by Carmer et al. [5] gives a symbolic algorithm for deciding security of garbled circuit schemes, and includes a tool, Linisynth, that generates such schemes and verifies their security using the algorithm.

All these tools have one thing in common: they only implement the algorithms described in the paper they accompany, and thus are intended mainly as proofs of concept, not as general tools for the generation and analysis of algorithms. The goal of CryptoSolve, however, is to serve as a tool for designing and experimenting with multiple types of cryptosystems, security properties, and algorithms. Although it is currently restricted to cryptographic modes of operation, it can be applied to two different properties (static equivalence to random and invertibility, using three different algorithms), and we are working on adding more.

There is also a large amount of related research in formulating and proving indistinguishability properties for the symbolic analysis of cryptographic protocols. These properties are analogous to the computational indistinguishability properties used in cryptography. The main differences are that 1) symbolic indistinguishability does not always imply computational security (see, for example Unruh [15]), and 2) the symbolic algorithms are optimized for protocols, not crypto-algorithms, so applying them directly is not always advisable. Even so, the approaches used in symbolic protocol analysis can be helpful. For example an undecidability result in Lin et al. [10] is based on an undecidability result for cryptographic protocols analysis due to Küsters and Truderung [7]. To facilitate this interaction between symbolic protocol analysis and symbolic cryptography, we use a specification language, due to Baudet et al. [4], that is based on the most popular formal language used by tools for the formal analysis of cryptographic protocols, the applied pi calculus [1]. This makes it easier to incorporate
algorithms and properties derived from related work in symbolic cryptographic protocol analysis.

4 Symbolic Library

The symbolic term library is a software library built on top of the Python programming language. It is a sandbox for students and researchers to experiment with unification, rewrite theory, and their applications. In addition, it is easy to write new algorithms that incorporate other software packages like scipy to provide linear algebra routines. In the following sections, we go over the Unification module and each of the security modules in detail.

4.1 Rewrite Module

Term rewriting is a critical component of the tool and has many applications outside of symbolic security analysis. In this section we give a brief overview of the term rewriting module. Rewrite rules are created by using RewriteRule. To apply a rewrite rule use the method apply.

```
from symcollab.rewrite import RewriteRule
r = RewriteRule(f(x, x), x)
term = f(f(x, x), f(x, x))
print(r.apply(term))
```

By not specifying a position in apply, a dictionary is returned where the key is the position and the value is the rewritten term. The empty string here corresponds to $\epsilon$.

```
{'': f(x, x), '1': f(x, f(x, x)), '2': f(f(x, x), x)}
```

```
print(r.apply(term, '2'))
```

```
f(f(x, x), x)
```
A rewrite system is a set of rewrite rules and can be defined by the following

```python
from symcollab.rewrite import RewriteSystem
r1 = RewriteRule(f(x, x), x)
r2 = RewriteRule(f(a, x), b)
rs = RewriteSystem({r1, r2})
```

In this library, variants are calculated by repeatedly applying rewrite rules. Internally it is represented as a tree where the depth corresponds to the number rules applied. There could be an infinite number of variants dependent on the properties of the rewrite system. Due to the potentially infinitary nature of variants, they are implemented as generators in Python. This allows for the variant to be computed only when requested. Variants are generated breadth-first.

```python
from symcollab.rewrite import Variants
term = f(a, f(b, b))
vt = Variants(term, rs)
print(next(vt))
```

There is a simple algorithm implemented in the library to check if the variants of a term is finitary.

```python
def is_finite(v: Variants, n: int) -> bool:
    iteration = 1
    for _ in v:
        if iteration > n:
            return False
        iteration += 1
    return True
```

We’ve also implemented an algorithm to narrow one term to another within a rewrite system. This returns an ordered list of rewrite rules that need to be applied from front to back in order to achieve the goal term.
The rewrite module contains an algorithm that takes a term, rewrite system, and a bound which then applies rewrite rules until either the bound is hit or the term cannot be matched anymore.

where the first element of the tuple is its normal form, and the second element is the list of rewrite rules applied.

### 4.2 Unification Module

Unification, especially equational unification, is a critical subroutine used in the tool to identify insecure MOOs (see [8,10]). The tool contains a growing library of unification methods. These methods can be used in the analysis of cryptographic algorithms but can also be used on their own or in other applications.

Our unification algorithms return a set of SubstituteTerm, where a SubstituteTerm represents a substitution. An empty set denotes that no unifier was found.

**Example 1.**
from symcollab.algebra import Function, Constant, Variable
from symcollab.Unification import unif

# Define terms
f = Function("f", 1)
x = Variable("x")
y = Variable("y")
a = Constant("a")
b = Constant("b")

# Run Syntactic Unification
print(unif(f(x, y), f(a, b)))

Currently, the tool contains the following standard unification algorithms: syntactic unification, xor unification, and local unification algorithms.

4.3 Unification and MOO Analysis

The method used to conduct the formal analysis of cryptographic protocols requires that the unification algorithm not only solve the equational unification problem for the appropriate theory, such as xor, but also obey the computable substitution restriction (See Section 2.2). Currently implemented are unification algorithms for f-rooted local unification (p_unif in the tool library), and ⊕-rooted local unification (XOR_rooted_security in the library).

Both f-rooted local unification and ⊕-rooted local unification can be considered as a special form of unification modulo the theory of xor, which can be represented as a rewrite system. The f-rooted local unification algorithm is an iterative process. It starts with a xor-unifier, then repeatedly refine the unifier to obey the computable substitution restriction. In the ⊕-rooted local unification algorithm, all possible substitutions that obey the computable substitution restriction are represented compactly as a meta-substitution. The algorithm then proceeds by instantiating the meta-substitution until a xor-unifier is obtained. See [8] for full details.

5 Security Modules

This part of the tool is based on the work under development in [10,13]. There, a method is developed for checking symbolic security, which in turn can be used to synthesize secure cryptographic modes of operation. See Section 2.2 for more background details. We give an overview of each of the components developed for checking symbolic security beginning with the $MOO_\oplus$-Programs.
5.1 *MOO⊕*-Programs

Briefly, a *MOO⊕*-program is a symbolic specification of a mode of operation using the *MOO⊕*-algebra. The tool contains a library implementation which allows for the representation and generation of *MOO⊕*-Programs. The library currently allows *MOO⊕*-Programs that are constructed over the signature \( \Sigma = \{\oplus/2, 0/0, f/1\} \) and represented as a simple recursive function. Once a *MOO⊕*-Program has been defined, the library can then apply a number of operations on that *MOO⊕*-Program, including: generating terms in a run of the *MOO⊕*-Program, checking symbolic security of the program, and checking invertibility of the program (discussed in Section 6). We start by describing the standard *MOO⊕*-Programs implemented.

**Standard and Custom *MOO⊕*-Programs** Currently there are several well-known cryptosystems implemented to serve as examples for users. For example, the cipher block chaining cryptosystem is defined below:

```python
from symcollab.moe import MOO

@MOO.register('cipher_block_chaining')
def cipher_block_chaining(iteration, nonces, P, C):
    f = Function("f", 1)
    i = iteration - 1
    if i == 0:
        return f(xor(P[0], nonces[0]))
    return f(xor(P[i], C[i-1]))
```

Notice that this provides a relatively simple example of the type of recursive cryptosystems built over an xor-theory that are currently supported. Here the base cipher block is defined as \( f(P_0 \oplus nonces[0]) \), where \( P_0 \) is the initial plaintext sent by the adversary, and \( nonces[0] \) is a bound variable representing the initialization vector. Then the recursive case is \( C_i = f(P_i \oplus C_{i-1}) \). The underlying libraries have been constructed to allow the encoded version of the system definition to closely match the theoretical one.

Similarly, a user can create their own custom mode of operation by adding the recursive definition to the MOO library.

**User defined schedule** In addition to the block-wise and message-wise schedules (as described in Section 2.2), the user can define their own schedules based on the iteration number. For example, this is a custom schedule that has the oracle only return ciphertexts on even iterations.
Automatically Generated Singly Recursive $MOO_{\oplus}$-Definitions A user can ask the library to generate a recursive definition of a mode of operation. Currently there is one method in the tool library to automatically generate MOOs, it works by recursively generating MOOs starting with the base components (IV, variables) and building singly recursive definitions using the xor and $f$ function, and recursive references to prior cipher blocks. The current method has limitations, for example only one nonce is used, the signature is limited to $\Sigma = \{\oplus/2, 0/0, f/1\}$, only single recursion is used, and the base case is fixed to the initialization vector. Thus, the current method won’t generate all possible $MOO_{\oplus}$s. For example, a MOO that uses two nonces in its recursive definition won’t be generated. We plan to expand this functionality in future versions of the tool allowing a user to automatically generate more classes of MOOs. Note, this doesn’t limit the possible MOOs that a user can analyze by using the custom module.

The user can also filter the recursive definitions by properties such as availability of the initialization vector, if it requires chaining, or if the number of calls to the encryption function $f$ is less than a specified bound. A mode of operation has the chaining property if it incorporates a previous ciphertext into its recursive definition.

```python
from symcollab.moe import MOO_Schedule
@MOO_Schedule.register('even')
def even_schedule(iteration: int) -> bool:
    return iteration % 2 == 0
```

```python
from symcollab.moe import MOOGenerator
gen = MOOGenerator()
next(gen)
```

```python
from symcollab.moe import FilteredMOOGenerator
gen = FilteredMOOGenerator(
    max_f_depth=3,
    requires_iv=True,
    requires_chaining=False)
next(gen)
```

```python
from symcollab.moe import f
from symcollab.moe import xor
```

```python
f(P[i])
xor(IV, P[i])
```
From the recursive definition, we can then pass it to the class `CustomMOO` in order to generate the base cases. By default it creates a new nonce.

### 5.2 Interactions with $MOO_{\oplus}$-Programs

Once a mode of operation and schedule have been defined, the tool can do several things with the definition. The first and simplest is to generate the terms corresponding to the symbolic representation of the cipher blocks.

**Example 2.**

```
Code                  Output
from symcollab.moe import { y_1 -> f(x + IV) }
MOOProgram
moo_session = MOOProgram(
    moo_name="cipher_block_chaining",
    schedule_name="every"
)
plaintext = Variable("x")
ciphertext = moo_session.rcv_block(plaintext)
print(str(ciphertext))
```

The second is to check security.

### 5.3 Checking Symbolic Security

The tool can check for symbolic security in several ways. The first, and most exhaustive, is via the local unification approach. In this approach cipher blocks of the $MOO_{\oplus}$-program under consideration are generated and the appropriate local unification algorithm is used to see if any blocks sum to 0, see [12] for the full details of this approach.

The difficulty with this approach is that it can be time consuming in practice. However, a second approach has been developed in [10]. The approach doesn’t require the generation of cipher blocks and works directly with the initial $MOO_{\oplus}$-program definition. This approach is not complete but works for many cases and has the advantage of being much more efficient. Therefore, we have implemented it as a first pass symbolic security check for the tool. If the first pass cannot decide symbolic security, then, the full security check requiring block generation will be used.

**Example 3.** Check for symbolic security and invertibility:
6 Invertibility

A cryptographic algorithm is invertible if given a ciphertext and a decryption key, the original plaintext can be retrieved. This is not a given for any MOO⊕-program, even a secure one. Therefore, in the automatic generated setting we will need methods for checking if the invertibility property holds for any particular MOO⊕-program. Currently the tool is able to check invertibility for a large class of recursively defined MOOs. This class includes the well known MOOs, such as CBC, ECB, and CFB. More detailed information on theory and method for checking invertibility has been presented in [10].

The invertibility checker is built into the MOO security check functionality in the tool and can be requested simply by setting the “invert_check” flag (which is the last flag) in the moo_check function. See Example 3 and Example 4.

Example 4.

```python
from symcollab.moe.check import moo_check
from symcollab.Unification.p_unif import p_unif
result = moo_check('cipher_block_chaining', 'every', p_unif, 2, True, True)
print(result.invert_result) # prints True
```

7 User Interface

We currently have two different interfaces to CryptoSolve: Command Line Interface (CLI) and Web Interface. These are included in the symcollab-moe package.

The command line interface is the most expressive since it exposes the entire library in a standard Python REPL. It can be invoked using the executable moo_tool and it comes with a built-in help function to get started.
The web interface starts up a webserver which provides a more user friendly interface to check the security of custom/automated MOOs. This can be invoked via the moo_website executable. The website has four pages: Tool, Simulation, Custom, and Random.

The tool page showcases the common parameters to assess symbolic security for modes of operation. The message schedule dictates when the oracle will respond with the most common being “Every” which means block-wise and “End” which means message wise. We can also specify if the adversary has knowledge of the initialization vector when trying to compute their attacks. The simulation page takes the MOO and schedule selected, and walks through the symbolic history generated with each additional plaintext. The custom page lets the user test MOOs with a custom recursive definition over the signature $\Sigma = \{ f(\oplus, /), P_i / 0, C_i - 1 / 0 \}$ ($P_i$ representing a variable and $C_i - 1$ bound variables). For example, a custom MOO uses the following syntax $f(\text{xor}(P[i], C[i - 1]))$. This page is very similar to the tool page, where the user can choose different unification algorithms, schedules, and restrictions. The random page, procedurally produces MOOs under given constraints and tests them for security and invertibility. An example of a common restriction is a bound on the number of times an encryption function $f$ is called.

### 8 Experiments

A benefit of the tool design is that it is easy to integrate the above described functions into a script which can then be used to run experiments. For example, we have included a script, located in the experiments directory of the tool, that allows the user to run longer experiments and can handle restarts. A version of that script, simplified due to space constraints, is as follows:

```python
from symcollab.moe import CustomMOO, MOOGenerator, moo_check
from symcollab.Unification.constrained.p_unif import p_unif
from symcollab.Unification.constrained.xor_rooted_unif import XOR_rooted_security
from symcollab.xor.xor import XorTerm
mgen = MOOGenerator()
```
while True:
  t = next(mgen)
  tm = CustomMOO(t)
  unif_algo = XOR_rooted_security if isinstance(t, XorTerm) else p_unif
  check_result = moo_check(tm.name, 'every', unif_algo, 3, True, True)
  print(check_result)

In this script, we generate new candidate MOOs one at a time and test them for security. The output of moo_check is the data structure called MOOCheckResult. This has the following fields: collisions (set of computable substitutions that cause a collision to occur), invert_result (whether or not the MOO is invertible), iterations_needed (number of iterations before a collision was found), and whether or not the MOO satisfies symbolic security up to the bound checked.
8.1 Initial Experimental Results

A sample of some of the secure MOOs found during early experiments are listed in Table 1. All of these MOOs were created automatically by the currently implemented recursive MOOGenerator. As a future work we plan to create additional generators that the user can select and allow for user defined generators.

<table>
<thead>
<tr>
<th>Secure MOOs Found via Automatic Generation and Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $C_0 = IV, C_i = f(f(f(P[i - 1]) \oplus r) \oplus C[i - 1])$</td>
</tr>
<tr>
<td>2. $C_0 = IV, C_i = f(f(f(P[i])) \oplus C[i - 1]) \oplus r$</td>
</tr>
<tr>
<td>3. $C_0 = IV, C_i = f(f(P[i]) \oplus C[i - 1]) \oplus C[i - 1]$</td>
</tr>
<tr>
<td>4. $C_0 = IV, C_i = f(f(f(P[i]) \oplus r \oplus C[i - 1]))$</td>
</tr>
<tr>
<td>5. $C_0 = IV, C_i = f(f(P[i]) \oplus C[i - 1]) \oplus f(C[i - 1])$</td>
</tr>
</tbody>
</table>

Table 1: Examples of secure MOOs found using the MOO generator

Experiments can also be done without the MOOGenerator, where MOOs are generated via hand or a custom script and then checked for security. This is an attractive option because it allows the user to easily customize the type of MOOs they are considering. Table 2 includes some example secure MOOs that were created by hand and then tested for security using the tool. Note, that although all three MOOs are secure only the first MOO can be shown by the tool to be invertible (via the method developed in [10])! Thus, secure but useless MOOs can also be discarded.

<table>
<thead>
<tr>
<th>Secure MOOs Found via Custom Generation and Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $C_0 = IV, C_i = f(P[i] \oplus f(C[i - 1])) \oplus f(C[i - 1])$</td>
</tr>
<tr>
<td>2. $C_0 = IV, C_i = f(P[i] \oplus f(C[i - 1])) \oplus f(P[i])$</td>
</tr>
<tr>
<td>3. $C_0 = IV, C_i = f(P[i] \oplus C[i - 1]) \oplus f(P[i]) \oplus f(C[i - 1])$</td>
</tr>
</tbody>
</table>

Table 2: Examples of secure MOOs found using a custom generator

Based on the initial experimentation with the tool there are some interesting early questions: Can the set of secure MOOs be closed under some operation such as applying encryption, $f$, on top? Are there cases where we can place a bound on the number of iterations to check security? We’re particularly motivated by the second question, due to the complexity of our saturation based decision procedures. For some of the MOOs we tested, it took on the order of days in order for the algorithm to find a collision.
9 Conclusion and Future Work

We present a new tool for the symbolic analysis of cryptographic algorithms. Currently, the tool supports symbolic analysis of cryptographic modes of operation. It is very flexible: it can both automatically generate modes of operation and accept user-defined ones; it accepts different schedules (e.g. message-wise schedule, block-wise schedule); users can put restrictions on the encryption depth. Using the tool, we found some new modes of operation, which are both secure and invertible. In Section 8, we made some interesting conjectures about the security of modes of operation.

We plan to improve the tool in several directions. For example, we would like to improve the efficiency of the security checking. Although the problem of checking security of modes of operation is undecidable in general, it is still possible to come up with efficient algorithms that work well in practice [10]. We are exploring additional methods that don’t require applying unification to every cipher block but rather work by analyzing the mode itself. We also hope to expand the functionality, by allowing a wider range of MOOs, and more user modification such as user defined schedules. We are working to improve the tool’s usability by improving the interface and output. For example, we hope to expand its output to include proofs of security. We are also working to extend our tool by considering other security features (e.g. authentication [9]) and other crypto systems (e.g. Garbled Circuits [5]). Finally, we will continue to polish and expand the underlying symbolic library.

References


Formal specification and model checking of lattice-based key encapsulation mechanisms in Maude

Duong Dinh Tran¹ ⋆, Kazuhiro Ogata¹ ⋆, Santiago Escobar² ⋆⋆, Sedat Akleylek³ ⋆ ⋆ ⋆, and Ayoub Otmani⁴

¹ Japan Advanced Institute of Science and Technology, Ishikawa 923-1292, Japan
{duongtd,ogata}@jaist.ac.jp
² VRAIN, Universitat Politècnica de València, Valencia, Spain
sescobar@upv.es
³ Ondokuz Mayis University, Samsun, Turkey
sedat.akleylek@bil.omu.edu.tr
⁴ University of Rouen Normandie, France
ayoub.otmani@univ-rouen.fr

Abstract. Advances in quantum computing have shown a serious challenge for widely used current cryptographic techniques because a sufficient large-scale quantum computer can efficiently solve hard mathematical problems on which the current public-key cryptography is relying. That is the reason why recently many researchers and industrial companies have spent lots of effort on constructing post-quantum cryptosystems, which are resistant to quantum attackers. Large numbers of post-quantum key encapsulation mechanisms (KEMs) have been proposed to provide secure key establishment - one of the most important building blocks in asymmetric cryptography. This paper presents formal security analysis of three lattice-based KEMs: Kyber, Saber, and SK-MLWR. We first formally specify each of them in Maude, a rewriting logic-based specification/programming language equipped with many functionalities, such as a reachability analyzer (or the search command) that can be used as an invariant model checker, and then conduct invariant model checking with the Maude search command, finding an attack.

Keywords: key encapsulation mechanism · Maude · post-quantum · model checking.

⋆ D. D. Tran and K. Ogata have been supported by JST SICORP Grant Number JPMJSC20C2, Japan.
** S. Escobar has been partially supported by the grant RTI2018-094403-B-C32 funded by MCIN/AEI/10.13039/501100011033 and ERDF A way of making Europe, by the grant PROMETEO/2019/098 funded by Generalitat Valenciana, and by the grant PCI2020-120708-2 funded by MICIN/AEI/10.13039/501100011033 and by the European Union NextGenerationEU/PRTR.
⋆⋆ S. Akleylek has been partially supported by TUBITAK under Grant No.121R006.
1 Introduction

In recent years, advanced research in the field of quantum computing and quantum information theory has brought a credible threat to cryptosystems currently in use. The most popular asymmetric (or public-key) primitives used today will become insecure under sufficient strong quantum computers because they can be efficiently broken by Shor’s algorithm [23]. The security of these primitives relies on one of the following three hard mathematical problems: the integer factorization problem, the discrete logarithm problem, and the elliptic-curve discrete logarithm problem. All of these problems are hard under conventional computers, but they can be easily solved on a sufficiently powerful quantum computer running Shor’s algorithm. On the other hand, symmetric primitives can be said to be secure against quantum attackers. Although Grover’s algorithm [18], one of the most well-known quantum algorithms, can reduce the complexity to break symmetric primitives, doubling the key size can efficiently ignore these attacks. For example, we can say that AES-256 would be as hard to break by a quantum computer as AES-128 is by a classical computer.

As a response to the quantum attack threat, there is extensive research to find new schemes which are secure even in the presence of quantum adversaries. In the past few years, many post-quantum asymmetric primitives have been proposed as replacements for those traditional ones currently in use. The National Institute of Standards and Technology of USA (NIST) also started the Post-Quantum Cryptography Project in 2017, calling for proposals of post-quantum cryptographic protocols that are secure against both conventional and quantum computers\(^5\). There were 82 submissions to this standardization project, implying the importance of this problem. Among these submissions, there are large numbers of proposals for post-quantum key encapsulation mechanisms (KEMs), which aim to securely establish a symmetric key between two parties. This is understandable because the key exchange algorithm can be said to be the most important building block of asymmetric cryptography.

Security analysis of cryptographic primitives and/or protocols can be fundamentally divided into two approaches: computational security and symbolic security. Proof in the computational model requires a definition of secure cryptographic construction (primitive, protocol), and some assumptions about the computationally infeasible problem. The proof can be regarded as a mathematical reduction, such that it makes sure that the only chance to violate the security of such a construction is to solve the infeasible problem. The authors of the three KEMs considered in this paper have already presented their security proofs in the computational model. However, such proofs are often not easy to understand for non-experts in cryptography. On the other hand, symbolic analysis is easier to understand, computer-verified and suitable for automation. Our approach presented in this paper belongs to the latter. Note that our approach can be applied to not only the three KEMs but also other KEMs and other kinds of primitives as well.

We formally specify and model check three KEMs: Kyber [7] (precisely CRYSTALS-Kyber), Saber [10], and SK-MLWR [1] (the KEM proposed in [1] is called SK-MLWR in the present paper). Because of space limitation, we choose Kyber as the only KEM to illustrate in this paper. The specifications of the other KEMs can be found at https://github.com/duongtd23/kems-mc. Kyber is a KEM whose security is based on the hardness of solving the learning-with-errors (LWE) problem even for quantum computers. While the security of Saber and SK-MLWR relies on the hardness of the Module Learning With Rounding (MLWR) problem. All of them belong to lattice-based cryptography. We use Maude [20], a programming/specification language based on rewriting logic, to formally specify the Dolev-Yao generic intruder [12] as well as these KEMs. By employing the Maude search command, a Man-In-The-Middle (MITM) attack is found for each KEM. Although this kind of attack is not a novel attack for KEMs, the formal specifications in Maude and the model checking experiments are worth reporting. Our ultimate goal is to come up with a new security analysis/verification technique for post-quantum cryptographic protocols, which use post-quantum cryptographic primitives, such as the three KEMs reported in this paper. Formally specifying such primitives is necessary for analyzing the security later on. What is described in the paper is our initial step toward the goal.

Related work. In 2012, Blanchet [4] has surveyed various approaches to security protocol verification in both symbolic model and computational model. In the symbolic model, there is a large number of tools existing for verifying cryptosystems, such as ProVerif [5], Maude-NPA [16], Tamarin [21], and Scyther [9]. The symbolic protocol verifier ProVerif, which was developed by Blanchet, can automatically prove security properties of cryptographic protocol specifications. ProVerif is based on an abstract representation of the protocol by a set of Horn clauses, and it determines whether the desired security properties hold by resolution on these clauses. The practicability of ProVerif has been demonstrated through case studies, such as [19,6]. ProVerif can handle an unbounded number of sessions (executions) of protocols, but termination is not guaranteed in general because the resolution algorithm may not terminate. To mitigate this challenge, Escobar et al. [17] proposed some techniques to reduce the size of the search space in Maude-NPA, such as generating formal grammars representing terms (states information) unreachable from initial states and using super lazy intruder to delay the generation of substitution instances as much as possible. Even though, the termination of the tool is not always guaranteed. Among many case studies that demonstrated the capabilities of Maude-NPA, [15] presented one case study with Diffie-Hellman key agreement protocol.

Scyther [9] is another tool for symbolic security verification of cryptographic protocols. Like ProVerif, Scyther also supports an unbounded number of sessions, but it supports only a fixed set of cryptographic primitives (symmetric and asymmetric encryption and signatures) and does not allow for user-specified equational theories. Its successor, namely Tamarin [21] prover, does support equational theories. Moreover, Tamarin provides two ways of constructing proofs: fully automated mode and interactive mode. The tool may not terminate in
the fully automated mode. In the interactive mode, the tool allows users to provide lemmas that must be proved. Several case studies on security analysis of cryptographic primitives and protocols with Tamarin can be found in [22,13].

Yadav et al. [25] explored NTRU key exchange, a lattice-based public key exchange protocol, and found that it is exposed to an MITM attack. The attack was found in the same manner as what we present in this paper. However, they used neither any tool nor formal specification language as we do.

In the computational security approach, game-based model is known as a standard model for proving security. Security for cryptographic primitives or protocols is defined as an attack game played between an adversary and some benign entity, which is called the challenger. The main idea of the game-based security model is simulation of interaction among these two parties. Eventually, the security proof typically leads to a proof that any supposed adversary can get an advantage over the challenger if and only if he/she is able to solve some computationally infeasible problem (e.g., discrete logarithm, integer factorization). When a proof becomes too complicated, the proof normally employs the sequence of games technique [24]. CryptoVerif [3] is a tool for mechanizing such proof. It can generate proofs by sequences of games automatically or with little user interaction. Alwen et al. [2] have employed CryptoVerif to analyze the security of the Hybrid Public Key Encryption (HPKE), which is a candidate for a new public key encryption standard.

In recent years, there are large numbers of proposals for post-quantum cryptosystems. In addition to key establishment algorithms, there are also many proposals for post-quantum digital signature algorithms, such as CRYSTALS-Dilithium [14] and Rainbow [11]. Amazon Web Services team also proposed a post-quantum Transport Layer Security (TLS) protocol [8], where TLS is known as one of the most widely deployed cryptographic protocols in practice, protecting numerous internet transactions every day. The post-quantum TLS handshake protocol uses a hybrid key exchange method: a traditional key exchange algorithm, such as Diffie-Hellman together with a post-quantum KEM, such as Saber.

**Roadmap.** The remaining of this paper is organized as follows: Sect. 2 first gives some preliminaries, such as KEM and state machine. Sect. 3 describes Kyber KEM, briefly explains the underlying learning with error problem. The specification of Kyber in Maude is presented in Sect. 4. The model checking result and the found attack are presented in Sect. 5. Finally, Sect. 6 summarizes the paper.

## 2 Preliminaries

### 2.1 Key encapsulation mechanism

A key encapsulation mechanism is a tuple of algorithms \((\text{KeyGen}, \text{Encaps}, \text{Decaps})\) along with a finite keyspace \(\mathcal{K}\):

- **KeyGen() → (pk, sk):** A probabilistic key generation algorithm that outputs a public key \(pk\) and a secret key \(sk\).
• **Encaps**(*pk*) → (*c*, *k*): A probabilistic encapsulation algorithm that takes as input a public key *pk*, and outputs a ciphertext (or encapsulation) *c* and a key *k* ∈ *K*.

• **Decaps**(*c*, *sk*) → *k*: A (usually deterministic) decapsulation algorithm that takes as input a ciphertext *c* and a secret key *sk*, and outputs a key *k* ∈ *K*.

A KEM is *ε*-correct if for all (*pk*, *sk*) ← **KeyGen**() and (*c*, *k*) ← **Encaps**(*pk*), it holds that \( Pr[\text{Decaps}(c, sk) \neq k] \leq \epsilon \). We say it is correct if \( \epsilon = 0 \).

### 2.2 State machine and Maude

A state machine \( M \equiv \langle S, I, T \rangle \) consists of a set \( S \) of states, a set \( I \subseteq S \) of initial states and a binary relation \( T \subseteq S \times S \) over states. The set \( R \) of reachable states with respect to \( M \) is inductively defined as follows: (1) for each \( s \in I \), \( s \in R \) and (2) for each \( (s, s') \in T \), if \( s \in R \), then \( s' \in R \). A state predicate \( p \) is an invariant property with respect to \( M \) if and only if \( p(s) \) holds for all \( s \in R \).

In this paper, to express a state of \( S \), we use a braced associative-commutative collection of name-value pairs. Associative-commutative collections are called soups, and name-value pairs are called observable components. That is, a state is expressed as a soup of observable components. The juxtaposition operator is used as the constructor of soups. Let \( oc_1, oc_2, oc_3 \) be observable components, and then \( oc_1 \; oc_2 \; oc_3 \) is the soup of those three observable components. A state is expressed as \( \{ oc_1 \; oc_2 \; oc_3 \} \). There are multiple possible ways to specify state transitions.

In this paper, we use Maude [20], a programming/specification language based on rewriting logic, to specify them as rewrite rules. Maude makes it possible to specify complex systems flexibly and is also equipped with model checking facilities (a reachability analyzer and an LTL model checker). A rewrite rule starts with the keyword rl, followed by a label enclosed with square brackets and a colon, two patterns connected with =\( >\) and ends with a full stop. A conditional one starts with the keyword crl and has a condition following the keyword if before a full stop. The following is a form of a conditional rewrite rule:

\[
\text{crl}\ [lb] : l => r \text{ if } \ldots /\ c_i /\ . . .
\]

where \( lb \) is a label and \( c_i \) is part of the condition, which may be an equation \( lc_i = rc_i \). The negation of \( lc_i = rc_i \) could be written as \( (lc_i =/= rc_i) = \text{true} \), where \( = \text{true} \) could be omitted. If the condition \( \ldots /\ c_i /\ \ldots \) holds under some substitution \( \sigma \), \( \sigma(l) \) can be replaced with \( \sigma(r) \).

Maude provides the search command that can find a state reachable from \( t \) such that the state matches \( p \) and satisfies condition(s) \( c \):

\[
\text{search } [n,m] \text{ in MOD : } t =>* p \text{ such that } c.
\]

where \( MOD \) is the name of the module specifying the state machine, and \( n \) and \( m \) are optional arguments stating a bound on the number of desired solutions and the maximum depth of the search, respectively. \( n \) typically is 1 and \( t \) typically represents an initial state of the state machine.
3 Kyber Key Encapsulation Mechanism

3.1 Notations

Let $\mathcal{B}$ denote the set \{0, \ldots, 255\}, i.e., the set of 8-bit unsigned integers (bytes). Consequently, $\mathcal{B}^k$ denotes the set of byte arrays of length $k$ and $\mathcal{B}^*$ denotes the set of byte arrays of arbitrary length. For two byte arrays $a$ and $b$, $(a||b)$ denotes the concatenation of $a$ and $b$.

The ring of integers modulo $q$ is denoted by $\mathbb{Z}_q$ where the integers are in $[0, q)$. We denote by $R$ the polynomial ring $\mathbb{Z}[X]/(X^n + 1)$ and by $R_q$ the quotient polynomial ring $\mathbb{Z}_q[X]/(X^n + 1)$. Thus polynomials in $R_q$ are of $n$ coefficients where each coefficient is in $[0, q)$. In Kyber, the values of $n$ and $q$ are always fixed to $n = 256$ and $q = 7681$ in all levels of security. Regular font letters denote elements in $R$ or $R_q$ (which includes elements in $\mathbb{Z}$ and $\mathbb{Z}_q$) and bold lower-case letters represent vectors with coefficients in $R$ or $R_q$. By default, all vectors are column vectors. Bold upper-case letters are matrices. For a vector $v$ (or matrix $A$), we denote by $v^T$ (or $A^T$) its transpose. For a vector $v$ we write $v[i]$ to denote its $i$-th entry (with indexing starting at zero); for a matrix $A$ we write $A[i][j]$ to denote the entry in row $i$ and column $j$ (again, with indexing starting at zero).

Let $H$ and $G$ be two hash functions, where $H : \mathcal{B}^* \rightarrow \mathcal{B}^{32}$ and $G : \mathcal{B}^* \rightarrow \mathcal{B}_q^{32} \times \mathcal{B}_q^{32}$. Let KDF denote the key derivation function, where $KDF : \mathcal{B}^* \rightarrow \mathcal{B}^{32}$. Let $x \in \mathbb{Q}$, where $\mathbb{Q}$ denotes the rational numbers set, then $\lfloor x \rfloor$ denotes rounding of $x$ to the closest integer. Let $x \in \mathbb{Z}_q$ and $d < \lfloor \log_2 q \rfloor$, functions Compress and Decompress are defined by the following equations:

\begin{align}
\text{Compress}_q(x, d) &= \lfloor (2^d/q) \cdot x \rfloor \mod 2^d, \quad (1) \\
\text{Decompress}_q(x, d) &= \lfloor (q/2^d) \cdot x \rfloor \quad (2)
\end{align}

Let $x' = \text{Decompress}_q(\text{Compress}_q(x, d), d)$, then we have:

$$|x' - x \mod q| \leq \lfloor \frac{q}{2^{d+1}} \rfloor \quad (3)$$

When $\text{Compress}_q$ and $\text{Decompress}_q$ are used with $x \in R_q$ or $x \in R_q^k$, they are applied to each coefficient individually.

3.2 Kyber

Fig. 1 describes the three algorithms (KeyGen, Encaps, Decaps) of Kyber KEM. It employs the three algorithms (KeyGen, Enc, Dec) of Kyber.CPAPKE, which are shown in Fig. 2. Let us suppose that there are two parties Alice and Bob. The interactions depicted in Fig. 1 are as follows. Alice performs the KEM.KeyGen step, generating a public/secret keys pair $pk$ and $sk$. She keeps the secret key $sk$ and sends the public key $pk$ to Bob. Upon receiving $pk$, Bob executes the KEM.Enc step (i.e., Encaps step) in Fig. 1. He randomly chooses an $m_0$, hashes it, and passes the outputs to the CPAPKE.Enc procedure (depicted in Fig. 2).
The obtained ciphertext \( c \) is sent back to Alice. Upon receiving \( c \), Alice performs the KEM.\( \text{Dec} \) step (i.e., Decaps step). She computes \( c' \) by employing the CPAPE.\( \text{Dec} \) and CPAPE.\( \text{Enc} \) procedures. With a very high probability \( c' \) is equal to \( c \), implying that \( m' \) on Alice’s side is equal to \( m \) on Bob’s side with an overwhelming probability. After that, they can derive the same key \( K \). Note that all multiplications and additions in the two Figures are computed over \( \mathbb{Z}_q[X]/(X^n + 1) \).

\[
\text{KEM.\text{KeyGen}()} \\
\quad z \leftarrow \mathbb{B}^{32} \\
\quad (pk, sk') = \text{CPAPKE.\text{KeyGen}()} \\
\quad sk = (sk'||pk'||H(pk)||z) \\
\quad \text{return } (pk, sk)
\]

\[
\text{KEM.\text{Dec}() } (c, sk) \\
\quad \text{return } (\bar{K}, r) \\
\quad c' = \text{CPAPKE.\text{Enc}()}(pk, m', r') \\
\quad \text{if } c = c' \text{ then return } K = \text{KDF}(\bar{K}, H(c)) \\
\quad \text{else return } K = \text{KDF}(z, H(c))
\]

Fig. 1. Kyber.KEM

Note that in [7], the authors describe Kyber with the employment of functions Encode and Decode. Function Encode serializes a polynomial or a vector of polynomials to a byte array, and function Decode is the inverse of Encode. Furthermore, to perform multiplications in \( R_q \) efficiently, the vectors and matrices are converted to NTT domain and vice versa, where NTT stands for number-theoretic transform. However, implementation or performance is out of the scope of the present paper; thus, for simplicity and ease of understanding, we omit those concepts. Consequently, the notation, e.g., \( pk = (t||\rho) \) is a misuse of notations because \( t \in R_q^n \) and then \( t \) is not a byte array. This notation is understood as \( pk \) is made of \( t \) and \( \rho \).

The procedure to generate matrix \( A \), which is denoted by generate(\( \rho \)) in Fig. 2, taking as input a random seed \( \rho \), is deterministic. Informally, if Alice and Bob share the same random seed \( \rho \), then they can agreeingly derive the same matrix \( A \), whose coefficients of each entry are close to a uniformly random distribution. In contrast, the procedure to sample noise (or error) components (e.g., \( e, e_1, \) and \( e_2 \)), namely sampleCBD, is probabilistic. It takes as input a random seed (e.g., \( \rho \) and \( r \)) and returns a polynomial whose coefficients are close to a centered binomial distribution (to sample a vector, e.g., \( s \) and \( e \), the procedure is called \( k \) times). Informally, coefficients of an output of sampleCBD are mostly close to 0, and their absolute value never greater than a specific small number (which is 5, or 4, or 3, depending on the level of security).
CPAPKE.KeyGen()
\( d \leftarrow B^{32} \)
\( (\rho, \sigma) = G(d) \)
\( R_{k \times k}^{q} \ni A = \text{generate}(\rho) \)
\( R_{k}^{q} \ni s, e \leftarrow \text{sampleCBD}(\sigma) \)
\( t = As + e \)
\( pk = (t||\rho) \)
\( sk = \ldots \)
return \((pk, sk)\)

CPAPKE.Enc\((pk, m, r)\)
\( (t||\rho) = pk \)
\( R_{k \times k}^{q} \ni A = \text{generate}(\rho) \)
\( R_{q}^{q} \ni r, e_{1} \leftarrow \text{sampleCBD}(r) \)
\( u = A^{T}r + e_{1} \)
\( v = t^{T}r + e_{2} + \text{Decompress}_{q}(m, 1) \)
\( c_{1} = \text{Compress}_{q}(u, d_{u}) \)
\( c_{2} = \text{Compress}_{q}(v, d_{v}) \)
return \( c = (c_{1}||c_{2}) \)

CPAPKE.Dec\((c, sk)\)
\((c_{1}||c_{2}) = c \)
\( u' = \text{Decompress}_{q}(c_{1}, d_{u}) \)
\( v' = \text{Decompress}_{q}(c_{2}, d_{v}) \)
\( m' = \text{Compress}_{q}(u' - s^{T}u', 1) \)
return \( m' \)

Fig. 2. Kyber.CPAPKE

4 Formal specification of Kyber

4.1 Formalization of polynomials, vectors, and matrices

We first introduce sort \textit{Poly} that represents polynomials as follows:

\texttt{sort Poly.} \quad \texttt{subsort Int < Poly.}
\texttt{op \_p+ : Poly Poly -> Poly [ctor assoc comm prec 33].}
\texttt{op \_p* : Poly Poly -> Poly [ctor assoc comm prec 31].}
\texttt{op \_p- : Poly Poly -> Poly [prec 33].}
\texttt{op neg_ : Poly -> Poly [ctor].}

where \textit{Int} is the sort of integers. The notation \texttt{subsort Int < Poly} indicates that any integer is also a polynomial. \texttt{_p+}, \texttt{_p*}, and \texttt{p-} denote the addition, multiplication, and subtraction, respectively, between two polynomials. \texttt{neg} denotes the negation of a polynomial. \texttt{assoc comm} indicates that \texttt{_p+} and \texttt{_p*} are declared to be associative and commutative. \texttt{prec 33} attached with \texttt{_p+} and \texttt{_p-} indicates that these operators have the same precedence 33, which is lower precedence than that of \texttt{_p*} (i.e., 31). Note that, we only consider polynomials in \( \mathbb{Z}_{q}[X]/(X^{n} + 1) \) (or \( R_{q} \)), where \( n = 256 \) and \( q = 7681 \). Let \( P1, P2, \) and \( P3 \) be Maude variables of \textit{Poly}. We declare some properties of the operators as follows:

\( \text{eq } P1 \ p+ 0 = P1 \).
\( \text{eq } P1 \ p* 0 = 0 \).
\( \text{eq } P1 \ p* 1 = P1 \).
\( \text{eq } P1 \ p* (P2 \ p* P3) = (P1 \ p* P2) \ p* (P1 \ p* P3) \).
\( \text{eq } P1 \ p- P2 = P1 \ p+ \text{neg}(P2) \).
\( \text{eq } \text{neg}(\text{neg}(P1)) = P1 \).
\( \text{eq } \text{neg}(P1 \ p* P2) = \text{neg}(P1) \ p* \text{neg}(P2) \).

In a similar way, we introduce sorts \textit{Vector} and \textit{Matrix} representing polynomial vectors and matrices, respectively; operators \texttt{v+}, \texttt{dot}, and \texttt{m*} representing the addition & inner product of two polynomial vectors, and multiplication of
a polynomial matrix and a vector, respectively. Let $V_1$, $V_2$, and $V_3$ be Maude variables of Vector. The declarations of the three operators and the distributive property of vectors are specified as follows:

\[
\text{op } _v+ : \text{Vector Vector } \rightarrow \text{Vector } \text{[assoc comm prec 33]} . \\
\text{op } _\text{dot} : \text{Vector Vector } \rightarrow \text{Poly } \text{[prec 31]} . \\
\text{op } _\text{m*} : \text{Matrix Vector } \rightarrow \text{Vector } \text{[prec 31]} . \\
eq (V_1 v+ V_2) \text{ dot } V_3 = (V_1 \text{ dot } V_3) p+ (V_2 \text{ dot } V_3) . \\
eq V_3 \text{ dot } (V_1 v+ V_2) = (V_3 \text{ dot } V_1) p+ (V_3 \text{ dot } V_2) .
\]

\[\text{4.2 Formalization of honest parties}\]

Two constructors for the two kinds of messages used in Kyber are as follows:

\[
\text{op msg1 } : \text{Prin Prin Prin Vector Poly MsgState } \rightarrow \text{Msg [ctor]} . \\
\text{op msg2 } : \text{Prin Prin Prin Vector Poly MsgState } \rightarrow \text{Msg [ctor]} .
\]

where \text{Prin} is the sort representing principals, and \text{Msg} is the sort denoting messages. \text{MsgState} is the sort representing message states, receiving one of the following three values: \text{sent} - the message was sent, \text{replied} - the message was sent and the receiver replied with another message, and \text{intercepted} - the message was intercepted by the intruder. The first, second, and third arguments of each of \text{msg1} and \text{msg2} are the actual creator, the seeming sender, and the receiver of the corresponding message. The first and last arguments are meta-information that is only available to the outside observer, while the remaining arguments can be seen by every principal. The fourth and fifth arguments of \text{msg1} carry \(t\) and \(\rho\), respectively, since the first message of Kyber sends the public key \(\text{pk}\) which consists of \(t||\rho\). Similarly, the fourth and fifth arguments of \text{msg2} carry \(c_1\) and \(c_2\), respectively, of CPAPKE.

We model the network as a multiset of messages, in which the intruder can use as his/her storage. Consequently, the empty network (i.e., the empty multiset) means that no messages have been sent. The intruder can fully control the network, that is he/she can intercept any message, glean information from it, and fake a new message to any honest party. To formally specify Kyber in Maude, we use the following observable components:

- \((\text{nv : msgs})\) - \text{msgs} is the soup of messages in the network;
- \((\text{keys\{p\} : keys})\) - \text{keys} is a soup of the computed shared keys of principal \(p\). Each entry of \text{keys} is in form of \text{key}(K,q), where \(K\) is the shared key and \(q\) is the principal whom \(p\) believes that he/she has communicated with;
- \((\text{prins : ps})\) - \text{ps} is the collection of all principals participating in the protocol;
- \((\text{d\{p\} : d_0})\) - \(d_0\) is the random seed \(d\) (used in Fig. 2) of principal \(p\);
- \((\text{m\{p\} : m_0})\) - \(m_0\) is the random seed \(m_0\) (shown in Fig. 1) of principal \(p\);
- \((\text{rd-d : rdds})\) - \text{rdds} is a list of available values as the random seed \(d\) (we use list, but not set, to reduce the state space for searching). Each time when a principal makes a query for a random value of \(d\), the top value in \text{rdds} is removed and returned to the principal;
• (rd-m : rdms) - rdms is a list of the available values as random seed m0;
• (glean-keys : gkeys) - gkeys is the soup of shared keys gleaned by the intruder;
• (ds : ds) - ds is the collection of the random seeds d used by the intruder.
Note that every entry in ds is different from any random value used by honest parties;
• (ms : ms) - ms is the collection of random seeds m used by the intruder.
Similarly to ds, every entry in ms is different from any random value used by honest parties.

Each state in $S_{Kyber}$ is expressed as \{obs\}, where obs is a soup of those observable components. We suppose that there are two honest principals alice and bob together with a malicious one, namely eve, participating in Kyber KEM.
The initial state init of $I_{Kyber}$ is defined as follows:

\[
{(nw: \text{empty}) \ (prins: (alice \ bob \ eve)) \ (rd-d: (d1, d2)) \ (rd-m: (m1, m2)) \ (keys[alice]: \text{empty}) \ (keys[bob]: \text{empty}) \ (glean-keys: \text{empty}) \ (d[alice]: 0) \ (d[bob]: 0) \ (m[alice]: 0) \ (m[bob]: 0) \ (ds: \text{empty}) \ (ms: \text{empty})}.
\]

With the honest parties, we specify three transitions: keygen, encaps, and decaps, which correspond to the three steps of the mechanism. Let OCs be a Maude variable of observable component soups, A, B, and C be Maude variables of principals (possibly intruder), and PS be a Maude variable of principal collections.
Let D, M, M2, Rho, RhoA, V, V', CV, and P1 be Maude variables of polynomials, and PoL be a Maude variable of polynomial lists. Let G and H denote the hash functions G and H, respectively. Let MS be a Maude variable of networks (i.e., message soups). The rewrite rule keygen is defined as follows:

\[
\text{crl [keygen]} : \{(rd-d: (D, PoL)) \ (d[A]: P1) \ (prins: (A \ B \ PS)) \ (nw: MS) \ OCs} => \{(rd-d: PoL) \ (d[A]: D) \ (prins: (A \ B \ PS)) \ (nw: (msg1(C,A,B,T,Rho,sent) MS)) \ OCs} \text{if } \text{RhoSig} := G(D).
\]

where RhoSig is a Maude variable denoting a pair of polynomials, 1st and 2nd are its projection operators. sample-A, sample-e, and sample-s represent the sampling procedures, outputting the matrix A, the vectors e, and s, respectively.
The rewrite rule says that when there exists a polynomial D in rd-d, A picks it as a random seed d, builds a message msg1 exactly following the KeyGen() step of the mechanism, and sends it to B. d[A] is set to D, and D is removed from rd-d.

The rewrite rule encaps is defined as follows:

\[
\text{crl [encap]} : \{(rd-m: (M, PoL)) \ (m[B]: P1) \ (keys[B]: KS) \ (nw: \text{msg1}(C,A,B,T,Rho,sent) MS) \ OCs} => \{(rd-m: PoL) \ (m[B]: M) \ (keys[B]: (KS \ \text{key}(KDF(1st(Kr), H'(CV,CV)), A))) \ (nw: \text{msg1}(C,A,B,T,Rho,replied) msg2(B,B,A, CU, CV, sent) MS) \ OCs} \text{if } \text{M'} := H(M) \ \text{\backslash Kr := G(pair(M', H'(T, Rho)))} \ \text{\backslash} \ \text{CU := enc-u(T, Rho, M', 2nd(Kr))} \ \text{\backslash CV := enc-v(T, Rho, M', 2nd(Kr))}.
\]

35
where \( T \) and \( CU \) are Maude variables of polynomial vectors, \( Kr \) is a Maude variable denoting a pair of polynomials, and \( KS \) is a Maude variable representing a soup of shared keys. Let \( Rseed \) be a Maude variable of polynomials. Let \( \text{sample-}r \), \( \text{sample-}e_{1} \), and \( \text{sample-}e_{2} \) represent the procedures sampling \( r \), \( e_{1} \), and \( e_{2} \), respectively. Following the CPAPKE.\( Enc(pk,m,r) \) in Fig. 2, \( \text{enc-}u \) and \( \text{enc-}v \) are defined as follows:

\[
\begin{align*}
\text{eq } \text{enc-}u(T,Rho,M,Rseed) &= \text{compr}(\text{tp}(\text{sample-}A(Rho)) \times \text{sample-}r(Rseed) \times \text{sample-}e_{1}(Rseed), du) . \\
\text{eq } \text{enc-}v(T,Rho,M,Rseed) &= \text{compr}(\text{tp}(T) \times \text{dot} \times \text{sample-}r(Rseed) \times p + \text{sample-}e_{2}(Rseed) \times p + \text{decompr}(M,1), dv) . 
\end{align*}
\]

where \( \text{tp}(A) \) denotes the transpose matrix of \( A \) and \( \text{tp}(T) \) denotes the transpose vector of \( T \). \( du \) and \( dv \) are constants of natural numbers, denoting \( du \) and \( dv \), respectively. \( \text{decompr}(M,1) \) and \( \text{compr}(M,1) \) denote \text{Decompress}_{q}(M,1) and \text{Compress}_{q}(M,1), respectively. \( \text{enc-}u(T,Rho,M,Rseed) \) and \( \text{enc-}v(T,Rho,M,Rseed) \) compute \( c_{1} \) and \( c_{2} \), respectively, in Fig. 2 given as inputs \( T|Rho, M, Rseed \). The rewrite rule \text{encaps} says that when there exists a message \( \text{msg1} \) sent from \( A \) to \( B \) in the network, \( B \) builds a message \( \text{msg2} \) exactly following the \text{Encaps()} step of Kyber, sends it back to \( A \). \( B \) also computes the shared key with \( A \), and the state of the message \( \text{msg1} \) is updated to \text{replied}.

The rewrite rule \text{decaps} is defined as follows:

\[
\begin{align*}
\text{crl } \text{[decaps]} : \{(d[A] : D) \text{ (keys[A] : KS)} \\
\quad (nu: (\text{msg1}(A,A,B,T,Rho,\text{MsgStat}) \text{ msg2}(C,B,A, CU, CV, sent) MS)) \text{ OCs}) \Rightarrow \{(d[A] : D) \text{ (keys[A] : (KS key(KDF(1st(Kr2), H'(CU,CV)), B)))} \\
\quad (nu: (\text{msg1}(A,A,B,T,Rho,\text{MsgStat}) \text{ msg2}(C,B,A, CU, CV, replied) MS)) \text{ OCs}) \text{ if RhoSig := G(D) } \land \text{ Rho == 1st(RhoSig) } \land \text{ T == sample-}A(Rho) \times \times \text{sample-}s(2nd(RhoSig)) \times \times \text{sample-}e(2nd(RhoSig)) / \times \text{U'} := \text{decompr}(C, du) / \times \text{V'} := \text{decompr}(C, dv) / \times \text{M2 := compr}(V' \times \times \text{tp}(\text{sample-}s(2nd(RhoSig))) \times \times \text{dot} \times \text{U'}) / \times \text{Kr2 := G(pair(M2, H'(T, Rho)))} / \times \text{enc-}u(T,Rho,M2,2nd(Kr2)) = CU / \times \text{enc-}v(T,Rho,M2,2nd(Kr2)) = CV . \}
\end{align*}
\]

where \( \text{MsgStat} \) is a Maude variable representing an arbitrary message state. The rewrite rule says that when \( A \) has sent a message \( \text{msg1} \) to \( B \) and there exists a message \( \text{msg2} \) replied from \( B \) to \( A \) in the network, \( A \) follows the \text{Decaps()} step of Kyber, computes the shared key with \( B \). We only consider the overwhelming case, i.e., Alice successfully recovers \( m \). We assume that the error tolerance gaps made by error components always be silent, making \( m' \) equals to \( m \). This is done by the following equation:

\[
\text{ceq } \text{compr}(E0 \times p + \text{decompr}(M,1), 1) = M \text{ if isSmall?(E0) .}
\]

where \( E0 \) is a variable of sort Poly, and \text{isSmall?}(E0) is a predicate, returning true if all coefficients of \( E0 \) are small in comparison with \( q \). Sampling procedures for \( s, e, r, e_{1}, \) and \( e_{2} \) return vectors or polynomials whose coefficients are small. These properties are specified by the following equations:
Using Eq. 3, we rewrite \( \text{Decompress} \left( \text{Compress}(v, dv), dv \right) \) and \( \text{Decompress}(\text{Compress}(u, du), du) \) by \( v + \epsilon_1 \) and \( u + \epsilon_2 \), respectively, where all coefficients of \( \epsilon_1 \) and \( \epsilon_2 \) are small in comparison with those of \( v \) and \( u \). In the specification, we specify \( \epsilon_1 \) as \( \text{epsilon1}(v) \), \( \epsilon_2 \) as \( \text{epsilon2}(u) \), and both \( \text{epsilon1}(v) \) & \( \text{epsilon2}(u) \) are “small”. This is done by the following equations:

\[
\begin{align*}
\text{eq decompr}(\text{compr}(V,dv),dv) &= V p+ \text{epsilon1}(V) . \\
\text{eq decompr}(\text{compr}(U,du),du) &= U v+ \text{epsilon2}(U) . \\
\text{eq isSmall?}(\text{epsilon1}(V)) &= true . \\
\text{eq isSmall?}(\text{epsilon2}(U)) &= true .
\end{align*}
\]

### 4.3 Formalization of intruders

We suppose that there is one intruder, namely \( \text{eve} \), participating in the mechanism. When there exists a message \( \text{msg1} \) sent from \( A \) to \( B \) in the network, the intruder can intercept that message, fake a new message, and send it to the receiver. This behavior is specified by the following rewrite rule:

\[
\begin{align*}
\text{crl [keygen-eve]} : \{(ds: (D \text{PoC1})) (nw: (\text{msg1}(A,A,B,TA,\text{RhoA},\text{sent}) \text{MS})) \text{OCs}\} \\
\Rightarrow \{(ds: (D \text{PoC1})) (nw: (\text{msg1}(A,A,B,TA,\text{RhoA},\text{intercepted}) \text{MS})) \text{OCs}\} \\
\text{if } \text{RhoSig} := G(D) .
\end{align*}
\]

where \( \text{PoC1} \) and \( \text{PoC3} \) are Maude variables representing arbitrary soups of polynomials. The intercepted message must have state \text{sent} at the beginning, which means that the message has not reached the receiver. \( \text{eve} \) then constructs a new faking message from an available value \( D \) for the random seed \( d \). This kind of random value cannot be gleaned from the network, but \( \text{eve} \) can only construct it by randomly choosing a new value as the rewrite rule \text{build-ds} as follows:

\[
\begin{align*}
\text{rl [build-ds]} : \{(rd-d: (D, \text{PoL})) (ds: \text{PoC1}) \text{OCs}\} \\
\Rightarrow \{(rd-d: \text{PoL}) (ds: (\text{PoC1} D)) \text{OCs}\} .
\end{align*}
\]

Similarly, the only way in which \( \text{eve} \) can construct values for the random seed \( m \) is by randomly choosing a new value. This is specified by the following rewrite rule \text{build-ms}:

\[
\begin{align*}
\text{rl [build-ms]} : \{(rd-m: (M, \text{PoL})) (ms: \text{PoC3}) \text{OCs}\} \\
\Rightarrow \{(rd-m: \text{PoL}) (ms: (\text{PoC3} M)) \text{OCs}\} .
\end{align*}
\]

Two more rewrite rules are introduced as follows:

\[
\begin{align*}
\text{crl [encaps-eve]} : \{(ms: (M \text{PoC3})) (\text{glean-keys}: \text{KS}) \\
(nw: (\text{msg1}(A,A,B,TA,\text{RhoA},\text{intercepted}) \text{MS})) \text{OCs}\} \\
\Rightarrow \{(ms: (M \text{PoC3})) (\text{glean-keys}: (\text{key}(\text{KDF}(\text{1st(Kr)},H'(\text{CU,CV})),A) \text{KS})) \text{OCs}\}
\end{align*}
\]
\( M' := H(M) \land Kr := G(pair(M', H'(TA, RhoA))) \land \\
CU := enc-u(TA, RhoA, M', 2nd(Kr)) \land CV := enc-v(TA, RhoA, M', 2nd(Kr)) \land \\
msg2(eve, B, A, CU, CV, sent) \in MS = false \land \\
msg2(eve, B, A, CU, CV, replied) \in MS = false. \)

crl \( [\text{decaps-eve}] : \{(ds: (D PoC1)) (glean-keys: KS) \}
\( \rightarrow \{(ds: (D PoC1)) (glean-keys: (key(KDF(1st(Kr2), H'(CUB, CVB)), B) KS)) \}
\( \rightarrow \{msg1(eve, A, B, T, Rho, replied) \land msg2(B, B, A, CUB, CVB, sent) \in MS\} \}
\( \rightarrow \{msg1(eve, A, B, T, Rho, replied) \land \\
msg2(B, B, A, CUB, CVB, intercepted) \in MS\} \)
\( \text{if RhoSig} := G(D) \land Rho == 1st(RhoSig) \land \\
T == \text{sample-A}(Rho) \land \text{sample-s}(2nd(RhoSig)) \land \text{sample-e}(2nd(RhoSig)) \land \\
UB' := \text{decompr}(CUB, du) \land VB' := \text{decompr}(CVB, dv) \land \\
M2 := \text{compr}(VB' p- tpV(\text{sample-s}(2nd(RhoSig))) \text{ dot UB'}, 1) \land \\
Kr2 := G(pair(M2, H'(T, Rho))) \land \\
enc-u(T, Rho, M2, 2nd(Kr2)) == CUB \land enc-v(T, Rho, M2, 2nd(Kr2)) == CVB. \)

\( \text{encaps-eve} \) says that when \( \text{eve} \) has intercepted a message \( \text{msg1} \) sent from \( A \) to \( B \), \( \text{eve} \) fakes a new message \( \text{msg2} \), sends it to \( A \), and computes a shared secret key with \( A \). \( \text{decaps-eve} \) says that when \( \text{eve} \) has faked a new message \( \text{msg1} \), sent it to \( B \), and \( B \) on his/her belief that the message truly comes from \( A \) has replied to \( A \) a message \( \text{msg2} \), \( \text{eve} \) intercepts the message \( \text{msg2} \), and computes a shared secret key with \( B \).

5 Model checking and Man-In-The-Middle-Attack

We introduce the following search command:

\( \text{search [1] in KYBER : init} =>* \\
\{\text{keys[alice]: key(K1,bob)} \land \text{keys[bob]: key(K2,alice)} \land \text{glean-keys: (key(K1,alice) key(K2,bob) KS)}\} \)

where \( K1 \) and \( K2 \) are Maude variables that denote arbitrary shared keys. \( K1 \) may or may not be equal to \( K2 \). The command tries to find a state reachable from \( \text{init} \) such that: \( \text{alice} \) in her belief obtains the shared key \( K1 \) with \( \text{bob} \), \( \text{bob} \) in his belief obtains the shared key \( K2 \) with \( \text{alice} \), and \( \text{eve} \) owns both \( K1 \) and \( K2 \). Maude found a counterexample, and this kind of vulnerability belongs to MITM attacks. Fig. 3 shows how this attack happens on Kyber, which is visualized from the path leading to the counterexample Maude returned. There are mainly five steps as follows:

**Step 1** Alice wants to construct a shared key with Bob, she starts by performing \( \text{KEM.KeyGen()} \), generating a public key \( pk \) and a secret key \( sk \). She keeps \( sk \), and sends \( pk \) to Bob.

**Step 2** Eve intercepts the first message sent from Alice to Bob. She takes a random \( d_e \), follows the \( \text{KEM.KeyGen()} \) step to generate a pair \( (pk_e, sk_e) \), and sends \( pk_e \) to Bob.
Alice

\(z \leftarrow \mathbb{B}^{32}\)

\((pk, sk') = \text{PKE.KeyGen()}\)

\(sk = (sk' || pk || H(pk) || z)\)

\(\text{return } (pk, sk)\)

Bob

\(d_e \leftarrow \mathbb{B}^{32}\)

\((\rho_e, \sigma_e) = \text{G}(d_e)\)

\(R_q \times k \ni A_e = \text{generate}(\rho_e)\)

\(R_q \ni s_e, e_e \leftarrow \text{sampleCBD}(\sigma_e)\)

\(t_e = A_e s_e + e_e\)

\(pk_e = (t_e || \rho_e)\)

\(sk_e = s_e\)

\(\text{return } (pk_e, sk_e)\)

Eve

\(m_0 \leftarrow \mathbb{B}^{32}\)

\(m = H(m_0)\)

\((K, r) = \text{G}(m || H(pk_e))\)

\(c = \text{PKE.Enc}(pk_e, m, r)\)

\(K_b = \text{KDF}(K, H(c))\)

\(\text{return } (c, K_b)\)

\(m' = \text{PKE.Dec}(c, sk_e)\)

\((K'_e, r'_e) = \text{G}(m'||H(pk))\)

\(c'_e = \text{PKE.Enc}(pk, m', r'_e)\)

if \(c = c'_e\) then

\(K_a = \text{KDF}(K_e, H(c_a))\)

\(\text{return } (c_a, K_a)\)

\(m' = \text{PKE.Dec}(c_e, s_e)\)

\((K'_e, r'_e) = \text{G}(m'||H(pk))\)

\(c'_e = \text{PKE.Enc}(pk, m', r'_e)\)

if \(c_e = c'_e\) then

\(K_a = \text{KDF}(K'_e, H(c_a))\)

\(\text{return } K_a = \text{KDF}(K'_e, H(c_a))\)

Fig. 3. A counterexample found by Maude (note that we use PKE as an abbreviation for CPAPKE to save space)

**Step 3** Bob receives \(pk_e\) thinking it is from Alice. As a response, he takes a random \(m_0\), performs KEM.Enc\((pk_e)\), and obtains a ciphertext \(c\) and a shared key \(K_b\). He sends the ciphertext \(c\) back to Alice, and keeps the key \(K_b\), which he believes that it is the shared key obtained by him and Alice.

**Step 4** Eve intercepts the replied message which contains ciphertext \(c\) sent from Bob to Alice. She first performs KEM.Dec\((c, sk_e)\) to obtain the shared key \(K_b\). She then takes a random \(m_{a0}\), performs KEM.Enc\((pk)\), and obtains a ciphertext \(c_e\) and a shared key \(K_a\). She sends the ciphertext \(c_e\) back to Alice as a response for the first message.

**Step 5** Alice receives the ciphertext \(c_e\) thinking it is from Bob. She performs KEM.Dec\((c_e, sk)\) to obtain the shared key \(K_a\). She believes that \(K_a\) is the shared key obtained by her and Bob.
The reachable state space in the experiment is finite. Indeed, if we try to run the following command: `search in KYBER : init =>* {OCs} .`, the number of returned solutions is finite, implying that the state space is finite. This can be understandable because of the following explanation. Each state is denoted as a braced associative-commutative soup of the ten observable components as shown in Sect. 4. The key point is that the numbers of possible values that each observable component (i.e., a name-value pair) can receive is finite. Indeed, `ps in (prins : ps)` is always `(alice bob eve)` because there is no rewrite rule that changes it. `rdds` and `rdms` in `(rd-d : rdds)` and `(rd-m : rdms)` never consist of more than `(d1,d2)` and `(m1,m2)`, respectively, because there is no rewrite rule that inserts element(s) into them. `d0` and `m0` in `(d[p] : d0)` and `(m[p] : m0)` can only be in the sets `{d1, d2}` and `{m1, m2}`, respectively. Similarly, the numbers of possible values for `ds` and `ms` in `(ds : ds)` and `(ms : ms)` are finite. `msgs` in `(nw : msgs)` consists of finite messages because (1) each of the two rewrite rules `keygen` and `encaps` adds a new message into the network, but simultaneously it also removes one element from `rdds` and `rdms` (note that `rdds` and `rdms` never consist of more than `(d1,d2)` and `(m1,m2)` as shown above); (2) the rewrite rule `keygen-eve` adds a new message `msg2` into the network, but simultaneously it also changes the status of an existing message `msg1` from `sent` to `intercepted`, thus, `keygen-eve` can only be applied finite times; and (3) the rewrite rule `encaps-eve` only adds a new message `msg2` into the network if that message does not exist before (note that the other rewrite rules do not change the network or only update the status of messages). Similarly, `keys in (keys[p] : keys)` consists of finite entries because: the rewrite rule `encaps` removes one element from `rdms`; and the rewrite rule `decaps` changes the status of an existing message `msg2` from `sent` to `replied`. In the same manner, we can show `gkeys` in `(glean-keys : gkeys)` is finite. In summary, we can conclude that the state space in our experiment is finite. Consequently, with a search command to find a state satisfying some conditions, in finite time Maude will either find no solutions or will find a state satisfying the conditions.

Remark. Readers may argue that this kind of attack is not a novel attack since Kyber KEM does not go along with any solution for authentication. We agree on it. The paper instead illustrates one symbolic approach for reasoning KEMs rather than focusing on this kind of attack. Our ultimate goal is to come up with a new security analysis/verification technique for post-quantum cryptographic protocols, such as post-quantum TLS. Such protocols use post-quantum cryptographic primitives, such as KEMs. Formally specifying such primitives is necessary to analyze the security. What is described in the paper is our initial step toward the goal.

Saber and SK-MLWR. In the same manner, we formalize the two other KEMs, i.e., Saber [10] and SK-MLWR [1], specify them in Maude, and run Maude search command trying to find the same kind of attack. The same MITM attacks are found by Maude. The reason is similar to the Kyber case study, that is because there is no authentication, and thus, the intruder can impersonate any party. Because of the page limitation, we cannot present these two case studies in
6 Conclusion

We have presented an approach to security analysis of some lattice-based KEMs in the symbolic model. We first used Maude as a specification language to formally specify the KEMs. After that, by employing Maude search command, an MITM attack was found for each KEM. The occurrence of the attack is basically because a KEM alone does not come with an authentication solution.

Researchers have proposed a post-quantum TLS protocol [8] that uses a hybrid key exchange method: a traditional key exchange algorithm together with a post-quantum KEM. The reason why a post-quantum KEM is required is clear. However, why do we still need to employ a traditional key exchange algorithm? One reason is that most post-quantum KEMs are not studied/analyzed deeply, and thus, nothing guarantees that there is not any potential flaw in them. Thus, deep security analysis of such KEMs in particular and other post-quantum cryptographic primitives/protocols is an important challenge to guarantee their reliability. One piece of our future work is to formally verify the security of the post-quantum TLS protocol against both classical and quantum computers. To this end, the most important task is to come up with a new intruder model because intruders will be able to utilize quantum computers on which quantum algorithms, such as Shor’s one [23], run in the post-quantum era.

References


42


Business Processes Analysis with Resource-aware
Machine Learning Scheduling in Rewriting Logic

Francisco Durán\textsuperscript{1}, Daniela Martínez\textsuperscript{2}, and Camilo Rocha\textsuperscript{2}

\textsuperscript{1} ITIS Software, Universidad de Málaga, Málaga, Spain
\textsuperscript{2} Dept. of Electronics and Computer Science, Pontificia Universidad Javeriana, Cali, Colombia

Abstract. A significant task in business process optimization is concerned with streamlining the allocation and sharing of resources. This paper presents an approach for analyzing business process provisioning under a resource prediction strategy based on machine learning. A timed and probabilistic rewrite theory specification formalizes the semantics of business processes. It is integrated with an external oracle in the form of a long short-term memory neural network that can be queried to predict how traces of the process may advance within a time frame. Comparison of execution time and resource occupancy under different parameters is included for a case study, as well as details on the building of the machine learning model and its integration with Maude.

1 Introduction

Business process optimization is the practice of increasing organizational efficiency by improving processes. The main motto of this area in business process management is that optimized processes lead to optimized business goals. Since efficiency is one of major quantitative tools in industrial decision making, the most common goals in process optimization are maximizing throughput and minimizing costs. For instance, companies that use a business process for a long time can greatly benefit from increasing the usage of resources within reasonable limits of redundancy and costs, and streamlining workflows. However, business process optimization tends to be a multi-objective optimization problem in which many variables can be involved. One main challenge is to integrate predictive tools at design stages for business process optimization.

Deep learning models are becoming increasingly important in business applications because they serve as a basis for monitoring and predicting process behavior (see, e.g., [10, 12]). In particular, applications concerned with resource allocation, time and cost optimization, fault monitoring, and process discovery are using event logs to train these models and predict variables of interest. Logs usually contain information about processes in an organization as a list of events. Each event is associated to a process instance, which is identified by a case number. A case is seen as a collection of activities or tasks with attributes. Typically, they include a name or case number, a timestamp, and the resources or costs associated to it, among other attributes. Recurrent neural networks [14] have been widely used for performing sequence prediction. They are trained to learn from information in event logs and predict the next events that are more likely to occur based on the gained knowledge.
This paper proposes a two-layered approach to formal process optimization. The first layer uses the prediction power of deep learning models to help anticipate the demand of resources (i.e., number of replicas) in a business process from a partial execution. The second layer, on top of the first one, formally specifies the concurrent behavior of many instances of the process that compete for the same collection of resources and replicas. The result is then the integration of these two layers, resulting in a sophisticated heuristic-oriented technique for the formal analysis and optimization of business processes. Their quantitative analysis can then be carried out by, e.g., computing the best combination of parameter values reducing the costs or processing time.

Long short-term memory neural networks (LSTM) [14], a type of recurrent neural network, are used for sequence prediction as proposed in [11]. More precisely, given a business process $B$ and a partial trace $t$ of tasks in $B$, an LSTM can predict an extension of $t$ that conforms to $B$ based on a (previous) training with event logs obtained for $B$. A rewriting logic semantics of the Business Process Modeling Notation (BPMN) simulates the concurrent behavior of many instances of $B$, under a given set of constraints over the resources, by querying the LSTM as a scheduler and adjusting the number of replicas accordingly. One novelty of the proposed approach is that resource allocation can happen at execution time based, not only on the current state of execution of a given instance of $B$, but also on some history of previous executions of the process.

For training an LSTM, the approach takes as input a BPMN process $B$ and a set of traces $T$ of $B$. The traces $T$ represent executions of $B$ in, e.g., a production environment. The process $B$ has information about the type of resources needed to complete the tasks. Therefore, the structure of $B$ and the traces are used to predict how resources are to be allocated/released in order to optimize their use: e.g., minimize the time a resource is not being used or, similarly, maximize the usage of resources meeting some budget and timing constraints associated to the process’ execution.

The LSTMs used in the proposed approach have been implemented in Python with Keras and TensorFlow. The integration with Maude is designed via socket communication, where the rewriting logic semantics is the client of the prediction server written in Python. From the partial concurrent execution of a given number of instances of a BPMN process in the Maude semantics, the neural network is queried with a time window. It then returns a sequence of events that extends the traces of the given instances, i.e., make a prediction on their continuation. This prediction is then used to adjust the number of replicas per resource at runtime. The analysis is based on the simulation of the execution guided by such trace. The percentage usage of resources and the number of replicas during the time span of all replicas, among other, are monitored and summarized for each experiment.

Outline. Sections 2 and 3 present overviews, respectively, of BPMN and LSTM neural networks. Section 4 presents the rewriting semantics of BPMN and its interaction with the LSTMs. A case study is presented in Section 5, while Section 6 concludes the paper.

2 The Business Process Modeling Notation (BPMN)

BPMN is a graphical notation for modeling business processes as collections of related tasks that produce specific services or products. In BPMN, processes are modeled using
graphical representations for tasks and gateways, which are connected through flows and events. In this work, the focus is on its control flow constructs, including the most common types of tasks, events, and gateways.

To introduce and illustrate the use of the supported BPMN constructs, and the analysis techniques presented in this work, the process depicted in Figure 1 is used. It describes a parcel ordering and delivery. The process consists of three lanes: one for clients, one for the order management, and one for the delivery management. In this process, the client first signs in and then repeatedly looks for products. Eventually, the client can decide to give up or to make an order by submitting it to the order management lane. The client then waits for a response (i.e., acceptance or refusal of this order). However, the client waits for a response for a maximum amount of time, as is represented by a timer-event branch. If the order can be completed, then the parcel is received and the client pays for it. Otherwise (i.e., timeout or order refused), the client fills in a feedback form. As far as the management lane is concerned, the first task aims at verifying whether the goods ordered by the client are available. If they are not available, then the order is canceled; otherwise, the order is confirmed. The order management takes care of the payment of the order whereas the delivery lane is triggered to prepare the parcel to be delivered. The delivery may be carried out by car or by drone.

The initiation and finalization of processes are represented by initial and final events. Events are also used to represent the sending of messages and the firing of timers. A task represents an atomic activity that has exactly one incoming and one outgoing flow. A sequence flow describes two nodes executed one after the other, i.e., imposing an execution order between these nodes. Tasks may send messages, which in such a case activate the corresponding message flows.

Gateways are used to control the divergence and convergence of the execution flows. In this work, exclusive, inclusive, parallel, and event-based gateways are supported. Gateways with one incoming branch and multiple outgoing branches are called splits (e.g., split inclusive gateway). Gateways with one outgoing branch and multiple incoming branches are called merges (e.g., merge parallel gateway). An exclusive gateway chooses one out of a set of mutually exclusive alternative incoming or outgoing
branches. For an inclusive gateway, any number of branches among all its incoming or outgoing branches may be taken. A parallel gateway creates concurrent flows for all its outgoing branches or synchronizes concurrent flows for all its incoming branches. For an event-based gateway, it takes one of its outgoing branches or accepts one of its incoming branches based on events.

In addition to the description of specific tasks and their sequencing, collaboration diagrams also involve pools and lanes, which are structuring elements that split processes into pieces. In BPMN, each lane in a collaboration diagram corresponds to a specific role or resource. However, other resources may also be involved, and tasks could require multiple resources or instances of the same resource. Therefore, instead of implicitly associating resources to lanes, in our approach, resources are explicitly defined at the task level. Hence, a task that requires resources for its execution can include, as part of its specification, the required resources. To do it graphically, symbols are associated to each resource type, and these symbols are depicted inside the corresponding tasks. For example, the process in Figure 1 relies on clerks for the handling of customers’ orders, workers for parcel packing, and couriers for car delivery. In addition, cars and drones are used to deliver the parcels. For instance, the diamonds at the right-top corners of the Check availability, Cancel order, and Confirm order tasks indicate that one instance of the clerk resource is required for the execution of the tasks. Task Deliver by car requires instances of the car and courier resources. To avoid dealing with multiple units of measurement, resources are counted as instances or replicas, and if more than one instance of a certain resource type is required, they are depicted as a number of icons in the task.

The process evolves by successively executing its tasks. However, the execution of a task requires the specified amounts of resources, which may lead to a competition for such resources: multiple instances of the process may also run concurrently, and multiple tasks in the same run may require the same resources. In our running example, e.g., clerks are used in several tasks, and multiple customers may be trying to simultaneously purchase products.

3 Using Long Short-Term Memory Neural Networks

Recurrent neural networks (RNNs) are a type of neural networks used to process and predict sequential data [14]. They consist of a set of neurons connected with each other, where each neuron has an input $x_t$ and output $h_t$ at a specified timestamp $t$, as well as a feedback loop that provides information of the previous timestamp. LSTM is a type of recurrent neural network used for sequence prediction. In particular, they are useful for solving problems involving long sequences that require previous information for larger periods of time. The main characteristic of this type of networks is their architecture composed by feedback loops to maintain information over time and a set of gates that control the flow of information into a memory cell. Memory cells enables the learning of longer patterns using a group of gates including the forget, input, output gates.

The model used in this work consists of a 2-layer LSTM with one shared layer. The main layers are in charge of activities and timestamp predictions. The data from event logs is transformed into a vector representation using one-hot encoding (i.e., a
sequence of bits among which the legal combinations of values are only those with a single 1): each vector has a 1 in the location that matches the corresponding task; its maximum length is given by the number of unique tasks found in the event logs. The vectors are then grouped in a sequence based on the time of their occurrence and this sequence is divided in a prefix and suffix. The prefix represents the activities known to have executed, and the suffix represent the activities to be predicted.

Once the data has been adapted to train the LSTM model, 80% of the available traces are used as training set and the remaining 20% as the testing set to evaluate the accuracy of the model. In each prediction, the LSTM model assigns a probability to all the tasks to decide which one will happen next. Furthermore, the prediction is adapted to take as input a number of tasks to predict.

4 Rewriting Logic Semantics with LSTM Integration

Two alternative ways of evolving business process models are presented. Namely, one guided by event logs, obtained from executions of the system, and another one, equipped with time information and probabilities modeling the actual behavior of the system, for its simulation. The second one is responsible for the simulation of the system and the managing and analysis of resources. It is also in charge of communicating with the Python process performing the predictions. The Maude process submits the event log of the activities carried out and, periodically, requests predictions. At this time the Maude BPMN process creates a replica of itself, which will be guided by the event sequence submitted by the predictor. Once the guidance consumes all the predicted events, the status of the resources is analyzed to decide on the allocation/releasing of replicas, and the simulation is restored. In summary, sharing a common representation of the core elements of processes, one specification defines the evolution of the process in accordance to the events in the log trace, and the other non-deterministically advances using the information with which models are annotated.

The information required by each of the specifications is not the same. To simulate the execution of processes, the process specifications are enriched with quantitative information. This additional information is added as annotations to the process model. Specifically, durations and delays associated to tasks and flows are expressed as stochastic expressions. Similarly, alternative constructs (split exclusive and inclusive gateways) are extended with probabilities associated to outgoing flows. Figure 2 shows the process given in Figure 1 enriched with such information.

Data-based conditions for split gateways are modeled using probabilities associated to outgoing flows of exclusive and inclusive split gateways. For instance, notice the exclusive split after the Search products task in the customer lane of the running example, which has outgoing branches with probabilities 0.6, 0.2, and 0.2, specifying the likelihood of following each corresponding path. The probabilities of the outgoing flows in an exclusive split must sum up to 1, while each outgoing flow in an inclusive split can be equipped with a probability between 0 and 1 without a restriction on their total sum.

The timing information associated to tasks and flows (durations or delays) is described either as a literal value (a non-negative real number) or sampled from a probability distribution function according to some meaningful parameters. The probability...
distribution functions currently available include exponential, normal/Gauss, and uniform (see, e.g., [13]). To simplify the reading of the process in Figure 2, the specification of task durations has been placed apart from the process description, at the bottom-left corner. In the modelling tool, these parameters would be specified as properties of the corresponding elements. For instance, the duration of the Sign in task is specified as $\text{Norm}(1, 0.5)$, which means that it follows a normal distribution with mean 1 and variance 0.5, and the Search products task follows a uniform distribution in the interval $[3, 30]$, that is specified as $\text{Unif}(3, 30)$. Also to simplify the specification of the process, the delays in all flows are set to $\text{Norm}(1.0, 0.2)$ to express that it takes some time to move from one task to the following one(s).

The specification builds on a specification that has evolved along different extensions through time [9, 2–8].

4.1 The Specification of BPMN Processes

In the Maude specification of BPMN, a process is represented as an object with sets of nodes and flows as attributes. The representation of each node type includes the necessary information to describe its structure and to contribute to the overall process analysis. For instance, a task node involves an identifier, a description, two flow identifiers (input and output), a stochastic function modeling its duration, a set of resources required for its execution, and a set of messages to be delivered after its completion. A split node includes a node identifier, a gateway type (exclusive, parallel, inclusive, or event-based), an input flow identifier, and a set of output flow identifiers. A merge node includes a node identifier, a gateway type, a set of input flow identifiers, and an output flow identifier. The representation of a flow includes a probability distribution function specifying its delay, and an optional message or timer. The message blocks the flow until it is received, whereas the timer represents a delay after which the execution is triggered.
Given unique identifiers for nodes, flows, resources, and events the process of the running example can be specified as shown in the excerpt in Figure 3. It shows how a Process object has attributes with the definition of its nodes and flows connecting them. For example, the exclusive split id("n005") (lines 5–6) has id("f004") as incoming flow, and id("f005"), id("f006"), and id("f007"), with associated probabilities 0.6, 0.2, and 0.2, respectively, as outgoing flows. Furthermore, the event-based split gate id("n007") (line 7) has id("f008") as incoming flow, and id("f009"), id("f010"), and id("f011") as outgoing flows. Note the definition of these flows in lines 17–19; after the corresponding delay, they become active upon the reception of the corresponding messages or by the id("timeout") timer firing. Finally, note that the specifications of tasks and flows also include their duration or delays as stochastic functions. For example, the duration of the Prepare parcel task follows a normal distribution with average 5 and variance 4, which is specified by the term Norm(5.0, 4.0). All flows are specified with Norm(1.0, 0.2) as second argument, stating that they all have a delay that follows a normal distribution with given parameters.

4.2 Autonomous Processes

A set of rewrite rules specifies how tokens evolve through a process. Each move of a token inside a BPMN process is modeled as a rewrite rule. E.g., one of the actions that may occur, and that is modeled by a corresponding rewrite rule, is that when there is a token in the incoming flow of an exclusive split, the token is moved to one of the outgoing flows of the gate, with its timer set to the value resulting from evaluating the stochastic expression of the flow, which represents the delay of the flow. Objects of classes Simulation, Workload, and Supervisor manage different aspects of the simulations. While process objects represent static processes, and they do not change along simulations, all the information on process execution is kept in simulation objects. Specifically, a Simulation object stores a collection of tokens (in a tokens attribute), a global time (gtime), a set of events (events, including messages and timers), and a set of resources (resources). It also keeps track of the metrics being computed. For analysis purposes,
during the execution of a process some information is collected in the corresponding
attributes: time stamps, task durations, and waiting time at parallel and inclusive merge
gateways. This information is necessary for guiding the execution of the process, peri-
odically evaluating the amount of resource instances, and for presenting the results to
the user for possible optimizations.

Tokens are used to represent the evolution of the workflow under execution. Since
there may be several simultaneous executions of a process, each execution is identified
with a unique identifier, which is used to associate tokens to executions. Thus, a token
is represented as a term token($\text{tid}$, $\text{id}$, $\text{t}$), where $\text{tid}$ is the execution instance the token
belongs to, $\text{id}$ is the identifier of the flow or node it is attached to, and $\text{t}$ represents a
timer, of sort $\text{Time}$, modeling a delay of the token, which represents the duration of a task
or the delay associated to a flow. Once its timer becomes 0, a token can be consumed.

Tokens are stored in the tokens attribute of the Simulation object — implemented
as a priority queue, so that tokens are processed according to their due time. However,
even if a token is at the front of the queue with timer 0, it may be required to delay its
execution. For example, consider a task that requires some resource that is not available,
or a parallel merge for which some incoming flow is not yet active. To avoid deadlocks,
the scheduler implements a shifting mechanism that identifies the first active token to
the front of the queue in case the current head needs to be delayed.

For each resource type, a number of instances or replicas are provisioned. At each
moment during a simulation, some of these instances can be in use and others can be
available for tasks to use them. Given a number of provisioned resources, if a running
task requires resources and they are available, it blocks them and initiates execution im-
mediately. Indeed, whenever a task requires several resource types, it atomically picks
them, or waits for all of them to be available. If the required resources are not all avail-
able, resource requests are submitted, and the task remains blocked until its requests are
satisfied. To support this, each resource type keeps a queue of requests.

Each resource type is represented by a resource operator that gathers all required
information: an identifier, the minimum and maximum number of allocatable replicas
(0, if unlimited), its allocation time, the total number of allocated replicas, the number
of available replicas, the total amount of time the replicas of this resource type have
been in use, and some historical information on resource usage, request queues, etc.,
which are handy for analysis purposes.

The execution of a task is modeled with two rules. The first rule, the initTask rule
shown in Figure 4, represents the task initiation, which is applied when a token with zero
time is available at the incoming flow (line 5). If all the resources required by this task
are available, which is checked with the allResourcesAvailable function (line 8), then a
new token is generated with the task identifier and the task duration (line 12). Otherwise,
the shifting mechanism is invoked (line 20) — note the ellipsis. If available, all required
resources are removed from the resource set (grabResources function, line 18). Note
also that rules update the information on execution times, task durations, etc. (see, e.g.,
the update of the task-ts tamps attribute, lines 13–16).

A second rule, which models task completion, is triggered when there is a token for
that task with zero time. In that case, the token is consumed and a new one is generated
for the outgoing flow. All resources are released, and all the message events associated to that task, if any, are added to the set of events.

The Simulation object is in charge of collecting the data on the chosen metric for the specified window of time (history length). A supervisor then analyzes the collected information and, if necessary, decides to update (increase or decrease) the number of resource instances. Simulation-based analysis techniques are typically parameterized by the workload, which defines the rate at which new instances of a given process are executed. In a closed workload, a fixed number of tokens will be injected in the process, corresponding to the number of times the process is to be executed.

Finally, a class CtrlSocket is in charge of the interaction with the predictor component. Every time a process starts or terminates, or the execution of a task begins or terminates, an event is sent to the predictor. As we will see in Section 4.4, when a prediction is due, the execution of the system stops, and a special event is sent to the predictor component to notify that a prediction is due.

### 4.3 Event-Guided Processes

Even though the representation of processes presented in Section 4.1 includes elements like probabilities, durations and delays that are not needed when the system is guided by a provided event log, the fact that both stages of the simulations — the autonomous execution described in Section 4.2 and the one described in this section — use the same representation greatly simplifies its specification, since although the control will be different, the process will just make a copy of itself to evolve. In fact, the main difference is that whilst in an autonomous process the execution is guided by tokens, inserted in the process by a workload manager object, in an event-guided process, it is the events who guide the execution.

In the context of business processes, event logs are collections of time-stamped events produced by the execution of business processes. Each event indicates the execution of a task of the process. For example, an event may specify that a given task
started or completed at a given time. Event logs are used for different purposes, including process mining, conformance checking, etc. They may be represented using different formats, like CSV, XES, MXML, XLSX or Parquet. Independently of the format chosen, they typically include fields for date and time, event identifier, source, and possibly others, in one way or another. Today, XES (eXtensible Event Stream) [1] is the most-widely-used standard for storing event logs.

Since an event may belong to any of the on-going execution sessions, and we do not require dates or any other information, instead of using any of the existing formats, our events are represented by sequences of events that include three values, separated by commas: a session identifier (a number), an event description, and a time-stamp. Event identifiers may be either initial, for the beginning of a process session, final, for the final node of a process session, or a task identifier followed by either -init or -end, representing the beginning and the end, respectively, of the execution of a task. For example, the following is a fragment of a sequence of events of our example:

- 298, n004-init, 3703784394059892335/9007199254740992
- 297, n014-init, 3705754467844272657/9007199254740992
- 356, n004-end, 1854135580310736077/4503599627370496
- 282, initial, 463663424722316089/1125899906842624
- 299, n004-end, 928653259810773583/2251799813685248
- 281, initial, 464495724782831233/1125899906842624
- 297, n014-end, 571624327443822323/9907199254740992

These event sequences are received through a socket, are parsed, and then represented using appropriate declarations. For example, the event

298. n004-init, 3703784394059892335/9007199254740992

is represented as

```
event(id("298"), id("n004"), event("init"), 3703784394059892335/9007199254740992)
```

An object of class Ctrl keeps the list of events, as is in charge of the interaction with the predictor, reading the sequence of events throw a socket, and then guiding the execution in accordance with such events.

```
class Ctrl | events: List{LogEvent}, socket: Maybe{Oid}, buffer: String .
```

Once in its events attribute, events will guide the execution by activating rules specifying the different actions that may occur in the system. For example, rule initTask in Figure 5 specifies the initiation of a task when an init task event is at the front of the event sequence. Although not shown in the rule to simplify the presentation, all the information on the execution (time stampts, resources, etc.) is gathered as in the autonomous simulations presented in Section 4.2. This information will be used, when the execution consumes all the events in the prediction, to update the number of instances of the resources. Note that the initRule mirrors quite closely that for the autonomous execution.

### 4.4 Resource Adaptation based on LSTM Predictions

The scheme proposed in [8] is followed for the definition of adaptation strategies. Among others, in [8], a strategy based on predictions was proposed. However, in that case, the prediction was carried out by using the execution of autonomous process itself, looking ahead before making a decision. Here, a process advances on its execution, but instead log traces are used for the prediction.
The general scheme assumes that resource instances are taken from a pool when required. However, instead of assuming a fixed number of instances, new instances may be allocated or released to adjust the offer of available resources to their demand, and in this way minimizing costs. The general scheme consists of periodically evaluating the amount of resources, by looking at different metrics on the recent history or the current state. To specify such a mechanism, a class `Supervisor` provides attributes `time-between-checks`, `time-to-next-check`, to keep a timer, and `check-interval` to specify the length of the history to look at.

```plaintext
```

This general procedure assumes that decisions are taken in accordance to some given thresholds, which are also provided as parameters. The algorithm periodically checks if the value of the considered property is greater than the upper-bound threshold, in which case a new instance of the resource is allocated to the set of available resources; if it is smaller than the lower bound, then an instance is removed so that it is no longer available for use.

The `Supervisor` class is extended in a subclass `SupervisorPrediction` to handle the new strategy. In addition to the thresholds attribute, with ranges for each resource type, it adds attributes `look-ahead-time`, to be able to consider different prediction sizes, and `forked-state`, to create an event-guided process to be executed on the prediction to be received from the Python predictor component.

```plaintext
class SupervisorPrediction |
    thresholds: Map{Id, Tuple{Float, Float}}, ---- usage thresholds
    look-ahead-time: Time
    forked-state: Maybe{System}.

class SupervisorPrediction < Supervisor.
```

The rules specifying the behavior of the supervisor object are shown in Figure 6. The supervisor-initiate-prediction rule is fired when the value of the time-to-next-check attribute is zero. It creates a copy of the part of the state needed for the event-guided execution (Section 4.3): the Simulation object collects information on the execution, including time-stamps and measure of resource usage, and the Process object. A new object of class CtrlSocket is created to read from the socket and collect the events to guide the execution. On the right-hand side of the rule there is a `send` message: a "PREDICT" event notifies the predictor that it is time to use the trace submitted until that time to feed the neural network, generate the prediction, and submit it through the socket.

To mark the end of the prediction, the predictor component will send an END event. When the CtrlSocket object in the forked-state attribute finds the END event, the second rule, supervisor-prediction-completed, is fired. It terminates the event-guided system and
updates the resources using the update function. This operation basically analyzes the resources along the execution of the prediction and decides whether changing the number of instances of each resource or not using the thresholds provided. Finally, notice that the tokens are restored in the Simulation object so that the simulation can be resumed. The time-to-next-check timer is reset with the value of the time-between-checks attribute.

5 Case Study

Table 1 presents experimental results, including the average and variance of the execution times, total cost, and resource usage for different parameters of the running example. In all these executions, (1) the population is 500; (2) the ranges for the different resources is [1, 2], except for drones, for which a range [1, 4] was chosen; (3) the allocation times (AT) go from 2 to 5 for the different resources; (4) similarly, resource costs are in the range 20-60; and (5) thresholds are fixed to 50 and 85. Of course, many other combinations are possible, but considering all of them involves many combinations, and should be handled as an optimization problem, and use some amenable technique for such a problem, such as genetic algorithms, or search-based algorithms like hill climbing or simulated annealing.

Although only a few combinations varying the time between checks (TBC) and the look-ahead time (LAT) are presented, some interesting observations can be made. TBC takes values 5, 10, 15, 20, and 25 in the different experiments. Given that the best average execution time is for case 5, with TBC 25, one may think that bigger TBCs may be better than smaller ones. However, note that executions 4 and 6, with TBCs 20 and 25, respectively, show the slowest executions, 125.95 and 119.81, respectively. Execution 5 has the best execution time but the worst cost, from those in the table. The smallest cost is shown by Case 3, with TBS 15 and LAT 5.

If both execution time average and cost are considered, the best combination is the one shown as case 1, with TBC 5 and LAT 5. The data collected along the simulations are used to generate charts with the evolution of resources. The charts for Case 1 are depicted in Figure 7.

6 Concluding Remarks

The results presented here are part of a long-standing effort to provide BPMN modeling with extensions and formal analysis tools [9, 2–8]. The novelty of this paper is in the integration of a rewriting logic semantics of BPMN and a deep learning scheduler for business process optimization. Both the semantics and scheduler have been presented, including details about their communication and usage, and illustrated with a running example. The reader is referred to [8] for a comprehensive summary of related work, complementing the references included throughout this paper.

Future work includes a detailed comparison of optimization heuristics, such as the ones presented in [8], with the one presented in this paper. Furthermore, new case studies need to be developed for such a comparison. Another future research direction is the use of deep learning techniques for business process optimization in a sense different
crl [supervisor-initiate-prediction]:
  < SId : Simulation | tokens: Tks, Atts1 >
  < PId : Process | Atts2 >
  < Sup : SupervisorPrediction |
    time-to-next-check: 0, ---- check is due
    forked-state: null,  
    look-ahead-time: T,  
    look-ahead-time: T,  
    Atts5 >
  < SH : SocketHandler | socket: SOCKET, Atts6 >
  => < SId : Simulation | Atts1 > ---- tokens are removed to stop the simulation  
  < PId : Process | Atts2 >
  < Sup : SupervisorPrediction |
  time-to-next-check: 0,  
  forked-state: { < SId : Simulation | Atts1 > ---- no tokens, so regular rules cannot apply  
                     < PId : Process | Atts2 >
                     < cs : CtrlSocket | socket: SOCKET, buffer: "", events: nil >
                     Receive(SOCKET, cs) },
  look-ahead-time: T,  
  tokens: Tks, ---- save tokens to restore the simulation  
  Atts5 >
  < SH : SocketHandler | socket: SOCKET, Atts6 >
  send(SOCKET, SH, "PREDICT " + string(PREDICTION-TIME, 10) + "\n")
  if Tks =/= nil .

rl [supervisor-prediction-completed]:
  < SId : Simulation | resources: Rs, gtime: T, Atts1 >
  < Sup : SupervisorPrediction |
  time-between-checks: TBC,  
  time-to-next-check: 0,  
  check-interval: CI,  
  thresholds: Thds,  
  forked-state: { < SId : Simulation | resources: Rs', gtime: T', Atts3 >  
                     < cs : CtrlSocket | buffer: "", events: END, Atts4 >  
                     Conf },
  tokens: Tks, ---- frozen tokens  
  Atts5 >
  => < SId : Simulation |
  resources: update(Rs, Rs', TBC, CI, T, T'),
  gtime: T,  
  tokens: Tks, ---- tokens are restored to resume the simulation  
  Atts1 >
  < Sup : SupervisorPrediction |
  time-between-checks: TBC,  
  time-to-next-check: TBC,  
  check-interval: CI,  
  thresholds: Thds,  
  forked-state: null,  
  Atts2 > .

Fig. 6. Supervisor’s rules.
<table>
<thead>
<tr>
<th>TBC</th>
<th>LAT</th>
<th>Resources</th>
<th>Usage (%)</th>
<th>Exec time (h)</th>
<th>Cost (€)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Name</td>
<td>Range</td>
<td>At</td>
<td>Cost</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>car</td>
<td>[1, 2]</td>
<td>5</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clerk</td>
<td>[1, 2]</td>
<td>4</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>courier</td>
<td>[1, 2]</td>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drone</td>
<td>[1, 2]</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>car</td>
<td>[1, 2]</td>
<td>5</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clerk</td>
<td>[1, 2]</td>
<td>4</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>courier</td>
<td>[1, 2]</td>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drone</td>
<td>[1, 2]</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>car</td>
<td>[1, 2]</td>
<td>5</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clerk</td>
<td>[1, 2]</td>
<td>4</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>courier</td>
<td>[1, 2]</td>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drone</td>
<td>[1, 2]</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>car</td>
<td>[1, 2]</td>
<td>5</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clerk</td>
<td>[1, 2]</td>
<td>4</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>courier</td>
<td>[1, 2]</td>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drone</td>
<td>[1, 2]</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>car</td>
<td>[1, 2]</td>
<td>5</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clerk</td>
<td>[1, 2]</td>
<td>4</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>courier</td>
<td>[1, 2]</td>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drone</td>
<td>[1, 2]</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>car</td>
<td>[1, 2]</td>
<td>5</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clerk</td>
<td>[1, 2]</td>
<td>4</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>courier</td>
<td>[1, 2]</td>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drone</td>
<td>[1, 2]</td>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 1. Outputs for some of the simulations carried out for the delivery example.
Fig. 7. Number of instances (left) and usage percentage (right) for each resource type for a simulation with the predictive-usage strategy, TBC=5, LAT=5, and Thds=(50, 85).
to the one explored here. Namely, deep learning methods can be used also for structural optimization of a process under some given constraints. Finally, the authors plan on making available a tool integrating most of the techniques and algorithms developed for BPMN formal analysis.

Acknowledgments. The authors would like to thank the reviewers for carefully reading the manuscript; their comments have been of great help in improving its quality and clarity. The first author has been partially supported by projects PGC2018-094905-B-100 and UMA18-FEDERJA-180, and by Universidad de Málaga, Campus de Excelencia Internacional Andalucía Tech. The third author was partially supported by the ECOS-NORD MinCiencias C19M03 project “FACTS: Foundational Approach to Computation in Today’s Society”.

References

1. IEEE Std 1849™-2016 standard for eXtensible event stream (XES) for achieving interoperability in event logs and event streams, September 2016.
Modeling, Algorithm Synthesis, and Instrumentation for Co-simulation in Maude

Simon Thrane Hansen\textsuperscript{1} and Peter Csaba Ölveczky\textsuperscript{2}

\textsuperscript{1} DIGIT, Department of Electrical and Computer Engineering, Aarhus University, Aarhus, Denmark
\textsuperscript{2} Department of Informatics, University of Oslo, Oslo, Norway

Abstract. Simulation-based analysis of cyber-physical systems is vital in the era of Industry 4.0. Co-simulation enables composing specialized simulation tools via a co-simulation algorithm. In this paper, we provide a formal model in Maude of co-simulation for complex scenarios involving algebraic loops and step negotiation. We show not only how Maude can formally analyze co-simulations, but also how Maude can be used to synthesize co-simulation algorithms, port instrumentations, and parameter values so that the resulting co-simulation satisfies desired properties.

1 Introduction

Modern cyber-physical systems (CPSs), such as, e.g., nuclear power plants, cars, and airplanes, consist of multiple heterogeneous subsystems that are developed by different companies using different tools and formalisms [23]. Although these companies usually do not share their models for commercial reasons, there is nevertheless a need to determine how the different subsystems interact and to explore and analyze different design choices as early as possible. One way of addressing this need is to use, for each subsystem, an interface that provides an abstraction of that subsystem. Simulation units (SUs) provide such abstractions and are widely used in industry. A class of SUs are described by the Functional Mock-up Interface Standard [3] (FMI), which is used commercially and is supported by many tools [7]. An SU implements a well-defined interface and represents a subsystem by computing its behavioral trace using a dedicated solver.

Co-simulation [19,11] addresses the need to simulate a CPS given as the composition of such black-box SUs. Co-simulation transforms a continuous system to a discrete simulation with discrete interactions between the different SUs. Furthermore, a digital twin can be a co-simulation connected to a physical systems.

The objective of a co-simulation is to capture as accurately as possible the behavior of the modeled system. This is challenging due to discretization, cyclic dependencies between the SUs, and the fact that very few assumptions be made about the SUs: an SU may, e.g., be unable to predict its future state at the next desired point in time. A co-simulation algorithm is responsible for orchestrating the interaction of the SUs: it determines how and when the different SUs interact.

Since the co-simulation algorithm should make the virtual system correspond to its physical counterpart, the virtual system can be analyzed, and different
design choices can be explored, to predict the behavior of the system to be built. However, the FMI standard is only informally described, and has been shown to be inconsistent [5]. For both of the above reasons, there is a need for formal methods to provide a formal semantics for co-simulation and to provide early model-based formal analysis of the co-simulations.

However, providing a formal semantics to co-simulation is challenging, due to, e.g., the complex behavior of the SUs, and the need to resolve cyclic dependencies between the SUs by fixed-point computations and to perform step negotiation to ensure that all SUs move in lockstep. Rewriting logic [21], with its modeling language and high-performance analysis tool Maude [6], should be a suitable formal method for co-simulation: Its expressiveness allows us to conveniently specify both complex dynamic behaviors and sophisticated functions (e.g., for detecting and resolving cyclic dependencies), and Maude provides automatic formal analysis capabilities for correctness analysis and design space exploration. Maude also supports connections to external objects, which means that Maude should be able to orchestrate the composition of real external components.

In this paper we present a formal framework for representing co-simulation in Maude. We give a formal model for co-simulation beyond the FMI 2.0 standard, also covering feed-through constraints, input instrumentations, and step rejection. We then use Maude to synthesize and symbolically execute suitable scenario-specific co-simulation algorithms, which enables the formal analysis of the resulting co-simulation. We also show how Maude can be used to synthesize instrumentations, parameter values, and co-simulation algorithms for such complex scenarios so that the resulting system satisfies desired properties. As discussed in Section 6, to the best of our knowledge this paper presents the first formal framework that covers design space exploration of complex co-simulation scenarios with algebraic loops and step rejection, and that also synthesizes correct-by-construction co-simulation algorithms and parameters for such scenarios.

Our framework currently does not connect to real-world SUs/FMUs; the interfaces of the SUs are abstractly represented in Maude. Nevertheless, as mentioned above, since Maude supports external objects, we believe that our framework can be naturally extended to perform co-simulation with real-world FMUs.

The rest of the paper is structured as follows. Section 2 provides necessary background to Maude and co-simulation. Section 3 presents a Maude model of co-simulation scenarios and SU behaviors. Section 4 shows how correct-by-construction co-simulation algorithms can be synthesized and executed in Maude. Section 5 describes how to synthesize instrumentation and parameter values such that the resulting co-simulation satisfies desired properties. Section 6 discusses related work and Section 7 gives some concluding remarks.

2 Preliminaries

2.1 Rewriting Logic and Maude

Maude [6] is a rewriting-logic-based executable formal specification language and high-performance analysis tool for object-based distributed systems.
A Maude module specifies a rewrite theory \((\Sigma, E \cup A, R)\), where:

- \(\Sigma\) is an algebraic signature; i.e., a set of sorts, subsorts, and function symbols.
- \((\Sigma, E \cup A)\) is a membership equational logic theory, with \(E\) a set of possibly conditional equations and membership axioms, and \(A\) a set of equational axioms such as associativity, commutativity, and identity, so that equational deduction is performed modulo the axioms \(A\). The theory \((\Sigma, E \cup A)\) specifies the system’s states as members of an algebraic data type.
- \(R\) is a collection of labeled conditional rewrite rules \([l] : t \rightarrow t'\) if \(\text{cond}\), specifying the system’s local transitions.

A function \(f\) is declared \(\text{op} f : s_1 \ldots s_n \rightarrow s\). Equations and rewrite rules are introduced with, respectively, keywords \(\text{eq}\) or \(\text{ceq}\) for conditional equations, and \(rl\) and \(crl\). A conditional rewrite rule has the form \(crl [l] : t \rightarrow t'\) if \(c_1 \land \ldots \land c_n\), where the conditions \(c_1, \ldots, c_n\) are evaluated from left to right. A condition \(c_i\) can be a Boolean term, an equation, a membership, or a matching equation \(u(x_1, \ldots, x_n) \rightarrow u'\) with variables \(x_1, \ldots, x_n\) not appearing in \(t\) and not instantiated in \(c_1, \ldots, c_{i-1}\); these variables become instantiated by \(\text{matching}\) \(u(x_1, \ldots, x_n)\) to the normal form of the (appropriate instance of) \(u'\). \(c_i\) can also be a rewrite condition \(u_i \rightarrow u'_i\), which holds if \(u'_i\) can be reached in zero or more rewrite steps from \(u_i\). Mathematical variables are declared with the keywords \(\text{var}\) and \(\text{vars}\), or can have the form \(\text{var:sort}\) and be introduced on the fly.

A class declaration \(\text{class} C | \text{att}_1 : s_1, \ldots, \text{att}_n : s_n\) declares a class \(C\) of objects with attributes \(\text{att}_1\) to \(\text{att}_n\) of sorts \(s_1\) to \(s_n\). An object instance of class \(C\) is represented as a term \(<O : C | \text{att}_1 : \text{val}_1, \ldots, \text{att}_n : \text{val}_n\>\), where \(O\), of sort \(\text{Oid}\), is the object’s identifier, and where \(\text{val}_1\) to \(\text{val}_n\) are the current values of the attributes \(\text{att}_1\) to \(\text{att}_n\). A system state is modeled as a term of the sort \(\text{Configuration}\), and has the structure of a multiset made up of objects and messages (and connections in our case).

The dynamic behavior of a system is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule (with label 1)

\[
rl [1] : \begin{cases} <0 : C | a1 : f(x, y), a2 : 0', a3 : z > \\
\Rightarrow <0 : C | a1 : x + z, a2 : 0', a3 : z > 
\end{cases}
\]

defines a family of transitions in which the attribute \(a_1\) of object 0 is updated to \(x + z\). Attributes whose values do not change and do not affect the next state, such as \(a_2\) and the right-hand side occurrence of \(a_3\), need not be mentioned.

\textit{Formal Analysis in Maude.} Maude provides a number of analysis methods, including rewriting for simulation purposes, reachability analysis, and linear temporal logic (LTL) model checking. The rewrite command \texttt{rew init} simulates one behavior from the initial state/term \texttt{init} by applying rewrite rules. Given a state pattern \texttt{pattern} and an (optional) condition \texttt{cond}, Maude’s \texttt{search} command searches the reachable state space from \texttt{init} for all (or optionally a given number of) states that match \texttt{pattern} such that \texttt{cond} holds:

\texttt{search init \rightarrow! pattern [such that cond] .}
The arrow $\Rightarrow!$ means that Maude only searches for final states (i.e., states that cannot be further rewritten) that match pattern and satisfies cond. If the arrow is $\Rightarrow*$ then Maude searches for all reachable states satisfying the search condition.

2.2 Co-simulation

Complex CPSs are composed of multiple communicating subsystems. For example, an autonomous car includes suspension, braking and collision avoidance subsystems. **Co-simulation** [12,19] is a technique enabling the discrete simulation of a continuous CPS, using multiple **simulation units** (SUs). Each such SU represents a subsystem and interacts with its environment through its ports.

**Co-simulation Scenarios.** A set of SUs can be composed into a **scenario** by **coupling** the input ports to output ports. A coupling connects an output port of an SU to an input port of another SU. The **coupling restriction** states that the value of an input and an output of a coupling must be the same at all times in the continuous system. However, in the discrete co-simulation, the coupling restrictions can only be satisfied at specific points in time called **communication points**. Therefore, each SU makes its own assumptions about the evolution of its input values between the communication points, which can introduce errors in the co-simulation [2]. An assumption about the evolution of an input can roughly be divided into two categories [14]:

- Interpolation (or **reactive**): the SU uses the current value at time $t$ and the **future** value at time $t + \Delta$ to predict the values in the interval $(t, t + \Delta)$.
- Extrapolation (or **delayed**): the SU uses the current value at time $t$ and the **previous** value at time $t - \Delta$ to predict the values in the interval $(t, t + \Delta)$.

The orchestrator computes the behavior of a scenario as a discrete trace, while it tries to satisfy the coupling restrictions, by exchanging values between the coupled ports. The orchestrator aims to find the communication points that minimize the co-simulation error while ensuring that the SUs move in lockstep by adapting to the behavior of the scenario. This is tricky, since the orchestrator needs to regard the SUs as nondeterministic blackboxes about which only few assumptions can be made. The optimal communication points furthermore depend on the approximation schemes used by the different SUs [14,22,15,24,13]. Unfortunately, most SUs will silently accept any given communication points, resulting in hard-to-debug erroneous results.

An example of the kind of nondeterministic behavior that the orchestrator needs to account for is **step rejection**, where an SU rejects a future state computation, since it implements error estimation and concludes that the desired step size may lead to an intolerable error. The FMI standard allows step rejections; however, they are generally unpredictable from the orchestrator’s perspective. An SU implementing error estimation has a maximal step size $h$, defining the interval for which it can reliably compute its future state.
The orchestrator addresses step rejections using step negotiation [17]. A scenario can also contain algebraic loops (cyclic dependencies) between the SUs, which are resolved using fixed-point computations [19,22,17]. Scenarios with algebraic loops and step rejections are called complex scenarios and are notoriously hard to simulate, since the orchestrator must adapt to the behavior of the non-deterministic SUs to ensure an accurate simulation using “angelic nondeterminism.”

The following definition of an SU is based on [4,16,17]:

Definition 1 (Simulation Unit). A simulation unit (SU) is a tuple

\[ SU \triangleq (S, U, Y, \text{set}, \text{get}, \text{step}) , \]

where:
- \( S \) is a set, denoting the state space of the SU.
- \( U \) and \( Y \) are sets, of input and output ports, respectively. The union \( \text{VAR} = U \cup Y \) of the inputs and outputs is called the ports of the SU.
- \( V \) is a set, denoting the values that a variable can hold. \( \mathcal{V}_T = \mathbb{R}_{\geq 0} \times V \) is the set of timestamped values exchanged between input and output ports.
- The functions \( \text{set} : S \times U \times \mathcal{V}_T \to S \) and \( \text{get} : S \times Y \to \mathcal{V}_T \) set an input and get an output, respectively.
- \( \text{step} : S \times \mathbb{R}_{>0} \to S \times \mathbb{R}_{>0} \) is a function; \( \text{step}(s, h) = (s', h) \) gives the state \( s' \) after time \( h \), where \( h \) is either \( H \) or the maximal time that the SU can progress from state \( s \).

Definition 2 (Scenario). A scenario \( S \) is a tuple

\[ S \triangleq (C, \{SU_c\}_{c \in C}, L, M, R, F) \]

- \( C \) is a finite set (of SU identifiers).
- \( \{SU_c\}_{c \in C} \) is a set of SUs, where each \( SU_c = (S_c, U_c, Y_c, \text{set}_c, \text{get}_c, \text{step}_c) \).
- \( L \) is a function \( L : U \to Y \), where \( U = \bigcup_{c \in C} U_c \) and \( Y = \bigcup_{c \in C} Y_c \), and where \( L(u) = y \) means that the output \( y \) is connected to the input \( u \).
- \( M \subseteq C \) denotes the SUs that may reject a future state computation.
- \( R : U \to \mathbb{B} \) is a predicate, which provides information about the SUs’ input approximation functions.
- \( F \) is a family of functions \( \{F_c : Y_c \to \mathcal{P}(U_c)\}_{c \in C} \). \( u_c \in F_c(y_c) \) means that the input \( u_c \) feeds through to the output \( y_c \) of the same SU.

The function \( R \) represents the instrumentation of the scenario. An input port \( u \) is reactive if \( R(u) \), and is delayed otherwise. Changing the instrumentation of a scenario changes the algorithm used to simulate the scenario. We assume that the instrumentation of a scenario is constant throughout the simulation, which is the case for most commercially used SUs [13]. Our definition extends the FMI 2.0 standard [3] with feed-through and port instrumentation. Figure 1 shows a way to graphically present co-simulation scenarios.
Co-simulation Algorithms. An orchestrator simulates a scenario by executing a co-simulation algorithm. A co-simulation algorithm consists of an initialization procedure and a co-simulation step [3]. This work focuses on the co-simulation step, which we refer to as “the algorithm” in the paper.

The state of a co-simulation scenario is defined as the combination of the states of its subcomponents:

**Definition 3 (Abstract SU State).** The observable abstract state $s^R_c$ of an SU SU$c$ in a scenario $S$ is an element of the set $S^R_c = \mathbb{R}_{\geq 0} \times S^R_{U_c} \times S^R_{Y_c} \times S^R_{V_c}$, where:
- $S^R_{U_c} : U_c \rightarrow \mathbb{R}_{\geq 0}$ is a function mapping each input port to a timestamp.
- $S^R_{Y_c} : Y_c \rightarrow \mathbb{R}_{\geq 0}$ is a function mapping each output port to a timestamp.
- $S^R_{V_c} : \text{VAR}_c \rightarrow \mathcal{V}$ is a function mapping each port to a value.

The first component of the abstract state denotes the time of the SU.

We use the abstract state $s^R_c$ of an SU $c$ instead of the internal state $s_c$ since the orchestrator cannot observe the latter.

**Definition 4 (Abstract Co-simulation State).** The abstract co-simulation state $s^R_S$ of a scenario $S = \langle C, \{SU_c\}_{c \in C}, L, M, R, F \rangle$ is an element of the set $S^R_S = \text{time} \times S^R_{U_S} \times S^R_{Y_S} \times S^R_{V_S}$ where:
- $\text{time} : C \rightarrow \mathbb{R}_{\geq 0}$ is a function, where $\text{time}(c)$ denotes the current simulation time of SU$c$. We denote by a time value $t \in \mathbb{R}_{\geq 0}$ the function $\lambda c.t$, which we use if all SUs are at the same time.
- $S^R_{U_S} = \prod_{c \in C} S^R_{U_c}$ maps all inputs of the scenario to a timestamp.
- $S^R_{Y_S} = \prod_{c \in C} S^R_{Y_c}$ maps all outputs of the scenario to a timestamp.
- $S^R_{V_S} = \prod_{c \in C} S^R_{V_c}$ maps all ports of the scenario to a value.

A co-simulation step $P$ is a sequence of operations that takes a co-simulation from one consistent state to another consistent state. We write $s \xrightarrow{P} s'$ if executing the co-simulation step $P$ from the state $s$ results in the state $s'$.

**Definition 5 (Co-simulation Step).** A co-simulation step $P$ is a sequence of SU actions that takes a consistent co-simulation state to another consistent co-simulation state. The state of the co-simulation is consistent if all input ports
have a source, and all coupled ports have the same value. Formally:

\[ \langle t, s^R_U, s^R_Y, s^R_V \rangle \xrightarrow{P} \langle t', s'^R_U, s'^R_Y, s'^R_V \rangle \]

\[ \implies (\text{consistent}(\langle t, s^R_U, s^R_Y, s^R_V \rangle)) \implies (\text{consistent}(\langle t', s'^R_U, s'^R_Y, s'^R_V \rangle) \land t' > t) \]

where consistent is defined as:

\[ \text{consistent}(\langle t, s^R_U, s^R_Y, s^R_V \rangle) \triangleq (\forall u_c \in U \exists y_d \in Y \cdot L(u_c) = y_d) \]

\[ \land (\forall u_c, y_d : L(u_c) = y_d \implies s^R_V(u_c) = s^R_V(y_d)) \]

Informally, the co-simulation step advances the scenario from an initial state at time \( t \) to a final state at time \( t + H \), where \( H > 0 \), and ensures that the coupling restrictions are satisfied at both the initial and the final state.

Figure 2 shows three different co-simulation steps of the scenario in Fig. 1 that are allowed by the FMI standard 2.0 [3].

![Algorithm 1](image1.png)

![Algorithm 2](image2.png)

![Algorithm 3](image3.png)

Fig. 2: Three co-simulation algorithms of the scenario in Fig. 1 conforming to the FMI Standard (version 2.0).

Although the three algorithms satisfy Definition 5 and consist of the same actions, they are not equivalent, and simulating with one algorithm instead of one of the others could change the co-simulation result as shown in [16,18]. To differentiate between them, we need to consider the semantics of the different SU actions described in Definition 1.

The semantics described in Definitions 6 to 8 is based on [12,18] and operates on abstract states. It describes which assumptions the orchestrator can place on the behavior of SUs and restricts how actions can be composed to construct a co-simulation step.

**Definition 6 (Get Action).** A value can be obtained from an output port \( y \) of an SU at time \( t \) using the action \( \text{get}^i(y) \). The action changes the state of the SU according to:

\[ s^R \xrightarrow{\text{get}^i(y)} (v, s'^R) \implies \text{preGet}(y, s^R) \land \text{postGet}(y, s^R, s'^R, v) \]

Where:

\[ \text{preGet}(y, \langle t, s^R_U, s^R_Y, s^R_V \rangle) \triangleq s^R_Y(y) < t \land \forall u \in F(y) \cdot s^R_U(u) = t \]
The precondition above states that no value must have been obtained from the output y since the SU was stepped \((s_0^R(y) < t)\). Furthermore, it requires that all the inputs that feed through to y have been updated, so they are at time t. The following postcondition ensures that the output is advanced to time t:

\[
\text{postSet}(y, \langle t, s_{U}^R, s_{Y}^R, s_{V}^R \rangle, \langle t, s_{U}^R, s_{Y}^R, s_{V}^R \rangle, v) \triangleq s_{Y}^R(y) = t \\
\land \forall y_m \in (Y \setminus y) \cdot s_{Y}^R(y_m) = s_{Y}^R(y_m)
\]

**Definition 7 (Set Action).** Setting a value \(\langle t_u, x \rangle\) on the input port \(u\) of an SU using \(\text{set}(s^{(i)}, u, \langle t_u, x \rangle)\) updates the time and value of the input port \(u\) such that they match \(\langle t_u, x \rangle\):

\[
s_{U} \xrightarrow{\text{set}(s^{(i)}, u, \langle t_u, x \rangle)} s_{U}' \implies \text{preSet}(u, s_{U}^R) \land \text{postSet}(u, v, s_{U}^R, s_{U}'^R)
\]

Where:

\[
\text{preSet}(u, \langle t_u, x \rangle, \langle t, s_{U}^R, s_{Y}^R, s_{V}^R \rangle) \triangleq s_{U}^R(u) < t_u \\
\land (\langle R(u_c) \land s_{U}^R(u) = t \rangle \lor \langle \neg R(u_c) \land s_{U}^R(u) < t \rangle)
\]

The precondition says that the input must not have been assigned a new value since the SU was stepped \((s_0^R(u) < t_u)\). Furthermore, it requires that the value \(\langle t_u, x \rangle\) respects the instrumentation of the input. The following postcondition ensures that the value and time of the input are updated so that they match the value assigned on the input:

\[
\text{postSet}(u, \langle t_u, x \rangle, \langle t, s_{U}^R, s_{Y}^R, s_{V}^R \rangle, \langle t, s_{U}^R, s_{Y}^R, s_{V}^R \rangle) \triangleq t_u = s_{U}^R(u) \\
\land (\forall u_m \in (U \setminus u) \cdot s_{U}^R(u_m) = s_{U}^R(u)) \land s_{U}'^R(u) = x
\]

**Definition 8 (Step Computation).** Stepping an SU using \(\text{step}(s^{(i)}, H)\) advances the state of the SU by at most \(H\):

\[
s_{U} \xrightarrow{\text{step}(s^{(i)}, H)} s_{U}' \implies \text{preStep}(H, s_{U}^R) \land \text{postStep}(H, s_{U}^R, s_{U}'^R)
\]

Where:

\[
\text{preStep}(H, \langle t, s_{U}^R, s_{Y}^R, s_{V}^R \rangle) \triangleq \forall u \in U \cdot ((R(u) \land t_{SU} + H = s_{U}^R(u)) \\
\lor (\neg R(u) \land t_{SU} = s_{U}^R(u))
\]

The above precondition states that all the SU’s inputs have been updated according to their instrumentation. The following postcondition ensures that the time of the SU advances by at most \(H\).

\[
\text{postStep}(H, \langle t, s_{U}^R, s_{Y}^R, s_{V}^R \rangle, \langle t', s_{U}^R, s_{Y}^R, s_{V}^R \rangle) \triangleq t + h' = t' \land h' \leq H
\]
An algorithm \( P \) must satisfy Definition 5 while respecting the defined semantics. This means that Algorithm 3 is correct, while Algorithms 1 and 2 are incorrect since they do not respect the semantics. In particular, Algorithms 1 and 2 try to perform a \( \text{step}_A \) action without respecting the reactive input \( u_g \); the state of SU \( A \) \( s^A = \langle 0, \{ u_g \to 0 \}, \{ y_f \to 0 \}, \_ \rangle \) does not contain \( \{ u_g \to H \} \). Intuitively, we try to step SU \( A \) without having provided it with a value on the reactive input \( u_g \); this violates \( \text{preStep}_A \).

**Problem Statement.** The two key problems in co-simulation that we address in this paper (in addition to the formalization of a co-simulation) are:

1. Given a scenario \( S \): Synthesize a co-simulation algorithm \( P \) for \( S \). That is, find a sequence of SU actions \( P \) which defines a valid co-simulation algorithm for \( S \). This involves solving possible algebraic loops and performing step negotiation to ensure that all SUs move in lockstep.

2. Given a parametric and partially instrumented scenario \( S \), where some SU parameters are unknown and where the instrumentation is incomplete, i.e., not all input ports are reactive or delayed: Find concrete values for the parameters, and concrete instrumentation of the input ports, such that the resulting instrumented scenario has desired properties.

### 3 Modeling Co-simulation Scenarios in Maude

This section describes how we model individual SUs and their composition in a co-simulation scenario in Maude. Due to space limitations, we only provide fragments of our Maude model. The entire model, including the synthesis and execution of co-simulation algorithms (Section 4) and the synthesis of instrumentations and parameters (Section 5) is available at https://github.com/SimplisticCode/Co-simulation_WRLA and consists of around 1400 LOC.

We formalize co-simulation scenarios in an object-oriented style. The state is a term \( \{ \text{SUs connections orchObjects} \} \) of sort \( \text{GlobalState} \), where \( \text{SUs} \) is set of objects modeling simulation units, \( \text{connections} \) denote the port couplings, and \( \text{orchObjects} \) are two additional objects used during synthesis and execution of co-simulation algorithms (see Section 4).

A simulation unit is modeled as an object instance of the following class:

```plaintext
class SU | time : Nat, inputs : Configuration, outputs : Configuration, canReject : Bool, fmistate : fmiState, parameters : LocalState, localState : LocalState .
```

The attribute \( \text{time} \) denotes the time of the SU; \( \text{inputs} \) and \( \text{outputs} \) denote the objects modeling the SU’s input and output ports; \( \text{canReject} \) is \( \text{true} \) if the SU implements error estimation (i.e., is an element of the set \( M \)); \( \text{fmistate} \) denotes the simulation mode (see [3]) of the SU; \( \text{localState} \) denotes the SU’s internal state; and \( \text{parameters} \) denotes the values of the SU’s parameters.

Input and output ports are modeled as instances of the following classes:
class Port | value : FMIValue, time : Nat, status : PortStatus, type : FMIType.
class Input | contract : Contract.
class Output | dependsOn : oidSet.
subclasses Input Output < Port.

value and time denote, respectively, the value of the port and the time of its last set/get operation; status is true if the port was updated at the current time; contract denotes the input port’s instrumentation (delayed or reactive); and dependsOn denotes the set of inputs that feed through to the output port.

Example 1. We illustrate our framework using a system where a controller controls the water level of a water tank with constant inflow of water, by opening and closing a valve in the tank. The system is modeled using one SU for the tank and one SU for the controller, and has the architecture in Fig. 1 without the feed-through. The water tank (in its initial state) is modeled as an object

< "tank" : SU | parameters : ("flow" |-> < 5 >), localState : ("waterlevel" |-> < 0 >),
  inputs : (< "valveState" State : Input | value : < 0 >, time : 0, contract : delayed >),
  outputs : (< "waterlevel" : Output | value : < 0 >, time : 0, status : Undef, dependsOn : empty >)
  time : 0, canReject : false >

The tank has one delayed input port and one output port, and the local state indicates that the tank is empty. The parameter flow denotes the amount of water that flows into the tank per time unit.

To formalize the behaviors of an SU we formalize the operations set, get, and step in Definition 1. For example, the get operation that updates the time and status of a set of output ports is formalized as follows:

op getAction : Object OidSet -> Object.
eq getAction(< SU1 : SU | >, empty) = < SU1 : SU | >.
eq getAction(< SU1 : SU | time : T,
  outputs : (< O : Output | time : T > OS) >, (O , P)) =
  getAction(< SU1 : SU | outputs :
    (< O : Output | time : T, status : Def > OS) >, P) .

The application-specific behavior of an SU is given by defining its step function:

Example 2. The following definition of the step function in our running example defines how the water level of the tank changes as a function of the step duration STEP, the parameter flow, and the state (value) of the input valve:

eq step(< "tank" : SU | time : T, parameters : ("flow" |-> < FLOW >),
  inputs : (< "valve" : Input | value : < STATE > >,
  outputs : (< "waterlevel" : Output | time : T >,
  localState : ("waterlevel" |-> < LEVEL >) >,

STEP) =
if STATE == 1 then --- valve is open
  < "tank" : SU | time : (T+STEP), localState : ("waterlevel" |-> < O >),
  outputs : (< "waterlevel" : Output | value : < 0 >, time : (T+STEP), status : Undef > >

3 We do not show variable declarations, but follow the convention that variables are written with capital letters.
else

--- valve is closed

< "tank" : SU | time : (T + STEP), localState : ("waterlevel" |-> < LEVEL + (STEP * FLOW)>),
outputs : < "waterlevel" : Output | value : < LEVEL + (STEP * FLOW) >,
        time : (T + STEP), status : Undef > > fi .

A connection/coupling connecting the output port o of SU su1 to the input port i of SU su2 is represented by the term su1 ! o == su2 ! i.

We define scenarios using constants simulationUnits and externalConnection to denote, resp., the simulation unit objects and their connections.

Example 3. The SUs and their couplings in our example are defined as follows:

eq simulationUnits = < "tank" : SU | parameters : ("flow" |-> <100>), localState : ("waterlevel" |-> <0>),
  time : 0, fmistate : Instantiated, canReject : false,
  inputs : (< "valveState" : Input | value : <0>, type : integer, time : 0,
            contract : delayed, status : Undef >),
  outputs : (< "waterlevel" : Output | value : <0>, type : integer, time : 0,
            status : Undef, dependsOn : empty >) >
< "ctrl" : SU | parameters : ("high" |-> <5>, "low" |-> <0>), canReject : false,
  localState : ("valve" |-> <false>), fmistate : Instantiated, time : 0,
  inputs : (< "waterlevel" : Input | value : <0>, type : integer, time : 0,
            contract : reactive, status : Undef >),
  outputs : (< "valveState" : Output | value : <0>, type : integer, time : 0,
            status : Undef, dependsOn : empty >) >

eq externalConnection = ("tank" ! "waterlevel" == "ctrl" ! "waterlevel")
("ctrl" ! "valveState" == "tank" ! "valveState") .

The constant setup defines the initial state, and adds appropriate initialized orchestration objects to the scenario:

op setup : -> GlobalState .
ceq setup = {INIT}
if SCENARIO := externalConnection simulationUnits
  /
validScenario(SCENARIO)
  /
LOOPS := tarjan(SCENARIO)
  /
NeSUIDs := getSUIDsOfScenario(SCENARIO)
  /
INIT := calculateSNSet(SCENARIO OData(1,LOOPS, NeSUIDs)) .

The function validScenario checks whether all inputs are coupled and that no input has two sources. The function tarjan returns (a possibly empty) set of algebraic loops in the scenario by searching for non-trivially strongly connected components in the graph constructed using the rules in [16]. The function getSUIDsOfScenario returns the set of all SU identifiers. Finally, calculateSNSet checks if step negotiation should be applied in the simulation of the scenario, and generates a global initial state with orchestration objects that store information about the discovered algebraic loops and whether step negotiation is needed.

4 Synthesizing and Executing Co-simulation Algorithms

This section describes how co-simulation algorithms for a given scenario can be synthesized and then executed in Maude.
4.1 Orchestration Data

The orchestration executes a given co-simulation algorithm on a scenario, and requires keeping track of the co-simulation algorithm and the execution state.

The following class SimData stores such data about the simulation:

```plaintext
class SimData | SNSet : OidSet, defaultStepSize : NzNat,
                  actualStepSize : NzNat, unsolvedSCC : AlgebraicLoopSet,
                  solvedSCC : AlgebraicLoopSet, guessOn : PortSet,
                  values : PortValueMap, simulationTime : Nat,
                  suids : NeOidSet .
```

The attribute SNSet denotes the set $M$ of SUs that may reject to step the desired step size (see Definition 2); defaultStepSize is the default step duration of the simulation, and the attribute actualStepSize is the negotiated step duration. The attributes actualStepSize and defaultStepSize are equal if $M = \emptyset$. The attributes unsolvedSCC and solvedSCC respectively denote the solved and unsolved algebraic loops. The attribute guessOn denotes the set of ports which are used to solve algebraic loops using the technique described in [17]; values is a map linking an input port to a value. The orchestration uses values to track which values it has obtained but not set on an input port. The attribute simulationTime describes the current time of the simulation, and suids denotes the identifiers of the SUs.

The following class AlgoData stores the co-simulation algorithm:

```plaintext
class AlgoData | CosimStep : ActionList, Initialization : ActionList,
                Termination : ActionList, endTime : NzNat .
```

The attributes Initialization and Termination denote the initialization procedure and termination procedure, respectively. The attribute CosimStep denotes the co-simulation step procedure that the orchestration applies until it reaches the end time of the simulation (given by endTime). All elements of the algorithm are of the sort ActionList, which is a list of SU operations (where we do not show actions for handling complex scenarios):

```plaintext
ops Set Get Step Save : -> ActionType [ctor] .
op portEvent:_SU:_PId:_ : ActionType SUID OidSet -> Action [ctor] .
subsort Action < ActionList .
```

4.2 Synthesis of Co-simulation Algorithms

We synthesize co-simulation algorithms for a scenario $\mathcal{S}$ by first performing and recording all possible SU actions, and then searching for consistent reachable final states. Any sequence of SU actions leading to such a state is a co-simulation algorithm.

A number of rewrite rules model the different SU actions. For example, the following rewrite rule describes a get operation:
A value $V$ is obtained from the output $O$ of $SU_1$ if the state satisfies all feed-through constraints $FT$ of the output $O$ (checked by `feedthroughSatisfied`). The rule updates the output $O$ using the operation `getAction`, inserts the output's value and time $(T; V)$ into `values`, and adds the performed action `portEvent: Get SU: SU1 PId: O` to its list `CosimStep` of performed actions.

All such “synthesis” rules in our model follow the same pattern: they rewrite the scenario while remembering how they did it.

We synthesize a co-simulation algorithm by starting with a consistent initial state and exploring how a consistent final state can be established. An algorithm for a given scenario is therefore synthesized using the following rewrite rule:

```maude
crl [get-syn] :
< SU1 : SU | fmistate: Simulation, inputs : IS,
  outputs : (< O : Output | time : T, status : Undef,
      value : V, dependsOn : FT > OS) >
  (SU1 ? O ==> SU2 ! I)
< OCH : SimData | values : PV >
< ALG : AlgoData | CosimStep : ALGO >
=>
getAction(< SU1 : SU | >, SU1 ? O)
  (SU1 ? O ==> SU2 ! I)
< OCH : SimData | values : insert((SU2 ! I), < T; V >, PV) >
< ALG : AlgoData | CosimStep : (ALGO ; EVENT) >
if feedthroughSatisfied(FT, IS, T)
  /\ EVENT := portEvent: Get SU: SU1 PId: O .
```

This rule checks whether the scenario `INIT` is a suitable initial state using the predicate `isNewState`. Then we construct an initial simulation configuration `CNF` as in Section 3. The key condition that does most of the work is the rewrite condition `{CONF} => {FINALSTATE}`, which searches for states reachable from `CONF` until it finds a state `FINALSTATE` that satisfies the property `allSUsUnloaded`, which ensures that all SUs have been properly simulated and unloaded. The function `getOrchestrator` extracts the synthesized co-simulation algorithm from this final state.

The following Maude command then synthesizes all valid co-simulation algorithms for a given `scenario`:

```maude
Maude> search scenario => FINALSTATE:GlobalState .
```
Many SU actions can happen independently at the same time, which means that multiple valid algorithms often can be synthesized for a scenario. For example, there are six different co-simulation algorithms for our water tank scenario.

4.3 Executing Co-Simulation Algorithms

This section describes how co-simulation algorithms can be executed. The state of such an execution is a term \( \text{run: algorithm on: scenario with: simData:} \).

\[
\text{op run: on: with: : Object Configuration Object \to SimState [ctor].}
\]

A co-simulation algorithm is executed by sequentially performing its actions, starting with performing all actions in the \text{Initialization}, then performing all actions of the \text{CosimStep}, and finally executing all actions in \text{Termination}.

The following rule shows the execution of the first action (\text{Get}) in \text{CosimStep}:

\[
\text{crl [get-exec]}:
\text{run: } \langle \text{ALG : AlgoData | CosimStep : } \langle \text{action: Get SU: SU1 PId: O} \rangle; \text{ALGO } \rangle
\text{on: CONF}
\langle \text{SU1 : SU | inputs : IS, outputs : } \langle \text{D : Output | time : T, value : V, dependsOn : FT } \rangle \text{O} \rangle
\text{with: } \langle \text{OCH : SimData | values : PV } \rangle
\to
\text{run: } \langle \text{ALG : AlgoData | CosimStep : ALGO } \rangle
\text{on: CONF getAction(\langle SU1 : SU \rangle, SU1 ! O \to SU2 ! INPUT)}
\text{with: } \langle \text{OCH : SimData | values : insert(SU2 ! INPUT) } \rangle
\text{getfeedthroughSatisfied(FT, IS, T).}
\]

We can combine algorithm synthesis and execution into the following rewrite rule, so that rewriting the term \text{runAnyAlgorithm scenario} synthesizes and executes a co-simulation algorithm for the co-simulation scenario \text{scenario}:

\[
\text{crl [runAlg]}: \text{runAnyAlgorithm INIT } \Rightarrow \text{run: ORC on: INIT with: SIMDATA}
\text{if LOOPS := tarjan(INIT)}
\text{\forall SUIDsNE := getSUIDsOfScenario(INIT)}
\text{\forall SIMDATA := initialOrchestrationData(1,LOOPS,SUIDsNE)}
\text{\forall ALGO := initialAlgorithmData(1)}
\text{\forall CONF := calculateSNSet(INIT ALGO) SIMDATA}
\text{\forall (CONF) } \Rightarrow \{\text{FINALSTATE}\}
\text{ORC := getOrchestrator(FINALSTATE)}
\text{allSUsinUnloaded(SUIDsNE, FINALSTATE).}
\]

This rule is similar to the rule \text{getAlgorithm}, and also extracts the resulting algorithm \text{ORC} and simulation data \text{SIMDATA}.

\text{Example 4. The water tank scenario described in Example 3 (waterTankScenario below) can be simulated by rewriting:}

\[
\text{Maude} > \text{frew (runAnyAlgorithm waterTankScenario) .}
\]

The command returns the final simulation state
4.4 Checking Confluence of Synthesized Co-simulation Algorithms

Section 4.2 shows that multiple valid co-simulation algorithms can be synthesized for a given scenario. Executing all these valid co-simulation algorithms for a given scenario should give the same result, since all the SUs are deterministic in the sense that if we try to step an SU $A$ from some initial state $s$ with step size $h$, it will always produce the same final state $A'$. All this indicates that some actions are independent of each other. Therefore, their relative execution order is irrelevant, and an optimized algorithm can merge such independent actions and perform them in parallel.

The following Maude command checks whether all generated co-simulation algorithms for a scenario $scenario$ result in the same final state:

Maude> search (runAnyAlgorithm $scenario$) =>! S:SimState .

This search command synthesizes and then executes all co-simulation algorithms for the scenario $scenario$. For our water tank scenario, the search produces a single result, which means that all synthesized algorithms give the same result.

5 Synthesizing Instrumentations and SU Parameters

Our framework makes possible different kinds of design space exploration to allow the practitioner to see how different design choices affect the behavior of the system. This section shows how our framework can be used to synthesize parameter values and instrumentations of the inputs that lead to desired simulations.

5.1 Instrumentation of a Scenario

Finding a good instrumentation of the input ports (i.e. deciding whether an input port should be reactive or delayed) is important not only to achieve accurate...
co-simulation results [14,22,18], but also because some instrumentations of a scenario may lead to algebraic loops while others do not.

We use reachability analysis to explore the consequences of different instrumentations of a scenario to find the instrumentation that yields the desired simulation results. To explore different instrumentations of a scenario, we create a partially instrumented scenario, where the some of the input ports have the contract noContract instead of reactive or delayed.

Example 5. In the following partially instrumented water tank scenario, the input port "valveState" of the SU "tank" and the "waterLevel" input port of the SU "ctrl" are not yet instrumented:

```plaintext
eq waterTankNotInstrumented =
  < "tank" : SU | parameters : ("flow" |-> <100>), localState : ("waterLevel" |-> <0>),
  time : 0, fsmState : Instantiated, canReject : false,
  inputs : (< "valveState" : Input | value : <0>, type : integer, time : 0,
               contract : noContract, status : Undef>),
  outputs : (< "waterLevel" : Output | value : <0>, type : integer, time : 0,
              status : Undef, dependsOn : empty>),

< "ctrl" : SU | parameters : ("high" |-> <5>), ("low" |-> <0>), canReject : false,
  localState : ("valve" |-> <false>), fsmState : Instantiated, time : 0,
  inputs : (< "waterLevel" : Input | value : <0>, type : integer, time : 0,
             contract : noContract, status : Undef>),
  outputs : (< "valveState" : Output | value : <0>, type : integer, time : 0,
             status : Undef, dependsOn : empty>), > .
```

We use an operator findInstr to instrument such partially instrumented scenarios, so that the state findInstr(scenario) becomes a fully instrumented scenario when all ports have been instrumented (rule remove-findInstr). The rules instr-delayed and instr-reactive set uninstrumented input ports to be either delayed or reactive:

```plaintext
rl [instr-delayed]:
  findInstr(< SU1 : SU | inputs : < I : Input | contract : noContract > IS > C>)

rl [instr-reactive]:
  findInstr(< SU1 : SU | inputs : < I : Input | contract : noContract > IS > C>)

crl [remove-findInstr]: findInstr(CONF) => CONF if instrumented(CONF).
```

The different instrumentations of a partially instrumented scenario are found and explored using the following rule:

```plaintext
crl [findInstrumentation]: findContracts(INIT) => CONF
  if findInstr(INIT) => CONF
  /
  \ empty == tarjan(CONF) \- no algebraic loops
  \ runAnyAlgorithm CONF => run: ORC on: FINAL with: SIMDATA
  \ simulationFinished(ORC)
  \ desiredProperty(FINAL) .
```

76
This rule generates an instrumented scenario \( CONF \) from the partially instrumented scenario \( INIT \). \( CONF \) is then simulated (in the rewrite condition), leading to a final state \( FINAL \). The instrumentation can be restricted by giving properties that the instrumented scenario \( CONF \) and/or the simulation result \( FINAL \) must satisfy. For example, the condition \( \text{empty} == \text{tarjan}(CONF) \) says that the instrumentation should not lead to algebraic loops, and the last conjunct in the condition says that the simulation result \( FINAL \) must satisfy \( \text{desiredProperty} \).

**Example 6.** We define \( \text{desiredProperty} \) to be that the water level of the tank is in a desired range. The following Maude command then finds all instrumentations which lead to simulations which end in a desired water level:

Maude> search findContracts(waterTankNotInstrumented) =>! C:Configuration .

This command returns the three instrumentations (with parts replaced by ‘...’)

Solution 1
\[
\begin{align*}
\text{C:Configuration} & \rightarrow \cdots \\
\langle \text{"ctrl"} & : \text{SU} \mid \text{inputs} : \langle \text{"waterlevel"} : \text{Input} \mid \text{contract} : \text{delayed} \rangle \rangle \\
\langle \text{"tank"} & : \text{SU} \mid \text{inputs} : \langle \text{"valveState"} : \text{Input} \mid \text{contract} : \text{reactive} \rangle \rangle \\
\end{align*}
\]

Solution 2
\[
\begin{align*}
\text{C:Configuration} & \rightarrow \cdots \\
\langle \text{"ctrl"} & : \text{SU} \mid \text{inputs} : \langle \text{"waterlevel"} : \text{Input} \mid \text{contract} : \text{reactive} \rangle \rangle \\
\langle \text{"tank"} & : \text{SU} \mid \text{inputs} : \langle \text{"valveState"} : \text{Input} \mid \text{contract} : \text{delayed} \rangle \rangle \\
\end{align*}
\]

Solution 3
\[
\begin{align*}
\text{C:Configuration} & \rightarrow \cdots \\
\langle \text{"ctrl"} & : \text{SU} \mid \text{inputs} : \langle \text{"waterlevel"} : \text{Input} \mid \text{contract} : \text{delayed} \rangle \rangle \\
\langle \text{"tank"} & : \text{SU} \mid \text{inputs} : \langle \text{"valveState"} : \text{Input} \mid \text{contract} : \text{delayed} \rangle \rangle \\
\end{align*}
\]

### 5.2 Synthesizing SU Parameters

An SU may have different parameters. In our framework, the user can specify a finite set of possible values for a parameter using a \textit{choose} operator, and we can then synthesize those parameter values that result in desired simulations.

**Example 7.** We want to synthesize the value of the parameter \( \text{flow} \) of the water tank such that the water level is above 10 in the final simulation state. The following predicate defines the desired water level:

\[
\begin{align*}
\text{op above10 : Configuration} & \rightarrow \text{Bool} \\
\text{eq above10}(\text{CONF} < \text{"tank"} : \text{SU} \mid \text{localState} : \text{"waterlevel"} \mid \rightarrow \langle V \rangle \rangle = V > 10 .
\end{align*}
\]

To synthesize a \( \text{flow} \) value from the set \( \{1, 2, 30\} \) we initialize \( \text{flow} \) accordingly:

\[
\begin{align*}
\langle \text{"tank"} & : \text{SU} \mid \text{parameters} : \langle \text{"flow"} \mid \rightarrow \text{choose}(1, 2, 30) \rangle \rangle \cdots
\end{align*}
\]

We use the following rule to synthesize parameter values that result in simulations that satisfy \( \text{above10} \):
The following Maude command gives all initialized scenarios which lead to desired simulations:

Maude> search selectParams(parametricWaterTank) => C:Configuration .

Solution 1
C:Configuration --> ... < "tank" : SU | parameters : "flow" |-> <30>, ... >

No more solutions.

We can also simultaneously synthesize both desired instrumentations and parameter values by having noContract ports and choose(...) values.

6 Related Work

A number of papers, e.g. [16,12,4,17], synthesize co-simulation algorithms for fixed scenarios. In contrast to our paper, this body of work does not provide formal models of co-simulation, and therefore no formal analysis. We exploit Maude’s formal analysis features to synthesize suitable instrumentations and SU parameters, which is not addressed by the mentioned related work.

Design space exploration of SU parameters is described in [8,9]. This work uses genetic algorithms to find optimal parameters values. However, it does not consider how different instrumentations can affect the simulation result.

Another example of DSE of a CPS using Maude can be found in [20], where Maude is used to validate and analyze drone/unmanned aerial vehicle flight strategies to find the optimal flight strategy using an external simulation engine. In contrast, we use Maude’s capabilities to validate co-simulation algorithms and formalize the co-simulation semantics instead of evaluating flight strategies.

Formal methods have been used for co-simulation, e.g., [25,1,5,26,18]. Thule et al. [25] formalize a given scenario and two given co-simulation algorithms for that scenario in PROMELA and use the SPIN model checker to compare the two simulation algorithms, e.g., in terms of reachability. In contrast, we provide a general formal framework for co-simulation, synthesize co-simulation algorithms for a given scenario, synthesize instrumentations and parameter values, and capture a broader class of co-simulation scenarios (e.g., including scenarios with algebraic loops and step rejection) than those in the case study in [25].

Cavalcanti et al. [5] provide the first behavioral semantics of FMI. They show how to prove essential properties of co-simulation algorithms using CSP, and also show that the co-simulation algorithm provided in the FMI standard is...
not consistent. We cover an extension of FMI scenarios, and also include feed-
through, step rejection, and input port instrumentation. Furthermore, as already
mentioned, we also synthesize co-simulation algorithms and parameters.

Amálio et al. [1] show how formal tools can detect algebraic loops in a sce-
nario. We not only detect such loops, but also solve them to synthesize co-
simulation algorithms. Zeyda et al. [26] formalize a co-simulation scenario in
Isabelle/UTP, and prove different properties—including behavioral properties—
about the scenario. In contrast, we use automatic model checking methods to
both synthesize and analyze co-simulation algorithms, and also cover complex
scenarios (algebraic loops, step rejection, etc.) not covered in [26].

On the Maude side, Mason et al. [20] use Maude and statistical model check-
ing to analyze a system of UAVs (“drones”). The key point is that they integrate
a quite realistic “external” UAV simulator, Ardupilot/SITL, into their Maude
simulations. Maude and the simulator communicate by message passing. This
work does not formalize co-simulation in our FMI sense, but shows that Maude
can execute together with, and coordinate, external simulators for CPSs.

7 Concluding Remarks

We have presented a formal model of co-simulation in Maude for complex scenar-
ios with algebraic loops and step negotiation. Using rewrite conditions, we have
used Maude to generate and execute co-simulation algorithms, and to synthesize
port instrumentations and parameter values (albeit from a finite set of possible
values), such that the resulting co-simulation satisfies desired properties.

In future work we should validate our framework on larger applications. We
should also explore how Maude’s symbolic analysis methods can be used to
synthesize algorithms and parameter values from symbolic initial states which
represent infinitely many concrete states. Although users can define complex
behaviors of their SUs, connecting Maude to a solver for real numbers such as
dReal [10] could support defining the continuous dynamics of SUs using differ-
ential equations. Finally, we should exploit Maude’s support for external objects
to execute the synthesized algorithms on real systems.

Acknowledgments: We thank Claudio Gomes, Jaco van de Pol, José Meseguer,
and Stefan Hallerstede for valuable discussions and feedback. We also thank the
anonymous reviewers for useful comments on a previous version of this paper.

References

vol. 10009. Springer (2016)

2. Arnold, M., Clauß, C., Schierz, T.: Error analysis and error estimates for co-
simulation in FMI for model exchange and co-simulation v2.0. In: Progress in


An Efficient Canonical Narrowing Implementation for Protocol Analysis

Raúl López-Rueda¹, Santiago Escobar¹, and José Meseguer²

¹ VRAIN, Universitat Politècnica de València, Valencia, Spain. {rloprue,sescobar}@upv.es
² University of Illinois at Urbana-Champaign, Urbana, IL, USA. meseguer@illinois.edu

Abstract. This work improves the canonical narrowing previously implemented using Maude 2.7.1 by taking advantage of the new functionalities that Maude 3.2 offers. In order to perform more faithful comparisons between algorithms, we have reimplemented Maude’s built-in narrowing using Maude’s metalevel. We compare these two metalevel implementations with Maude’s built-in narrowing, implemented at the C++ level, through a function that collects all the solutions, since the original command only returns one at a time. The results of these experiments are relevant for narrowing-based protocol analysis tools, as well as for improving the analysis of many other narrowing-based applications such as logical model checking, theorem proving or partial evaluation.

Keywords: Canonical narrowing · Reachability analysis · Maude · Narrowing modulo · Security protocols.

1 Introduction

Since verification of protocol security properties modulo the algebraic properties of a protocol’s cryptographic functions for an arbitrary number of sessions is generally undecidable, and the state space is infinite, symbolic techniques such as unification and narrowing modulo a protocol’s algebraic properties, as well as SMT solving, are particularly well suited to support symbolic model checking and theorem proving verification methods.

The Maude-NPA [7] is a symbolic model checker for cryptographic protocol analysis based on the above-mentioned symbolic techniques, which are efficiently supported by the underlying Maude language [4]. These Maude-based symbolic techniques are also used by other protocol analysis tools such as Tamarin [12] and AKISS [2].

This work has been partially supported by the EC H2020-EU grant agreement No. 952215 (TAILOR), by the grant RTI2018-094403-B-C32 funded by MCIN/AEI/10.13039/501100011033 and ERDF “A way of making Europe”, by the grant PROMETEO/2019/098 funded by Generalitat Valenciana, and by the grant PCI2020-120708-2 funded by MICIN/AEI/10.13039/501100011033 and by the European Union NextGenerationEU/PRTR.
State explosion is a significant challenge in this kind of symbolic model checking analysis modulo algebraic properties, particularly because unification modulo algebraic properties can generate large numbers of unifiers when computing symbolic transitions. Although Maude-NPA has quite effective state space reduction techniques [6], further state space reduction gains can be obtained by more sophisticated equational narrowing techniques such as canonical narrowing [8], whose state space reduction advantages were experimentally validated using Maude 2.7. The main motivation for the present work comes from the fact that the new unification and narrowing features supported by the current Maude 3.2, as well as its meta-level features, make possible a new implementation of canonical narrowing that we show can achieve additional computational and performance improvements. Throughout this work, we consider several experimental examples in order to demonstrate the effectiveness of the new implementation in Maude 3.2. Below we briefly explain these examples together with their Maude specification.

The first defined module, Example 5 below, is a classic in the Maude system. It is the coffee and apple vending machine, in which dollars and quarters are inserted to buy combinations of those products. The second defined module, Example 6 below, goes one step further at the level of complexity. In this case we implement a Maude specification of a very simple protocol using exclusive-or. Likewise, the third module, Example 7 below, is a very simple module with just one transition rule where symbolic reasoning takes place modulo the theory of an abelian group.

A fourth example explores the advantages of canonical narrowing modulo an equational theory that includes the idempotence property. The reason why we have chosen this property is because it is highly problematic in automated reasoning (even for matching and rewriting). It makes easier the representation of sets, in contrast to multisets, and it is useful when dealing with processes or agents. If we have several processes working at the same time, and it turns out that two of them are the same, the idempotence property allows us to eliminate one of them to avoid redundancy and reduce the use of computational resources.

Example 1. We can modify the equational theory of the vending machine to add some equations that express idempotence:

```
mod IDEMPOTENCE-VENDING-MACHINE is
  sorts Coin Item Marking Money State . subsort Coin < Money . subsort Money Item < Marking .
  op empty : -> Money .
  op <_> : Marking -> State . ops $ q : -> Coin . ops c a : -> Item .
  var M : Marking .
  eq [idem-item-a] : a a M = a M [variant] .
  eq [change] : q q q q M = $ M [variant] .
ends
```

Note that idempotence is not specified for quarters (q), but only for dollars ($), apples (a) and cups of coffee (c). This is because there is already an equation that
reduces the repetition of four quarters to a dollar, so that adding idempotence for quarters would create a conflict.

If we consider an initial term $< M_1 >$ that only contains a variable of type $\text{Money}$, we would obtain several traces by using the narrowing algorithm. In each one of the observed narrowing states, it is necessary to unify with the left-hand side of the rules to determine the new narrowing steps that can be taken. Each of those possible steps results in a new branch in the reachability tree. One of these traces takes us to the term $< \, \$ \, a \, c \, q \, q \, M_4 \, >$, which also contains a variable $M_4$ of type $\text{Money}$. The narrowing sequence associated to this term is as follows:

$$< M_1 > \sim_{\sigma_1} < \, \$ \, a \, q \, M_2 \, > \sim_{\sigma_2} < \, a \, c \, q \, M_3 \, > \sim_{\sigma_3} < \, \$ \, a \, c \, q \, q \, M_4 \, >$$

where $M_2$ and $M_3$ are also variables of type $\text{Money}$ and the computed substitutions are $\sigma_1 = \{ M \mapsto \$ M_2 \}$, $\sigma_2 = \{ M_2 \mapsto \$ M_3 \}$, and $\sigma_3 = \{ M_3 \mapsto \$ M_4 \}$. Note that in the first narrowing step, the substitution applied to the left-hand side of the rule $\text{buy-a}$ is $\rho_1 = \{ W_1 \mapsto \$ M_2 \}$ for $W_1$ the variable of a renamed version of rule $\text{buy-a}$. For the second narrowing step, the substitution applied to the left-hand side of the rule $\text{buy-c}$ is $\rho_2 = \{ W_2 \mapsto a \, q \, M_3 \}$ for $W_2$ the variable of a renamed version of rule $\text{buy-c}$. For the third narrowing step, the substitution applied to the left-hand side of the rule $\text{buy-a}$ is $\rho_3 = \{ W_3 \mapsto \$ a \, c \, q \, M_4 \}$ for $W_3$ the variable of a renamed version of rule $\text{buy-a}$. Note that extra $\$ are introduced by $\rho_1$ and $\rho_3$ due to equational unification using the variant equations and the axioms.

As we will see later, the use of canonical narrowing will allow us to introduce irreducibility constraints in the algorithm, which in many cases will significantly reduce the number of branches in the narrowing reachability tree.

The remaining of this paper is organized as follows. Section 2 provides some preliminaries on rewriting logic and narrowing. Section 3 gives a detailed presentation of canonical narrowing. Section 4 describes our new implementation of canonical narrowing in Maude 3.2. Section 5 presents the experiments carried out using (i) the standard built-in narrowing, (ii) our implementation of standard narrowing, and (iii) our implementation of canonical narrowing. Finally, Section 6 summarizes the paper and presents some future work. All of the Maude modules and experiments are available at https://github.com/ralorueda/canonical-narrowing.

2 Preliminaries

We follow the classical notation and terminology from [17] for term rewriting, and from [13] for rewriting logic and order-sorted notions.

We assume an order-sorted signature $\Sigma$ with a poset of sorts $(S, \leq)$. The poset $(S, \leq)$ of sorts for $\Sigma$ is partitioned into equivalence classes, called connected components, by the equivalence relation $(\leq \cup \geq)^+$. We assume that each connected component $[s]$ has a top element under $\leq$, denoted $\top [s]$ and called the top sort of $[s]$. This involves no real loss of generality, since if $[s]$ lacks a top sort, it can be easily added.
We assume an $S$-sorted family $X = \{X_s\}_{s \in S}$ of disjoint variable sets with each $X_s$ countably infinite. $T_{\Sigma}(X)$ is the set of terms of sort $s$, and $T_{\Sigma,s}$ is the set of ground terms of sort $s$. We write $T_{\Sigma}(X)$ and $T_{\Sigma}$ for the corresponding order-sorted term algebras. Given a term $t$, $\text{Var}(t)$ denotes the set of variables in $t$.

A substitution $\sigma \in \text{Subst}(\Sigma, X)$ is a sorted mapping from a finite subset of $X$ to $T_{\Sigma}(X)$. Substitutions are written as $\sigma = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ where the domain of $\sigma$ is $\text{Dom}(\sigma) = \{X_1, \ldots, X_n\}$ and the set of variables introduced by terms $t_1, \ldots, t_n$ is written $\text{Ran}(\sigma)$. The identity substitution is $id$. Substitutions are homomorphically extended to $T_{\Sigma}(X)$. The application of substitution $\sigma$ to a term $t$ is denoted by $t\sigma$ or $\sigma(t)$.

A $\Sigma$-equation is an unoriented pair $t = t'$, where $t, t' \in T_{\Sigma}(X)_s$ for some sort $s \in S$. Given $\Sigma$ and a set $E$ of $\Sigma$-equations, order-sorted equational logic induces a congruence relation $=_E$ on terms $t, t' \in T_{\Sigma}(X)$ (see [14]). Throughout this paper we assume that $T_{\Sigma,s} \neq \emptyset$ for every sort $s$, because this affords a simpler deduction system. We write $T_{\Sigma/E}(X)$ and $T_{\Sigma/E}$ for the corresponding order-sorted term algebras modulo the congruence closure $=_E$, denoting the equivalence class of a term $t \in T_{\Sigma}(X)$ as $[t]_E \in T_{\Sigma/E}(X)$.

An equational theory $(\Sigma, E)$ is a pair with $\Sigma$ an order-sorted signature and $E$ a set of $\Sigma$-equations. An equational theory $(\Sigma, E)$ is regular if for each $t = t'$ in $E$, we have $\text{Var}(t) = \text{Var}(t')$. An equational theory $(\Sigma, E)$ is linear if for each $t = t'$ in $E$, each variable occurs only once in $t$ and in $t'$. An equational theory $(\Sigma, E)$ is sort-preserving if for each $t = t'$ in $E$, each sort $s$, and each substitution $\sigma$, we have $\tau \sigma \in T_{\Sigma}(X)_s$ iff $\tau' \sigma \in T_{\Sigma}(X)_s$. An equational theory $(\Sigma, E)$ is defined using top sorts if for each equation $t = t'$ in $E$, all variables in $\text{Var}(t)$ and $\text{Var}(t')$ have a top sort.

An $E$-unifier for a $\Sigma$-equation $t = t'$ is a substitution $\sigma$ such that $\sigma =_E t' \sigma$. For $\text{Var}(t) \cup \text{Var}(t') \subseteq W$, a set of substitutions $\text{CSU}_W^E(t = t')$ is said to be a complete set of unifiers for the equality $t = t'$ modulo $E$ away from $W$ iff: (i) each $\sigma \in \text{CSU}_W^E(t = t')$ is an $E$-unifier of $t = t'$; (ii) for any $E$-unifier $\rho$ of $t = t'$ there is a $\sigma \in \text{CSU}_W^E(t = t')$ such that $\sigma|_W =_E \rho|_W$ (i.e., there is a substitution $\eta$ such that $\sigma|_W =_E \rho|_W$; and (iii) for all $\sigma \in \text{CSU}_W^E(t = t')$, $\text{Dom}(\sigma) \subseteq (\text{Var}(t) \cup \text{Var}(t'))$ and $\text{Ran}(\sigma) \cap W = \emptyset$.

A rewrite rule is an oriented pair $l \rightarrow r$, where $l \notin X$ and $l, r \in T_{\Sigma}(X)_s$ for some sort $s \in S$. An (unconditional) order-sorted rewrite theory is a triple $(\Sigma, E, R)$ with $\Sigma$ an order-sorted signature, $E$ a set of $\Sigma$-equations, and $R$ a set of rewrite rules. The set $R$ of rules is sort-decreasing if for each $t \rightarrow t'$ in $R$, each $s \in S$, and each substitution $\sigma$, $t\sigma \in T_{\Sigma}(X)_s$ implies $\sigma \in T_{\Sigma}(X)_s$.

The rewriting relation on $T_{\Sigma}(X)$, written $t \rightarrow_R t'$ or $t \rightarrow_{p, R} t'$ holds between $t$ and $t'$ iff there exist $p \in \text{Pos}_{\Sigma}(t)$, $l \rightarrow r \in R$ and a substitution $\sigma$, such that $tl = lr$, and $t' = t[\sigma]_p$. The relation $\rightarrow_{R/E}$ on $T_{\Sigma}(X)$ is $=_E \rightarrow_{R/A} =_E$. The transitive (resp. transitive and reflexive) closure of $\rightarrow_{R/E}$ is denoted $\rightarrow^*_{R/E}$ (resp. $\rightarrow^*_{R/E}$). A term $t$ is called $\rightarrow_{R/E}$-irreducible (or just $R/E$-irreducible) if there is no term $t'$ such that $t \rightarrow_{R/E} t'$. For $\rightarrow_{R/E}$ confluent and terminating, the irreducible version of a term $t$ is denoted by $t^*_{R/E}$. 

85
A relation \( \rightarrow_{R,E} \) on \( T\Sigma(X) \) is defined as: \( t \rightarrow_{p,R,E} t' \) (or just \( t \rightarrow_{R,E} t' \)) iff there are a non-variable position \( p \in \text{Pos}_\Sigma(t) \), a rule \( l \rightarrow r \) in \( R \), and a substitution \( \sigma \) such that \( t|_p = \sigma \lor \sigma \) and \( t' = t[r\sigma]_p \). Reducibility of \( \rightarrow_{R,E} \) is undecidable in general since \( E \)-congruence classes can be arbitrarily large. Therefore, \( R/E \)-rewriting is usually implemented \([11]\) by \( R,E \)-rewriting under some conditions on \( R \) and \( E \) such as confluence, termination, and coherence.

We call \((\Sigma, B, E)\) a decomposition of an order-sorted equational theory \((\Sigma, E\cup B)\) if \( B \) is regular, linear, sort-preserving, defined using top sorts, and has a finitary and complete unification algorithm, which implies that \( B \)-matching is decidable, and the equations \( E \) oriented into rewrite rules \( E \) are convergent, i.e., confluent, terminating, and strictly coherent \([15]\) modulo \( B \), and sort-decreasing.

Given a decomposition \((\Sigma, B, E)\) of an equational theory, \((t', \theta)\) is an \( E, B \)-variant \([3,10]\) (or just a variant) of \( t \) if \( \theta t_{E,B} \approx_E t' \) and \( \theta t'_{E,B} \approx_E \theta \). A complete set of \( E, B \)-variants \([10]\) (up to renaming) of a term \( t \) is a subset, denoted by \([t]_{E,B} \), of the set of all \( E, B \)-variants of \( t \) such that, for each \( E, B \)-variant \((t', \theta)\) of \( t \), there is an \( E, B \)-variant \((t'', \theta)\) of \( t \) such that \((t'', \theta) \equiv_{E,B} (t', \theta) \).

A decomposition \((\Sigma, B, E)\) has the finite variant property (FVP) \([10]\) (also called a finite variant decomposition) iff for each \( \Sigma \)-term \( t \), a complete set \([t]_{E,B} \) of its most general variants is finite.

In what follows, the set \( G \) of equations will in practice be \( G = E \cup B \) and will have a decomposition \((\Sigma, B, E)\).

**Definition 1 (Reachability goal).** \([16]\) Given an order-sorted rewrite theory \((\Sigma, G, R)\), a reachability goal is defined as a pair \( t \rightarrow^*_{R/G} t' \), where \( t, t' \in T\Sigma(X) \).

It is abbreviated as \( t \rightarrow^* \) when the theory is clear from the context; \( t \) is the source of the goal and \( t' \) is the target. A substitution \( \sigma \) is a \( R/G \)-solution of the reachability goal (or just a solution for short) iff there is a sequence \( \sigma(t) \rightarrow_{R/G} \sigma(u_1) \rightarrow_{R/G} \cdots \rightarrow_{R/G} \sigma(u_{k-1}) \rightarrow_{R/G} \sigma(t') \).

A set \( \Gamma \) of substitutions is said to be a complete set of solutions of \( t \rightarrow^*_{R/G} t' \) iff (i) every substitution \( \sigma \in \Gamma \) is a solution of \( t \rightarrow^*_{R/G} t' \), and (ii) for any solution \( \rho \) of \( t \rightarrow^*_{R/G} t' \), there is a substitution \( \sigma \in \Gamma \) more general than \( \rho \) modulo \( G \), i.e., \( \sigma|_{\text{Var}(t)\cup \text{Var}(t')} \equiv_G \rho|_{\text{Var}(t)\cup \text{Var}(t')} \).

This provides a tool-independent semantic framework for symbolic reachability analysis of protocols under algebraic properties. Note that we have removed the condition \( \text{Var}(r) \subseteq \text{Var}(l) \) for rewrite rules \( l \rightarrow r \in R \) and thus a solution of a reachability goal must be applied to all terms in the rewrite sequence. If the terms \( t \) and \( t' \) in a goal \( t \rightarrow^*_T G t' \) are ground and rules have no extra variables in their right-hand sides, then goal solving becomes a standard rewriting reachability problem. However, since we allow terms \( t, t' \) with variables, we need a mechanism more general than standard rewriting to find solutions of reachability goals. Narrowing with \( R \) modulo \( G \) generalizes rewriting by performing unification at non-variable positions instead of the usual matching modulo \( G \).
Specifically, narrowing instantiates the variables in a term by a G-unifier that enables a rewrite modulo G with a given rule of R and a term position.

**Definition 2 (Narrowing modulo G)**. [16] Given an order-sorted rewrite theory \((\Sigma, G, R)\), the narrowing relation on \(T_\Sigma(X)\) modulo G is defined as \(t \leadsto_{\sigma, R, G} t'\) (or \(\sigma\) if \(R, G\) is understood) iff there is \(p \in \text{Pos}_\Sigma(t)\), a rule \(l \rightarrow r \in R\) such that \(\text{Var}(t) \cap (\text{Var}(l) \cup \text{Var}(r)) = \emptyset\), and \(\sigma \in CSU_G^V(t_p = l)\) for a set \(V\) of variables containing \(\text{Var}(t)\), \(\text{Var}(l)\), and \(\text{Var}(r)\), such that \(t' = \sigma(t[r])\).

The reflexive and transitive closure of narrowing is defined as \(t \leadsto^*_{\sigma, R, G} t'\) iff either \(t = t'\) and \(\sigma = \text{id}\), or there are terms \(u_1, \ldots, u_n\), \(n \geq 1\), and substitutions \(\sigma_1, \ldots, \sigma_{n+1}\) s.t. \(t \leadsto_{\sigma_1, R, G} u_1 \leadsto_{\sigma_2, R, G} u_2 \cdots u_n \leadsto_{\sigma_{n+1}, R, G} t'\) and \(\sigma = \sigma_1 \cdots \sigma_{n+1}\).

Soundness and completeness of narrowing with rules \(R\) modulo the equational theory \(G\) for solving reachability goals are proved in [11,16] for order-sorted topmost rewrite theories, i.e., rewrite theories were all the rewrite steps happen at the top of the term.

### 3 Canonical narrowing

This section gives an overview of the canonical narrowing strategy of [8]. The canonical narrowing relation \(\leadsto_{R/E,B}\) includes irreducibility constraints only for the left-hand sides of the rules.

**Definition 3 (Canonical Constrained Narrowing)**. [8] Given an order-sorted rewrite theory \((\Sigma, E \cup B, R)\) such that \((\Sigma, B, E)\) is a decomposition of \((\Sigma, E \cup B)\), the canonical narrowing relation with irreducibility constraints holds between \(\langle t, \Pi \rangle\) and \(\langle t', \Pi' \rangle\), denoted

\[
\langle t, \Pi \rangle \leadsto_{\sigma, R/E,B} \langle t', \Pi' \rangle
\]

iff there exists \(l \rightarrow r \in R\), which we always assume renamed, so that \(\text{Var}(\langle t, \Pi \rangle) \cap (\text{Var}(r) \cup \text{Var}(l)) = \emptyset\), and a unifier \(\alpha \in CSU_{E,B}^V(t = l)\), where \(W = \text{Var}(\langle t, \Pi \rangle) \cup \text{Var}(r) \cup \text{Var}(l)\), and

1. \(\langle t', \Pi' \rangle = \langle r\alpha, \Pi \alpha \cup \{(l\alpha)\}_{E,B} \rangle\), and
2. \(\Pi \alpha \cup \{(l\alpha)\}_{E,B}\) are \(\overrightarrow{E}, B\)-irreducible.

Soundness and completeness of canonical narrowing with rules \(R\) modulo the equational theory \(E \cup B\) w.r.t. canonical rewriting for solving reachability goals are proved in [8].

Note that we do not require a narrowing step to compute \(CSU_{E,B}(t = l)\) anymore, we perform regular equational unification but impose an irreducibility constraint on the normal form of the instantiated left-hand side, which can be handled in Maude by using asymmetric unification [5].

The irreducibility constraints are computed by using the normalized left-hand side of the rules that are used in the narrowing steps. Each trace will carry
a different set of irreducibility constraints, although several of the conditions are shared by having common predecessor nodes. In each new narrowing step, the list of irreducibility constraints computed previously in that sequence must be taken into account, so that if it is necessary to reduce one of the terms appearing in the list to compute a new step, it will be discarded. In this way, we eliminate redundancy as well as branches of the reachability tree, which will be less and less wide than the tree resulting from using standard narrowing. In some cases, we will even get infinite reachability trees to become finite, ensuring termination.

**Example 2.** If we look at the module of Example 1, we can define an equational unification problem of the form \( t = t' \). Specifically, if we consider the narrowing trace shown in that example, we can place ourselves in the third term, just before taking the last step. To compute the next possible steps from that term, it is necessary to try to unify it with the left-hand side of each of the defined rules. In this case, we will focus on the rule \texttt{buy-a}, which is also used to take the first step of the trace. The specification of the unification problem would then be \( t = \langle a \ c \ q \ M3 \rangle \) and \( t' = \langle W3 \ downarrow \rangle \), where \( W3 \) is a variable of type \texttt{Marking} (money, items, or combinations of them) corresponding to the variable of a renamed version of rule \texttt{buy-a}. If we run the unification problem using Maude’s command, we will get 5 unifiers as a solution:

Maude> variant unify \langle a \ c \ q \ M3:Money \rangle =? \langle W3:Marking \ downarrow \rangle .

Note that \( \rho_3 \) of Example 1 corresponds to the third unifier. But of those 5 unifiers, there are 3 that could be ignored, since the accumulated substitution makes the left-hand side of the \texttt{buy-a} rule used at the first narrowing step reducible. Canonical narrowing would have computed irreducibility constraints that come from normalizing the instantiated left-hand side of the rules when taking the first and second step. That is, the terms \( \langle M3 \ downarrow \rangle \) (i.e., \( \langle W1 \ downarrow \rangle \maple{p_1}^{E,B} = \langle W1 \ downarrow \rangle \maple{p_2}^{E,B} = \langle W2 \ downarrow \rangle \maple{p_2}^{E,B} = \langle a \ q \ M3 \rangle \) are assumed to be irreducible when we want to take the last step of the trace. Maude’s unification command allows us to indicate this irreducibility constraint using such that \( M3:Money \ downarrow \) irreducible at the end command. If we run it now, we can see how the number of unifiers found is reduced to 2, since the first, third and fourth unifiers from the previous command are discarded:
As can be seen, the use of irreducibility constraints manages to reduce the number of unifiers. By applying them to the narrowing algorithm, as canonical narrowing does, then this implies the reduction of possible steps (branches in the reachability tree) from the term in which we were, since for each one of the unifiers found between the term and the right part of a rule, we will have a new narrowing step.

4 Implementation

Our approach has been to create a meta-level command in which one of the input parameters allows us to choose between the standard narrowing algorithm or the canonical narrowing algorithm.

4.1 Using the meta-level

To achieve the implementation of the command it is necessary to use some calls to the Maude meta-level available in Maude 3.2. Thanks to this, we can reuse functionalities that are integrated at the native level in C++, achieving much better performance than if we implemented them from scratch; as it happened in the previous implementation in Maude 2.7.

Each user command in Maude is represented by a corresponding command at the meta-level, allowing us greater control and management of their outputs. For example, the `variant unify` command that we saw in Example 2 corresponds to the `metaVariantUnify` command at the meta level. It is precisely this command that we use to carry out the unification step in our implementation, since it allows us to perform equational unification modulo variant equations and axioms. The operator that defines the command is the following:

```
op metaVariantUnify : Module UnificationProblem TermList Qid VariantOptionSet Nat ~> UnificationPair? .
```

The command receives six parameters and returns a structure of type `UnificationPair?`, an error or a pair consisting of a substitution and an identifier of the family of variables used. The first command received is the module that defines the rewriting theory to work on. The second is the unification problem to which solutions are sought. The third is a list of irreducibility terms, which is of vital importance in the canonical narrowing algorithm. The fourth corresponds to the identifier of the family of variables to avoid (the one used for the variables of the unification problem). The fifth is a parameter used to indicate if we want to filter the returned unifiers. Finally, a natural number parameter is received in which the unifier to be searched is indicated. We show an execution of this command in Example 3, using in turn the module of the vending machine with idempotence as a rewriting system (see Example 1).
Example 3. Considering the module from Example 1 again as a rewriting system, we can use the `metaVariantUnify` command to find the unifiers seen in Example 2. We simply indicate the same equational unification problem, and by means of the last argument of the command we can select each of the unifiers to obtain. Additionally, we can use an irreducibility condition to reduce the number of unifiers just like we have seen before. For example, by using the same irreducibility condition, we can obtain one of the unifiers as follows:

```
Maude> reduce in META-LEVEL :
> metaVariantUnify(upModule('IDEMPOTENCE-VENDING-MACHINE, true),
> '<_>'['$.Coin','M3:Money'], '@, none, 0) .
result UnificationPair: {
'M3:Money <- '__['q.Coin,'q.Coin','q.Coin','#1:Money'] ;
'W3:Marking <- '__['a.Item','c.Item','#1:Money'], '#})
```

Another meta-level functionality that has been necessary to use is the `metaNarrowingApply` command. It performs a narrowing step, using the arguments shown in its definition below. Thanks to this command and the `metaVariantUnify` one, we can abstract from the unification processes, which are the most costly at the computational level. By invoking meta-level commands to do so, execution is done natively in C++ code, which turns out to be much faster and more efficient. The operator that defines the command is the following:

```
op metaNarrowingApply :
Module Term TermList Qid VariantOptionSet Nat -> NarrowingApplyResult? .
```

In this case, the command receives as the first parameter, again, the module that represents the rewrite theory to be used. The second parameter represents the term from which to perform the narrowing step. The third parameter is a list of irreducibility terms, important for canonical narrowing. The fourth parameter is the identifier of the family of variables to avoid. The fifth parameter is used to indicate if we want to filter the returned unifiers in order to get only the most general unifiers. Finally, the sixth parameter is the step that you want to take, that is, the “branch” of the tree that you want to generate from the given term. The result will be of type `NarrowingApplyResult?`, a data structure that contains either an error, or the necessary information from the narrowing step performed.

Example 4. We use again the module from Example 1. As an initial term we consider the same that we will use later for the experiments whose results are shown in Table 4. The `metaNarrowingApply` command allows us to give (among others) the first step of narrowing from that term:

```
Maude> reduce in META-LEVEL :
> metaNarrowingApply(upModule('IDEMPOTENCE-VENDING-MACHINE, true),
> '<_>'['M1:Money'], empty, '@, none, 0) .
result NarrowingApplyResult: { '<_>'['M1:Money'], 'State, 
```

The output returned by Maude shows how the rule labeled as `buy-a` has been used to perform the narrowing step, resulting in two different assignments. On
the one hand, a dollar is assigned together with a fresh variable to the variable $M_1$ of type $\text{Money}$. On the other hand, the same fresh variable is assigned to the variable $M_2$ of type $\text{Marking}$ (Note that in this case, this is possible only because $\text{Money}$ is a subsort of $\text{Marking}$).

There is also a \texttt{metaNarrowingSearch} command that performs the entire instead of only one step, but we have not used it since we need to perform intermediate operations between each narrowing step to implement the canonical narrowing algorithm.

### 4.2 Data structures and the narrowing command

All narrowing algorithms perform one-step transitions from one symbolic state to another—the narrowing steps—using the rewrite rules of the given specification. We use a tree as a data structure, in which each of these narrowing steps gives rise to a new node, with its associated term. Thus, the root node of the tree will have as its associated term the initial term (reduced to normal form) indicated by the user. At the same time, each of the nodes is itself a data structure, in which we not only find the associated term, but also some extra information that allows us to locate the node and generate new terms from it.

Our implementation is built in such a way that ten parameters are requested from the user to invoke the command, as follows:

\[
\text{narrowing(Module, Term, SearchArrow, Term, Algorithm, VariantOptionSet, TermList, Qid, Bound, Bound)}
\]

The first argument receives the rewrite theory to perform the unification and narrowing steps. The second and fourth arguments are used to indicate the initial term and the target term respectively. The third argument corresponds to the search arrow that we want to use, so that solutions are included or discarded depending on the rewriting steps performed to achieve them. This argument may take values to indicate that only solutions that involve a single rewrite step, one or more steps, or any number of steps can be considered. The combination of the fifth and sixth parameters will indicate the type of algorithm to use. Combinations indicating the use of standard narrowing or canonical narrowing are currently accepted. The seventh argument is used to indicate a list of initial irreducibility terms to consider. This argument will be taken into account in all unification calls and in each narrowing step, allowing the value \texttt{empty} to indicate that we do not want to use irreducibility constraints. The eighth argument receives the identifier used to name the variables in the initial and target terms, to avoid later clashes. Finally, the ninth and tenth arguments are used to impose bounds on the algorithm, being able to indicate a maximum depth to expand the search tree or a maximum of solutions to search.

### 4.3 Search for Solutions

When we receive the parameters from the user, the first necessary step is to verify that the value of the depth limits and solutions are admissible. If they
are, the strategy to follow will be determined according to the indicated search
arrow.

Once all the above is prepared, the first nodes of the search tree are generated
from the root, that is, from the initial term. The tree will be generated by levels,
so that children of any node belonging to the next level will not be generated until
that level is completely generated. Each node contains its associated term plus
some extra information. Specifically, for each node we need a unique identifier,
a reference to its parent node, the branch of the tree to which it belongs, the
depth to which it is located and, in the case of the use of canonical narrowing,
a list of the irreducibility terms calculated so far in that branch.

Each time a new node is generated, an attempt is made to unify its associated
term with the target term indicated by the user. If unifiers exist, a solution will
be built for each of the unifiers found. To do this, it is necessary to go backwards
through the branch to which the node belongs, combining the substitutions made
to compute the accumulated substitution. If we are using canonical narrowing,
when a new node is generated it will also be necessary to modify the list of
irreducibility terms, adding the irreducibility term that is calculated from the
normalized left-hand side of the rule used to reach the node (see Definition 3).

4.4 Avoiding variable clashes

For the generation of new nodes, some calls are made to internal commands of
the Maude meta-level. These commands only allow the indication of a variable
identifier to avoid (which must be the one used previously), preventing possible
variable clashes. Each of the calls to these internal commands will result in a
random use of the rest of the variable identifiers handled by Maude. This gives
rise to the possibility that variables can be repeated in different nodes, which is
not an a priori problem, but it can’t be assumed when it is required to calculate
the cumulative substitution of a reachability solution.

To avoid this problem, we have chosen the strategy of renaming each of
the fresh variables that Maude generates on the fly, using a new identifier, the $ symbol. That is why in the final result returned to the user, all the fresh variables
that contain the narrowing solutions will be identified with that symbol, thus
ensuring that none of them clashes with the rest.

4.5 Algorithm performance improvement

Due to the nature of the algorithm and the uses for which it is intended, perfor-
mance of the algorithm plays a very important role. To improve this character-
istic, different aspects have been taken into account regarding the sequence in
which the algorithm acts and the data structures it handles.

Regarding the operators and equations in the code, they have been divided
into three main parts, which correspond to the main steps of the algorithm at a
theoretical level: (i) the generation of nodes (terms) in the reachability tree, (ii)
the attempt to unify each new term with the target term, and (iii) the computa-
tion of solutions in case the unification is successful. Likewise, each of these

92
parts is divided into subparts that facilitate not only the understanding of the code, but also a structured scheme to add new functionalities easily. Thanks to this, once we reimplemented the standard narrowing algorithm, it was relatively easy to add the new functionalities that modified it to achieve the canonical narrowing algorithm.

We can also consider the way in which the algorithm handles the data structures it works with. A priori, it could be thought that the nodes that are generated can go to a set of nodes that is subsequently processed. However, our strategy is to use an ordered list in which the nodes are processed taking into account an order similar to that of a recursion queue. In the same way, the nodes that are being processed in that list (that is, those in which the children have been generated) go to another list. This second list is used for the computation of the accumulated substitutions in the solutions. There is also another list in which the found unifiers are stored. It is also ordered to facilitate working with it recursively and calculating the solutions from the unifiers.

In addition to all this, extra parameters are dragged in the main data structure and also locally in each of the nodes. These parameters will later help to perform certain operations more quickly and efficiently. For example, each node has a reference to its parent node identifier, making it easy to go backwards on its branch if a cumulative substitution needs to be calculated.

5 Experiments

To test the operation and efficiency of the new command, as well as to check the performance differences between the different algorithms, we have used the modules mentioned in the introduction. That is, we used for the experiments the module of the vending machine (Example 5 below), the module of a protocol using the exclusive-or property (Example 6 below), the module of a process counter using the properties of an abelian group, the module of the vending machine with idempotence (Example 7 below), and Example 1. These modules allow us to check how the narrowing algorithms behave in those cases, subjecting the command to executions of different complexity for various applications.

The reimplementation of both the standard narrowing and canonical narrowing in the same command presented in Section 4 allows us to perform more faithful comparisons between the algorithms, independently of the standard built-in narrowing algorithm provided by Maude at the C++ level. However, since the built-in narrowing returns only one solution when executed via its meta-level function, we have also built a command that iteratively obtains all solutions. In this way, in the tables below we include (i) the standard built-in narrowing, (ii) our implementation of standard narrowing, and (iii) our implementation of canonical narrowing. It can be noted how, in certain cases, our canonical narrowing algorithm manages to surpass even the built-in narrowing command.
Table 1. Experiments using the vending machine module.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Depth limit</th>
<th>Execution time</th>
<th>Solutions found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>4</td>
<td>32 ms</td>
<td>163</td>
</tr>
<tr>
<td>Standard</td>
<td>4</td>
<td>75 ms</td>
<td>163</td>
</tr>
<tr>
<td>Canonical</td>
<td>4</td>
<td>60 ms</td>
<td>137</td>
</tr>
<tr>
<td>Native</td>
<td>5</td>
<td>112 ms</td>
<td>550</td>
</tr>
<tr>
<td>Standard</td>
<td>5</td>
<td>496 ms</td>
<td>550</td>
</tr>
<tr>
<td>Canonical</td>
<td>5</td>
<td>324 ms</td>
<td>119</td>
</tr>
<tr>
<td>Native</td>
<td>6</td>
<td>460 ms</td>
<td>1850</td>
</tr>
<tr>
<td>Standard</td>
<td>6</td>
<td>6384 ms</td>
<td>1850</td>
</tr>
<tr>
<td>Canonical</td>
<td>6</td>
<td>2724 ms</td>
<td>1213</td>
</tr>
<tr>
<td>Native</td>
<td>7</td>
<td>3092 ms</td>
<td>6216</td>
</tr>
<tr>
<td>Standard</td>
<td>7</td>
<td>160828 ms</td>
<td>6216</td>
</tr>
<tr>
<td>Canonical</td>
<td>7</td>
<td>45808 ms</td>
<td>3559</td>
</tr>
</tbody>
</table>

5.1 Experiments with the vending machine

Example 5. This Maude’s system module is a classic in the Maude community. It is the coffee and apple vending machine, in which dollars and quarters are inserted to buy combinations of those products. To do this, we specify that each coffee costs one dollar and each apple three-quarters of a dollar. Two rules handle state transitions for those specifications. Furthermore, an equation is used to specify the change of four-quarters of a dollar to one dollar. Note the addition of a variable $M$ of type Marking to make the rules and equations ACU-coherent.

We use the reachability problem $<M1> \Rightarrow_{a,R/E,B}^* St$ where $M1$ is a variable of type $Money$ and $St$ is a variable of type $State$. That is, we are asking for all the states that can be reached from an initial state containing only quarters and dollars. It is a fairly generic problem that allows us to see the number of nodes that are being generated in the reachability tree. Table 1 shows the results of running the command with this reachability problem.

These initial experiments use a simple rewrite theory and a simple reachability problem. As a consequence, the narrowing included natively in Maude turns out to be faster than either of our two algorithms, thanks to its coding in C++.
However, we can see that even in these cases, if we compare our standard narrowing implementation with our canonical narrowing implementation, the latter has always a better performance. This leads us to think that a natively programmed canonical narrowing would be able to outperform Maude’s standard narrowing even using these simple parameters. To strengthen this idea, we can look at the number of solutions (which in this case represent the number of states in the tree) found. For example, for depth level 7, canonical narrowing is capable of reducing the number of states generated by almost half regarding standard narrowing. If it was implemented natively in Maude, its execution time would obviously be much less, since it has to go through far fewer rewriting steps. In addition, the decrease in solutions represents in itself a relevant improvement, since those that come from redundancy in the rewriting traces are being discarded.

5.2 Experiments with a protocol using the exclusive-or property

Example 6. The equational theory used in the protocol below corresponds to the XOR property. Note the addition of the second equation for AC-coherence.

```plaintext
fmod EXCLUSIVE-OR is
  sort XOR .
  op mt : -> XOR .
  op _*_ : XOR XOR -> XOR [assoc comm] .
  vars X Y Z U V : [XOR] .
endfm
```

In the XOR-PROTOCOL module, the equation theory is imported and the rest of the protocol is implemented. The main structure is a state that stores the set of messages that have been sent and the new messages to be sent. The exchange of messages is done between two users for the protocol to take place. The - and + symbols are used as operators to distinguish between the messages to be received or sent respectively. The Nonces generation is included in the protocol, as well as data structures that specify the knowledge that an intruder might have.

```plaintext
mod XOR-PROTOCOL is protecting EXCLUSIVE-OR .
  sorts Name Nonce Fresh Msg .
  subsort Name Nonce XOR < Msg .
  subsort Nonce < XOR .
  ops a b c : -> Name .
  op n : Name Fresh -> Nonce .
  op pk : Name Msg -> Msg .
  ops r1 r2 r3 : -> Fresh .
  sort SMsg .
  sort SMsgList .
  subsort SMsg < SMsgList .
  ops + - : Msg -> SMsg .
  op nil : -> SMsgList .

  sort Strand .
  sort StrandSet .
  subsort Strand < StrandSet .
  op '['._''] : SMsgList SMsgList -> Strand .
  op st : -> StrandSet .
  op _& : StrandSet StrandSet -> StrandSet [assoc comm id: mt] .

  sort IntruderKnowledge .
  op at : -> IntruderKnowledge .
  op inI : Msg -> IntruderKnowledge .
  op nI : Msg -> IntruderKnowledge .
  op _',_ : IntruderKnowledge IntruderKnowledge -> IntruderKnowledge [assoc comm id: mt] .
```

95
Table 2. Experiments using the XOR-protocol module.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Execution time</th>
<th>Solutions found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>1660 ms</td>
<td>84</td>
</tr>
<tr>
<td>Standard</td>
<td>16124 ms</td>
<td>84</td>
</tr>
<tr>
<td>Canonical</td>
<td>2300 ms</td>
<td>1</td>
</tr>
</tbody>
</table>

We can define a reachability problem by using a basic message exchange between users. To do this, we consider a backwards execution, so that the target term will be the initial state of the message stack, while the initial term will be the final state. If a solution is found, it means that that execution trace exists, so it could occur in the protocol. The Maude-NPA [7] tool works in a similar way to this.

Considering X and Y as variables of type \texttt{Msg}, the reachability problem that we have defined for the experiments is the following:

\[\begin{align*}
&\{ \texttt{nil}, +(\texttt{pk}(a,n(b,r1)))\}, -(\texttt{pk}(b,Y)), +(Y \ast n(b,r1)) | \texttt{nil} \\
\land &\{ \texttt{nil}, -(\texttt{pk}(a,X)), +(\texttt{pk}(b,n(a,r2))), -(X \ast n(a,r2)) | \texttt{nil} \\
\quad &\{\texttt{inI}(X \ast n(a,r2)), \texttt{inI}(\texttt{pk}(a,X)), \texttt{inI}(\texttt{pk}(b,Y))\} \texttt{ narrowing}^* \\
\land &\{ \texttt{nil}, -(\texttt{pk}(a,X)), +(\texttt{pk}(b,n(a,r2))), -(X \ast n(a,r2)) | \texttt{nil} \\
\quad &\{\texttt{inI}(X \ast n(a,r2)), \texttt{inI}(\texttt{pk}(a,X)), \texttt{inI}(\texttt{pk}(b,Y))\} \end{align*}\]

In this context we are working with a finite search space, so we can ignore the limit of solutions and the depth limit (although all the solutions are found in depth 6, so we could also use that depth limit). Results are shown in Table 2.

In this case, the native standard narrowing in Maude again manages to be faster than our two algorithms, although the difference is less than before. The usual impact that canonical narrowing has on the number of returned solutions is further noticeable. As we mentioned before, thanks to carrying out a “pruning” of the tree by discarding those redundant traces, canonical narrowing is able to reduce the 84 initial solutions to only 1.

It is important to note here the usefulness of canonical narrowing in the field of security protocols, and specifically for tools relying on unification and/or narrowing, such as the Maude-NPA tool [7], Tamarin [12] and AKISS [2]. By
managing to rule out redundancies when generating the branches of the reachability tree, as seen in Example 2, when we analyze a protocol with canonical narrowing we achieve higher performance with less numerous reachable states but still complete results.

5.3 Experiments using the properties of an abelian group

We have already seen that canonical narrowing is useful even when —due to the prototype nature of its present implementation— it cannot be faster than the C++ based native standard narrowing in Maude. We have also concluded that if it were also integrated natively, it could be substantially faster and generate fewer states than standard narrowing in many cases. But if we also carry out experiments with systems in which there are many variants, these claims will be further reinforced.

We can see the real potential of canonical narrowing by resorting to a module using only one simple transition rule and the equations of an abelian group.

Example 7. We first implement a simple module defining the properties of an abelian group. That will be the equational theory used.

```plaintext
fmod ABELIAN-GROUP is
  sort Int .
  ops 0 1 : -> Int [ctor] .
  op _-_ : Int -> Int .
  vars X Y Z : Int .
  eq X + 0 = X [variant] .
  eq X + (- X) = 0 [variant] .
  eq X + (- X) + Y = Y [variant] .
  eq - (- X) = X [variant] .
  eq - 0 = 0 [variant] .
  eq (- X) + (- Y) = -(X + Y) [variant] .
  eq -(X + Y) + Y = -(X + Y) + Z [variant] .
  endfm
```

A rewrite theory which consists of a pair of integers that function as process counters is defined. The first integer represents the processes that are running, while the second represents those that have finished their execution. The transition rule represents the termination of a process that was in running, so that the value of the first integer of the pair is decreased by one, and at the same time the value of the second integer is increased by one. The transition rule allows for narrowing, while the abelian group equations allow for the generation of variants. Combining everything, we get a system of transitions that, despite looking simple, turns out to be quite complex, due to the large number of variants that any term will normally have.

```plaintext
mod PROC-COUNTER is protecting ABELIAN-GROUP .
  sort State .
  op <_,_> : Int Int -> State [ctor] .
  vars X Y Z : Int .
```

97
Table 3. Experiments using the process counter module.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Depth limit</th>
<th>Execution time</th>
<th>Solutions found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>1</td>
<td>424 ms</td>
<td>184</td>
</tr>
<tr>
<td>Standard</td>
<td>1</td>
<td>420 ms</td>
<td>184</td>
</tr>
<tr>
<td>Canonical</td>
<td>1</td>
<td>440 ms</td>
<td>184</td>
</tr>
<tr>
<td>Native</td>
<td>2</td>
<td>&gt; 8 h</td>
<td>−</td>
</tr>
<tr>
<td>Standard</td>
<td>2</td>
<td>&gt; 8 h</td>
<td>−</td>
</tr>
<tr>
<td>Canonical</td>
<td>2</td>
<td>2752 ms</td>
<td>719</td>
</tr>
<tr>
<td>Native</td>
<td>3</td>
<td>&gt; 8 h</td>
<td>−</td>
</tr>
<tr>
<td>Standard</td>
<td>3</td>
<td>&gt; 8 h</td>
<td>−</td>
</tr>
<tr>
<td>Canonical</td>
<td>3</td>
<td>12548 ms</td>
<td>2033</td>
</tr>
<tr>
<td>Native</td>
<td>4</td>
<td>&gt; 8 h</td>
<td>−</td>
</tr>
<tr>
<td>Standard</td>
<td>4</td>
<td>&gt; 8 h</td>
<td>−</td>
</tr>
<tr>
<td>Canonical</td>
<td>4</td>
<td>73070 ms</td>
<td>4969</td>
</tr>
</tbody>
</table>

\[
\text{rl [finish-proc]} : < (X + 1), Y > \Rightarrow < ((X + 1) + (-1)), (Y + 1) > \text{[narrowing] .}
\]

Considering \(X\) and \(Y\) are variables of type \(\text{Int}\), we use a common initial term: 
\(< 0, 1 + X >\). The target term will vary slightly allowing us to fix the depth at
which we want to find the solution. For depth one, it will be \(< -1, Y >\). For
depth two, it will be \(< -(1 + 1), Y >\). For depth three, it will be \(< -(1 + 1 + 1), Y >\), and for depth four, it will be \(< -(1 + 1 + 1 + 1), Y >\).

Apparently, the initial term and the target terms are very simple in this
example, but due to the large number of variants that an abelian group contains,
the computation becomes very complex, since the reachability tree will grow very
quickly in width. Table 3 shows the results of executing the above problems using
different depth limits.

In this case, we can see how the execution time for the first level (i.e., first
reachability problem) is very similar in any of the three algorithms. Furthermore,
it is striking that the solutions returned are the same. This is normal,
since canonical narrowing does not have any kind of impact on the first level,
because it has not yet calculated (see Definition 3) irreducibility constraints
(unless we specify them as part of the initial call). However, we can see how
from depth 2, our standard narrowing algorithm does not even manage to finish
in a reasonable execution time. The built-in narrowing doesn’t do it either. In
contrast, the canonical narrowing algorithm does terminate, returning a large
number of solutions in a relatively short time. The deeper we go into the tree,
the more solutions are found. At the same time, the execution time grows, but
it is still acceptable.

This is a clear example of the enormous improvement that canonical nar-
rowing can bring over standard narrowing in many cases, even using the one
found natively in Maude. Obviously, if we put canonical narrowing at the same
level, that is, included in Maude natively, the performance difference would be
extremely large in favor of canonical narrowing, especially in this type of cases.
5.4 Experiments with the vending machine using idempotence

As we mentioned earlier in the introduction, idempotence is a very important property in computing, since, for example, set data types enjoy it. Canonical narrowing seems to behave very well managing this property when compared to standard narrowing (even better than the experiments with an abelian group). We have done some experiments in which this property is used to corroborate this. We use the vending machine module with idempotence on items and dollars (see Example 1). The reachability problem defined in this case is \(< M_1 > \sim^*_{a,R/E,B} < $ a \ c \ M_2 >\), where \(M_1\) is a variable of type \(\text{Money}\) and \(M_2\) is a variable of type \(\text{Marking}\).

Table 4 shows the results obtained when using the reachability problem. We must bear in mind that in this case, once again, the growth of the tree in width is very large, due to the large number of variants of the system.

In this case we can see that the executions with a lower depth limit are extremely fast. However, even in those cases the difference is obvious, with the canonical narrowing being faster and returning fewer solutions. As we increase the depth limit, the difference in performance becomes more and more noticeable.

Just by looking at the experiments with depth limit 4, we can see that canonical narrowing achieves a performance at the computational level about 10 times better than Maude’s native standard narrowing. And if we instead look at our Maude reimplementation of standard narrowing, for a fair comparison, the difference is huge. More than half an hour of execution is reduced to just 11 seconds. The number of solutions returned is also reduced to one tenth, showing that the large percentage of those calculated by standard narrowing were unnecessary. For depth limit 5, the execution times of the standard narrowing are no longer reasonable, while the canonical narrowing manages to finish in just 5 minutes.

6 Conclusions

In this work, we have presented a new efficient implementation of canonical narrowing in Maude. The algorithm uses irreducibility constraints to reduce the
width of the reachability tree without losing completeness. The experiments that we have presented demonstrate the improvements that canonical narrowing offers over standard narrowing, both in terms of performance and solutions. Typically, the greater the number of variants calculated for each unification problem, the greater the improvement. Furthermore, the deeper the reachability tree to be generated, the greater the performance relationship between both algorithms. The results are important for tools such as Maude-NPA or others that are used to analyze protocols. They are also relevant in many other areas of computing, such as when performing symbolic model checking verification of concurrent systems, theorem proving or partial evaluation.

The most obvious next step is to include the command at the user level, making the outputs returned by it more readable and understandable. Another interesting step forward consists of integrating canonical narrowing with the computation of most general unifiers [1,9]. This would involve combining an improvement of the standard narrowing algorithm with an improvement in unification with equations and axioms. If formalized and implemented correctly, this should result in an even better algorithm than the current canonical narrowing.

References


Checking Sufficient Completeness
by Inductive Theorem Proving

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

Abstract. Sufficient completeness of an equational program ensures that each input can be fully evaluated to a data result. Checking this fundamental property for programs in expressive equational languages supporting conditional equations, types and subtypes, and rewriting modulo structural axioms is a challenging problem for which few methods currently exist. This work presents a new method that reduces sufficient completeness verification to a standard inductive theorem proving problem for a wide class of conditional equational programs in such languages.

1 Introduction

Sufficient completeness of an equational program is a fundamental property. It means that the recursive equations defining each of its functions cover all the required cases, so that any concrete input evaluates to actual data. Lack of sufficient completeness is a common programming error, similar to bugs missing or getting wrong some conditions in an imperative program. Besides being an essential correctness requirement, sufficient completeness is also an essential condition in program verification. For example, in inductive theorem proving, induction is typically based on data constructors, so that the correctness of an inductive proof essentially depends on the specification’s sufficient completeness.

As equational programming languages become more expressive, their very richness and power makes achieving sufficient completeness more challenging for several reasons: (i) the greater expressiveness of conditional equations can lead to missing some conditions needed for evaluation; (ii) the power and expressiveness of rewriting modulo structural axioms such as associativity and/or commutativity and/or identity allows recursive function definitions that use a rich variety of patterns in its recursive equations; but this greater power, as all power, has to be used wisely; (iii) types and subtypes naturally support case analysis and, together with structural axioms, allow definition of very sophisticated data structures; but this again requires greater care when defining functions.

Although methods for proving sufficient completeness have been studied since the 1970s to the present, e.g., [16, 33, 6, 20, 21, 2, 17, 19, 18, 1, 31, 30, 22, 32, 25], in general this is an undecidable property [21] whose verification can only be fully automated in restricted cases. For example, sufficient completeness of unconditional equational programs such that the lefthand sides of its recursive equations do not have repeated variables can be automatically checked by either: (i)
tree automata techniques, e.g., [6], which, using equational tree automata, have been generalized to support structural axioms, types and subtypes, and context-sensitive rewriting in [19, 18]; or (ii) Boolean operations on term patterns [23, 30]; in particular, [30] supports the general order-sorted case, involving types and subtypes. For more general and expressive classes of programs, verifying completeness becomes harder. As further discussed in Section 6, for expressive equational languages supporting conditional equations, structural axioms and order-sorted typing such as OBJ [15], CafeOBJ [13] and Maude [5], few sufficient completeness verification methods currently exist for general programs.

This work proposes a new method of proving sufficient completeness of programs in expressive equational languages such as the ones mentioned above. It offers four main advantages: (1) it applies to a class of order-sorted conditional equational programs modulo structural axioms that is quite general in actual practice; (2) it does not require developing a new sufficient completeness verification tool, because it reduces such verification to standard inductive theorem proving; any such prover supporting conditional order-sorted specifications modulo axioms can be used; (3) it supports a hierarchical proof methodology allowing simpler automated methods, such as those in [19], to be applied to subprograms, so that only more complex function definitions need to be dealt with; and (4) such a hierarchical proof methodology is shared with that in the companion paper [25], which supports verification of both ground confluence, and of sufficient completeness by methods that nicely complement the one proposed in this work. Proofs of all theorems are relegated to Appendix B.

2 Preliminaries

I assume familiarity with the notions of an order-sorted signature $\Sigma$ on a poset of sorts $(S, \leq)$, an order-sorted $\Sigma$-algebra $A$, and the term $\Sigma$-algebras $T_\Sigma$ and $T_\Sigma(X)$ for $X$ an $S$-sorted set of variables. I also assume familiarity with the notions of: (i) $\Sigma$-homomorphism $h : A \rightarrow B$ between $\Sigma$-algebras $A$ and $B$, so that $\Sigma$-algebras and $\Sigma$-homomorphisms form a category $\text{OSAlg}_\Sigma$; (ii) order-sorted (i.e., sort-preserving) substitution $\theta$, its domain $\text{dom}(\theta)$ and range $\text{ran}(\theta)$, and its application $t\theta$ to a term $t$; (iii) preregular order-sorted signature $\Sigma$, i.e., a signature such that each term $t$ has a least sort, denoted $\text{ls}(t)$; (iv) the set $\tilde{S} = S/(\geq \cup \leq)^+$ of connected components of a poset $(S, \leq)$ viewed as a DAG; and (v) for $A$ a $\Sigma$-algebra, the set $A_s$ of its elements of sort $s \in S$, and the set $A_s = \bigcup_{s \in S} A_s'$ of all elements in a connected component $[s] \in \tilde{S}$. We furthermore assume that all signatures $\Sigma$ have non-empty sorts, i.e., $T_\Sigma s \neq \emptyset$ for each $s \in S$. $[A \rightarrow B]$ denotes the $S$-sorted functions from $A$ to $B$. All these notions are explained in detail in [28, 14]. The material below is adapted from [29, 26, 25].

Order-Sorted Algebra and $E$-Unification. An OS equational theory is a pair $T = (\Sigma, E)$, with $E$ a set of (possibly conditional) $\Sigma$-equations. $\text{OSAlg}_{(\Sigma, E)}$ denotes the full subcategory of $\text{OSAlg}_\Sigma$ with objects those $A \in \text{OSAlg}_\Sigma$ such
that A \models E$, called the $(\Sigma, E)$-algebras. \textsc{Osalg}$_{(\Sigma, E)}$ has an initial algebra $T_{\Sigma,E}$. If $E \models T_{\Sigma,E}$, $T_{\Sigma,E}$ abbreviates $T_{\Sigma,E}$. The inference system in [28] is sound and complete for OS equational deduction, i.e., for any OS equational theory $(\Sigma, E)$, and $\Sigma$-equation $u = v$ we have an equivalence $E \vdash u = v \iff E \models u = v$. Deducibility $E \vdash u = v$ is abbreviated as $u =_E v$, called $E$-equality. An $E$-unifier of a system of $\Sigma$-equations, i.e., of a conjunction $\phi = \psi_1 \land \ldots \land \psi_n$ of $\Sigma$-equations, is a substitution $\sigma$ such that $u_1 \sigma =_E \psi_1 \sigma$, $1 \leq i \leq n$. An $E$-unification algorithm for $(\Sigma, E)$ is an algorithm generating a complete set of $E$-unifiers $\text{Unif}_E(\phi)$ for any system of $\Sigma$ equations $\phi$, where “complete” means that for any $E$-unifier $\sigma$ of $\phi$ there is a $\tau \in \text{Unif}_E(\phi)$ and a substitution $\rho$ such that $\sigma =_E (\tau \rho)|_{\text{dom}(\sigma) \cup \text{dom}(\tau)}$, where $=_E$ here means that for any variable $x$ we have $x\sigma =_E x(\tau \rho)|_{\text{dom}(\sigma) \cup \text{dom}(\tau)}$. The algorithm is finitary if it always terminates with a finite set $\text{Unif}_E(\phi)$ for any $\phi$. Given a set of equations $B$ used for deduction modulo $B$, a prerelular OS signature $\Sigma$ is called $B$-prerelular if for each $u = v \in B$ and substitutions $\rho$, $\text{ls}(up) = \text{ls}(vp)$.

Convergent Theories and Constructors. Given an order-sorted equational theory $E = (\Sigma, E \cup B)$, where $B$ is a collection of associativity and/or commutativity and/or identity axioms and $\Sigma$ is $B$-prerelular, we can associate to it a corresponding rewrite theory [27] $\bar{E} = (\Sigma, B, \bar{E})$ by orienting the equations $E$ as left-to-right rewrite rules. That is, each $(u = v) \in E$ is transformed into a rewrite rule $u \rightarrow v$. For simplicity we recall here the case of unconditional equations; for how conditional equations (whose conditions are conjunctions of equalities) are likewise transformed into conditional rewrite rules see, e.g., [24]. The main purpose of the rewrite theory $\bar{E}$ is to reduce the complex bidirectional reasoning with equations to the much simpler unidirectional reasoning with rules under suitable assumptions. We assume familiarity with the notion of subterm $t|_p$ of $t$ at a term position $p$ and of term replacement $t[w]_p$ of $t|_p$ by $w$ at position $p$ (see, e.g., [8]). The rewrite relation $t \rightarrow_{\bar{E},B} t'$ holds iff there is a subterm $t|_p$ of $t$, a rule $(u \rightarrow v) \in \bar{E}$ and a substitution $\theta$ such that $u \theta =_B t|_p$, and $t' = t[w|_p]$. We denote by $\rightarrow_{\bar{E},B}^{*}$ the reflexive-transitive closure of $\rightarrow_{\bar{E},B}$. The requirements on $\bar{E}$ allowing us to reduce equational reasoning to rewriting are the following: (i) $\text{vars}(v) \subseteq \text{vars}(u)$; (ii) $\text{sort-decreasingness}$: for each substitution $\theta$ we have $\text{ls}(u \theta) \geq \text{ls}(v \theta)$; (iii) strict $B$-coherence: if $t_1 \rightarrow_{\bar{E},B} t'_1$ and $t_1 =_B t_2$ then there exists $t_2 \rightarrow_{\bar{E},B} t'_2$ with $t'_1 =_B t'_2$; (iv) confluence (resp. ground confluence) modulo $B$: for each term $t$ (resp. ground term $t$) if $t \rightarrow_{\bar{E},B}^* v_1$ and $t \rightarrow_{\bar{E},B}^* v_2$, then there exist rewrite sequences $v_1 \rightarrow_{\bar{E},B}^* w_1$ and $v_2 \rightarrow_{\bar{E},B}^* w_2$

1. If $B = B_0 \cup U$, with $B_0$ associativity and/or commutativity axioms, and $U$ identity axioms, the $B$-prerelularity notion can be broadened by requiring only that: (i) $\Sigma$ is $B_0$-prerelular in the standard sense that $\text{ls}(up) = \text{ls}(vp)$ for all $u = v \in B_0$ and substitutions $\rho$; and (ii) the axioms $U$ oriented as rules $\bar{U}$ are $\text{sort-decreasing}$ in the sense explained below.
such that \( w_1 =_B w_2 \); (v) termination: the relation \( \rightarrow_{\vec{E},B} \) is well-founded (for \( \vec{E} \)
conditional, we require operational termination [24]). If \( \vec{E} \) satisfies conditions (i)–(v) (resp. the same, but (iv) weakened to ground confluence modulo \( B \)), then it is called convergent (resp. ground convergent). The key point is that then, given a term (resp. ground term) \( t \), all terminating rewrite sequences \( t \rightarrow_{\vec{E},B}^* w \) end in a term \( w \), denoted \( t \downarrow_{\vec{E}} \), that is unique up to \( B \)-equality, and its called \( t \)’s canonical form. Three major results then follow for the ground convergent case: (1) for any ground terms \( t, t' \) we have \( t =_{E\cup B} t' \) iff \( t \downarrow_{\vec{E}} = B t' \downarrow_{\vec{E}} \), (2) the \( B \)-equivalence classes of canonical forms are the elements of the canonical term algebra \( C_{\Sigma;E,B} \), where for each \( f : s_1 \ldots s_n \rightarrow s \) in \( \Sigma \) and \( B \)-equivalence classes of canonical terms \( \{t_1\}, \ldots, \{t_n\} \) with \( bs(t_i) \leq s_i \) the operation \( f_{C_{\Sigma;E,B}} \) is defined by the identity: \( f_{C_{\Sigma;E,B}}(\{t_1\} \ldots \{t_n\}) = \{f(t_1 \ldots t_n)\}_{\vec{E}} \), and (3) we have an isomorphism \( T_E \cong C_{\Sigma;E,B} \).

A ground convergent rewrite theory \( \vec{E} = (\Sigma, B, \vec{E}) \) is called sufficiently complete with respect to a subsignature \( \Omega \), whose operators are then called constructors, iff for each ground \( \Sigma \)-term \( t \), \( t \downarrow_{\vec{E}} \in T_D \). Furthermore, for \( \vec{E} = (\Sigma, B, \vec{E}) \) sufficiently complete w.r.t. \( \Omega \), a ground convergent rewrite subtheory \( (\Omega, B_{\Omega}, \vec{E}_{\Omega}) \subseteq (\Sigma, B, \vec{E}) \) is called a constructor subspecification iff \( T_{\vec{E}}|_{\Omega} \cong T_{\Omega}\cup B_{\Omega} \). If \( E_\Omega = \emptyset \), then \( \Omega \) is called a signature of free constructors modulo axioms \( B_\Omega \). Note that \( \vec{E} = (\Sigma, B, \vec{E}) \) is sufficiently complete with respect to \( \Omega \), iff each ground \( \Sigma \)-term \( f(u_1, \ldots, u_n) \) with \( f \in \Sigma \\setminus \Omega \) and \( u_i \in T_D \), \( 1 \leq i \leq n \), is \( \vec{E}, B \)-reducible, i.e., \( f(u_1, \ldots, u_n) \rightarrow_{\vec{E},B} t \) for some \( t \in T_D \).

Generator Sets generalize standard structural induction on the constructors of a sort. They are particularly useful for inductive reasoning when constructors obey structural axioms \( B \) like associativity or associativity-commutativity for which structural induction may be ill-suited. A generator set for a sort \( s \) is a set of constructor terms of sort \( s \) or smaller such that, up to \( B \)-equality, any ground constructor term of sort \( s \) is a ground substitution instance of one of the patterns in the generator set. Here is the general definition (identity axioms are not needed thanks to the theory transformation\(^2\) \( \vec{E} \mapsto \vec{E}_U \) in [9]):

**Definition 1.** For \( \Omega \) an order-sorted signature of constructors which may satisfy axioms \( B \) of associativity and/or commutativity, and \( s \) a sort in \( \Omega \), a \( B \)-generator set for sort \( s \) is a finite set of terms \( \{u_1, \ldots, u_k\} \), with \( u_1, \ldots, u_k \in T_D(X)_s \) and such that

\[
T_{\Omega/B,s} = \bigcup_{1 \leq i \leq k} \{[u_i, \rho] \in T_{\Omega/B,s} \mid \rho \in [X \rightarrow T_D]\}.
\]

Checking the Correctness of Generator Sets. How do we know that a proposed generator set \( \{u_1, \ldots, u_k\} \) it truly one modulo axioms \( B \) for a given

\(^2\) If a theory \( \vec{G} \) has axioms \( B \equiv U \), with \( B \) associative and/or commutative axioms and \( U \) unit element axioms, then the axioms \( U \) can be eliminated by turning them into rules \( \vec{U} \) by means of the semantics-preserving theory transformation \( \vec{G} \mapsto \vec{G}_U \), defined in [9], so that the axioms of the semantically equivalent \( \vec{G}_U \) are just \( B \).
sort \( s \) and constructors \( \Omega \)? Assuming that the terms \( u_1, \ldots, u_k \) are all linear, i.e., have no repeated variables —which is the usual case for generator sets— this check can be reduced to an automatic *sufficient completeness check* with Maude’s Sufficient Completeness Checker (SCC) tool [19], which is based on tree automata decision procedures modulo axioms \( B \). The reduction is extremely simple: define a new unary predicate \( s : s \to \text{Bool} \) with equations \( s(u_i) = \text{true}, 1 \leq i \leq k \). Then, \( \{u_1, \ldots, u_k\} \) is a correct generator set for sort \( s \) modulo \( B \) for the constructor signature \( \Omega \) iff the predicate \( s \) is sufficiently complete, which can be automatically checked by the SCC tool. Furthermore, if \( \{u_1, \ldots, u_k\} \) is not a generator set for sort \( s \), the SCC tool will output a useful counterexample.

**Inductive Theorem Proving.** An inductive theorem prover implements a sound inference system to prove *inductive theorems* \( \varphi \) in a given equational theory \( \mathcal{E} \), i.e., formulas \( \varphi \) such that \( T_{\mathcal{E}} \models \varphi \). Although the methods I present for proving sufficient completeness by inductive theorem proving do not depend on the given inductive inference system, the examples presented in Section 5 will use an extended version of the inductive inference system for order-sorted equational specifications presented in [26].

### 3 A Hierarchical Methodology

I present a hierarchical methodology to prove sufficient completeness by inductive theorem proving. This methodology uses the same assumptions about the input theory and about theory hierarchies as those in a similar hierarchical method for proving ground confluence and sufficient completeness presented in the companion paper [25], which is not based on the use of a standard inductive theorem prover. Thus, the two methods help each other and can profitably be used in combination. Since this section focuses on the assumptions and infrastructure common to both methods, it follows closely the presentation in [25].

**Basic Assumptions about \( \bar{\mathcal{E}} \).** We assume throughout a, possibly conditional, equational theory \( \mathcal{E} = (\Sigma, E \cup U \cup B) \) such that: (i) \( \Sigma \) decomposes as a disjoint union \( \Sigma = \Delta \cup \Omega \), where \( \Omega \) are the —intended but not yet proved to be— constructor symbols that furthermore are free modulo \( B \) and \( \Delta \) are the intended defined symbols; (ii) \( B \) is any combination of associativity and/or commutativity axioms, but any binary \( f \in \Delta \) may not satisfy any axioms except commutativity;\(^3\) (iii) \( \hat{U} \) are sort-decreasing unit axiom rules of the form \( c(e, x) \to x \) or

---

\(^3\) For any \( f \) that is commutative we always assume a top typing \( f : s \to s_0 \) with all other typings of the form \( f : s' \to s'_0 \), with \( s \leq s', s_0 \leq s'_0 \). Regarding the absence of unit element axioms, they are precisely the equations \( \hat{U} \), that will be used as rules \( \hat{U} \) (see, e.g., Example 4, and Footnote 2). I.e., our results apply as well to theories \( \bar{\mathcal{G}} \) with axioms \( B \cup \hat{U} \) such that \( \bar{\mathcal{G}} \) has the properties (i)-(vi) listed in what follows.

\(^4\) Since axioms \( B \) are primarily used to specify constructor data structures, in actual practice, limiting axioms for defined symbols is a mild restriction. Furthermore, as explained in [25, Footnote 3], this restriction on axioms can be lifted *a posteriori* by further inductive theorem proving.
\[ c(x, e) \rightarrow x, \text{ where } c \text{ is a constructor name and } e \text{ is an } \Omega\text{-term. However, } c(x, e) \text{ is not an } \Omega\text{-term because some of } c\text{'s type declarations do not belong to } \Omega \text{ (see Examples 4 and 5 in Section 5). This makes it possible for constructors to be free modulo } B \text{ in spite of such unit rules; (iv) } E = \bigcup_{f \in \Delta} E_f, \text{ where for each } f \in \Delta, \text{ its associated rewrite rules } \tilde{E}_f \text{ are sort-decreasing and have the form: } 
\tilde{E}_f = \{[i] : f(u_i) \rightarrow r_i \mid I_i \in I\} \text{ such that: (a) the } u_i \text{ are } \Omega\text{-terms; and (b) for each } i \in I, I_i = \bigwedge_{j \in I} w_{ij} = w'_{ij} \text{ and } \text{vars}(f(u_i)) \supseteq \text{vars}(r_i) \cup \text{vars}(I_i). \}

\text{(v) There is a } B\text{-compatible recursive path order (RPO) } (\text{see [8]} \text{) such that for each } i \in I, f(u_i) > r_i \text{ and, for } j \in J, f(u_i) > w_j \text{ and } f(u_i) > w'_{ij}, \text{ which makes the rules } \tilde{E} \cup \tilde{U} \text{ operationally terminating modulo } B. \text{ (vi) The rules } \tilde{E} \cup \tilde{U} \text{ are strictly } B\text{-coherent.}

\text{The main goal of this paper is to develop a hierarchical method based on inductive theorem proving to prove that a theory } \tilde{E} \text{ enjoying properties (i)--(vi) above is sufficiently complete with respect to a constructor signature } \Omega. \text{ We first need to consider theory hierarchies based on the "call graph" of } \tilde{E}.\]

\textbf{Call Graph and Theory Hierarchies.} We assume that all function symbols in } \Delta \text{ are subsort-overloaded, i.e., for any } f : s_1 \ldots s_n \rightarrow s \text{ and } f' : s'_1 \ldots s'_n \rightarrow s' \text{ we have } [s] = [s'], \text{ and } [s_i] = [s'_i], 1 \leq i \leq n. \text{ This can always be achieved by renaming any "ad-hoc overloaded" symbols — i.e., symbols } f \text{ with typings } f : s_1 \ldots s_n \rightarrow s \text{ and } f' : s'_1 \ldots s'_n \rightarrow s' \text{ failing the above condition — that might exist in } \Delta. \text{ Let } F_\Delta \text{ be the set of } \text{names for the function symbols in } \Delta, \text{ disregarding their typing. The } \text{calling relation} \text{ is a binary relation } C \text{ on } F_\Delta, \text{ where for each } f, g \in F_\Delta, (f, g) \in C \text{ if there exists a rule } f(u_i) \rightarrow r_i \text{ if } I_i \text{ in } \tilde{E}_f \text{ such that the function symbol } g \text{ occurs in either } r_i \text{ or in } I_i. \text{ Let } C^* \text{ denote the reflexive-transitive closure of } C, \text{ and } =_C \text{ the equivalence relation on } F_\Delta \text{ defined by the equivalence: } f =_C g \text{ iff } fC^*g \text{ and } C^*f. \text{ Then, the quotient set } F_\Delta/=_C \text{ has an associated partial order defined by the equivalence } [f] \geq [g] \iff fC^*g. \text{ The hierarchical method we propose is based on a hierarchy of theory inclusions chosen as follows. Given our theory } \tilde{E} \text{ we: (i) identify a subtheory } \tilde{E}_0 \text{ having subsignature } \Delta_0 \subseteq \Omega \text{ containing all the subsort-overloaded typings of any } f \in \Delta_0 \text{ and having rules } \tilde{U} \cup \tilde{E}_0, \text{ with } \tilde{E}_0 = \bigcup_{f \in \Delta_0} \tilde{E}_f, \text{ and axioms } B_0 = B_{\Delta_0 \subseteq \Omega} = \bigcup_{f \in \Delta_0 \subseteq \Omega} B_f, \text{ where } B_f \text{ are the associative and/or commutative axioms, if any, for } f \text{ in } B, \text{ and such that } \tilde{E}_0 \cup \tilde{U} \text{ is sufficiently complete with respect to } \Omega \text{ and ground convergent. Of course, we should choose } \tilde{E}_0 \text{ as big as possible; in the worse case we may have } \tilde{E}_0 = \emptyset \text{ and keep only } \tilde{U}. \text{ We furthermore assume that we can find a sequence of theory inclusions:}

\tilde{E}_0 \subset \tilde{E}_1 \subset \ldots \tilde{E}_{n-1} \subset \tilde{E}_n

\text{such that: (a) } \tilde{E}_n = \tilde{E}, \text{ (b) each } \tilde{E}_k \text{ has signature } \Delta_k \subseteq \Omega \text{ containing all subsort-overloaded typings of any } f \in \Delta_k \text{ and having axioms } B_k = B_{\Delta_k \subseteq \Omega} \text{ and, besides } \tilde{U}, \text{ rules } \tilde{E}_k = \bigcup_{f \in \Delta_k} \tilde{E}_f, \text{ where for each } k \geq 1 \text{ and each rule } [i] : f(u_i) \rightarrow r_i \text{ if } I_i \text{ in } \tilde{E}_f, \text{ the condition } I_i \text{ is a conjunction of } \tilde{E}_{k-1}\text{-equalities, (c) for each } 0 \leq k < k + 1 \leq n, \text{ there exists a function symbol } g \in F_{\Delta_{k+1}\setminus F_{\Delta_k}} \text{ such that}
\( F_{\Delta_{n+1}} = F_{\Delta_n} \cup \{g\} \); that is, we add all the symbols in a new \( \equiv_{C} \)-equivalence class \( \{g\} \) to climb up each step in the theory hierarchy.

**The Hierarchical Proof Method.** All now boils down to finding proof methods to climb up the theory hierarchy one step at a time. We then repeat this method \( n \) times, with \( n \) the length of the chain of theory inclusions. That is, we focus on a single theory inclusion \( \bar{\Sigma}_0 \subset \bar{\Sigma} \), where \( \bar{\Sigma}_0 \) has already been proved ground convergent and sufficiently complete with respect to the constructor signature \( \Omega \), and then prove that \( \bar{\Sigma} \) is also ground convergent and sufficiently complete as follows. First of all, we define a new theory \( \bar{\Sigma}^A \), with the same rules \( \bar{\Sigma} \cup \bar{\Delta} \) as in \( \bar{\Sigma} \), and having also a theory inclusion \( \bar{\Sigma}_0 \subset \bar{\Sigma}^A \), but where, if \( \bar{\Sigma}_0 \) and \( \Sigma \) are the respective signatures of \( \bar{\Sigma}_0 \) and \( \bar{\Sigma} \), and \( \Delta = \Sigma \setminus \Sigma_0 \), then \( \bar{\Sigma}^A \) has a signature \( \Sigma^A \) that extends \( \Sigma_0 \) by: (i) adding to each connected component of the poset of sorts \((\bar{\Sigma}, \leq)\) the kind \([s]\) as a new top sort, i.e., \( \forall s' \in [s], s' \leq [s]\), and (ii) lifting to the kind levels all \( f \in \Sigma \). That is, we extend the function symbols of \( \Sigma_0 \) by adding for each \( f : s_1 \ldots s_n \rightarrow s \), \( n \geq 1 \), in \( \Sigma \) a function symbol \( f : [s_1] \ldots [s_n] \rightarrow [s] \) to \( \Sigma^A \). In \( \bar{\Sigma}^A \) the axioms \( B \) are lifted to kinds. Note that \( \Sigma^A \) adds no new terms to the original sorts \( s \in \bar{\Sigma} \), i.e., \( T_{\Sigma^A}(X)_s = T_{\bar{\Sigma}_0}(X)_s \).

For three concrete examples of the \( \bar{\Sigma}^A \) construction, see modules \( \text{OE-Delta} \), \( \text{NAT-Presburger-Delta} \) and and \( \text{MultiSet-Algebra-Delta} \) in Section 5. The hierarchical proof methodology then proceeds as follows:

1. We first prove that \( \bar{\Sigma}^A \) is ground convergent.
2. We then prove that for any \( f \in \Delta \), maximal typing \( f : s_1, \ldots, s_n \rightarrow s \) and ground constructor substitution \( \rho \), the term \( f(x_1, \ldots, x_n)\rho \), with \( x_i \) of sort \( s_i, 1 \leq i \leq n \), can be rewritten with some rule in \( \bar{\Sigma}^A \).
3. (1) and (2) actually prove that \( \bar{\Sigma} \) is ground convergent and sufficiently complete with respect to \( \Omega \).

Methods for proving (1), as well as a proof that (1) and (2) imply (3) can be found in [25]. In this paper we focus on a new method that reduces proving (2) to a proof by standard inductive theorem proving.

## 4 Proving Sufficient Completeness Inductively

The reduction of sufficient completeness proofs to inductive proofs is based on a new general theory transformation \( \bar{\Sigma} \mapsto \bar{\Sigma} \); defined below.

**The \( \bar{\Sigma} \mapsto \bar{\Sigma} \): Transformation.** We assume a ground convergent and possibly conditional theory (with rules having no extra variables in their right hand side or condition) \( \bar{\Sigma} = (\Sigma, B, \bar{\Delta}) \) such that: (i) \( B \) are associativity and/or commutativity axioms and \( \Sigma \) is \( B \)-preregular; (ii) all its function symbols are subsort-overloaded; (iii) its poset of sorts \((\bar{\Sigma}, \leq)\) is such that each connected component \([s]\) of \( \bar{\Sigma} \) has a top sort, which we denote \( \top_{[s]} \in [s] \); (iv) any \( f : s_1 \ldots s_n \rightarrow s \), \( n \geq 1 \), in \( \Sigma \) has also a typing \( f : \top_{[s_1]} \ldots \top_{[s_n]} \rightarrow \top_{[s]} \); (v) there is a \( B \)-compatible RPO \( \succ \) making \( \bar{\Sigma} \) operationally terminating. In what follows \( \bar{\Sigma} \) will
always be a theory of the form $\vec{E}^\Delta$ as defined in the methodology of Section 3. The transformation $\vec{E} \rightarrow \vec{E}$ maps $\vec{E} = (\Sigma, B, \vec{E})$ to $\vec{E}^\Delta = (\Sigma, B, \vec{E} \cup \vec{E}_\Sigma)$, where:

- $\Sigma$, extends $\Sigma$ by adding: (i) a fresh new sort $\text{Pred}$ of predicates with a constant $\texttt{tt}$ not related to any other sort in the subsort order; and (ii) for each connected component $[s]$ of the sort poset $(S, <)$ in $\Sigma$ and each $s' \in [s]$ a unary function symbol $\prec s' : \Sigma \rightarrow \text{Pred}$ called a sort predicate.

- the set $\vec{E}_\Sigma$ of rewrite rules contains: (i) for each $s \in S$ a rule,

$$x : s \rightarrow \texttt{tt}$$

where $x$ is a variable of sort $s$; (ii) for each $f : s_1 \ldots s_n \rightarrow s$ in $\Sigma$, $n \geq 0$, a rule,

$$f(x_1, \ldots, x_n) : s \rightarrow \texttt{tt} \text{ if } x_1 : s_1 = \texttt{tt} \land \ldots \land x_n : s_n = \texttt{tt}$$

where $x_i$ has sort $\top_{[s_i]}$, $1 \leq i \leq n$; (iii) for each $s < s'$ in $(S, <)$ a rule,

$$x : s' \rightarrow \texttt{tt} \text{ if } x : s = \texttt{tt}.$$

where $x$ has sort $\top_{[s]}$.

Two key results about the $\vec{E} \rightarrow \vec{E}$: transformation include:

**Theorem 1.** Under the assumptions on $\vec{E}$, $\vec{E}$ is operationally terminating.

**Theorem 2.** Under the assumptions on $\vec{E}$, $\vec{E}$ is ground convergent. Furthermore, for each $s \in S$ and $t \in T_{\Sigma, \top_{[s]}}$ we have the equivalences:

$$(t : s)_{\vec{E}} = \texttt{tt} \iff s \gg \text{ls}(t!_{\vec{E}}) \quad \text{and} \quad (t : s)_{\vec{E}} = t!_{\vec{E}} : s \iff s \not\gg \text{ls}(t!_{\vec{E}}).$$

**Reducing Sufficient Completeness Checking to Inductive Reasoning.**

As already mentioned our focus of interest is in the transformation: $\vec{E}^\Delta \rightarrow \vec{E}^\Delta$, which will give us the desired reduction of sufficient completeness checking to inductive theorem proving. Here is the main theorem:

**Theorem 3.** Let $\vec{E}_0 \subset \vec{E}$ and $\vec{E}_0^\Delta \subset \vec{E}^\Delta$ be theory inclusions satisfying the assumptions in Section 3, where $\vec{E}_0$ is sufficiently complete with respect to $\Omega$ and $\vec{E}^\Delta$ has already been proved ground convergent. Then, for all $f \in \Delta$, maximal typing $f : s_1, \ldots, s_n \rightarrow s$ of $f$, and all ground constructor substitutions $\rho$, the term $f(x_1, \ldots, x_n)\rho$, with $x_i$ of sort $s_i$, $1 \leq i \leq n$, can be rewritten with some rule in $\vec{E}_0^\Delta$ if for $E_\Delta = \{f_1, \ldots, f_k\}$, and each maximal typing $f_j : s_{i_1} \ldots s_{i_k} \rightarrow s_i$, $i \in I_j$, for each $f_j$, $1 \leq j \leq k$, we have,

$$T_{\vec{E}, \Delta} = \bigwedge_{1 \leq j \leq k, i \in I_j} f_j(x_{i_1}^i, \ldots, x_{i_k}^i) : s_i = \texttt{tt}$$

where $x_n^i$ has sort $s_{i_m}^i$, $1 \leq m \leq n_j$.  

109
The reason for proving the above conjunction as an inductive theorem, as opposed to proving each conjunct \( f_j(x_1^1, \ldots, x_{n_j}^1) : s_i = tt \) separately, is that, since \( F_\Delta \) is one of the nodes in the calling graph of \( \vec{E} \), the symbols in \( F_\Delta \) call each other, typically due to some mutual recursion. Therefore, it may be considerably easier to prove the entire conjunction than proving each conjunct separately. Let us see a simple example.

**Example 1.** (Odd and Even). Let \( \mathcal{OE} \) be the theory of the Peano Naturals with sort \( Nat \), which imports the Booleans, has constructors 0 and \( s : Nat \rightarrow Nat \), and with \( \Delta \) the predicates \( odd : Nat \rightarrow Bool \) and \( even : Nat \rightarrow Bool \), which are defined by rules: \( odd(0) \rightarrow false, even(0) \rightarrow true, odd(s(n)) \rightarrow \neg even(n), \) and \( even(s(n)) \rightarrow \neg odd(n) \), so that the calling graph has the single equivalence class node \( \{odd, even\} \). Then, to prove sufficient completeness with the above method we just need to prove in \( \mathcal{OE}:^\Delta \) the inductive theorem:

\[
odd(n);Nat \land even(n);Nat
\]

Theorem 3 has a quite useful corollary for finite sorts.

**Corollary 1.** Under the assumptions and notation in Theorem 3, if a maximal typing in \( \Delta \), say, \( f_j : s_i^1 \ldots s_i^{n_j} \rightarrow s_i \) is such that \( T_{ij/B,s_i} \) is a finite set, say, \( T_{ij/B,s_i} = \{[u_1], \ldots, [u_m]\} \), then Theorem 3 still holds replacing the conjunct \( f_j(x_1^i, \ldots, x_{n_j}^i) : s_i = tt \) by the disjunction:

\[
\bigvee_{1 \leq k \leq m} f_j(x_1^i, \ldots, x_{n_j}^i) = u_k.
\]

Theorem 3 and Corollary 1 provide the desired reduction of sufficient completeness checking for \( \vec{E} \) to inductive theorem proving in \( \vec{E}:^\Delta \). In fact, if all operators in \( \Delta \) have finite sorts, the conjunction of disjunctions in Corollary 1 can be inductively proved just in \( \vec{E}:^\Delta \).

**Example 2.** (Odd and Even Revisited). From the above corollary it follows that to prove sufficient completeness of \( \mathcal{OE} \) it is enough to prove that

\[
(odd(n) = true \lor odd(n) = false) \land (even(n) = true \lor even(n) = false)
\]

is an inductive theorem of \( \mathcal{OE}:^\Delta \).

**5 Some Examples**

As a warmup exercise, let us prove sufficient completeness of the \( \mathcal{OE} \) theory in Example 1. Of course, since the equations are left-linear, a different, automatic proof based on tree automata can be given for \( \mathcal{OE} \) using, for example, Maude’s SCC tool [19]. But one can easily find similar mutually recursive function definitions—for example, conditional ones—outside the scope of automated tools.

**Example 3.** In Maude, the theory \( \mathcal{OE}:^\Delta \) can be specified as follows:
fmod OE-DELTA is protecting BOOL-OPS .
sort Nat .
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor] .
op even : [Nat] -> [Bool] .
var n : Nat .
eq odd(0) = false .
eq odd(s(n)) = not(even(n)) .
eq even(0) = true .
eq even(s(n)) = not(odd(n)) .
endfm

where [Nat] and [Bool] are the “kind” supersorts automatically added by Maude above, respectively, Nat and Bool, and therefore need not be declared.
The theory BOOL-OPS can be trivially shown to be sufficiently complete by truth table inspection, so in this case the theory OE_0 in the inclusion OE_0 ⊆ OE^Δ is just BOOL-OPS together with the constructors {0, s}. Also, OE-DELTA can easily be checked to be confluent and terminating. As pointed out in Example 2, we just need to prove that:

\[(\text{odd}(n) = \text{true} \lor \text{odd}(n) = \text{false}) \land (\text{even}(n) = \text{true} \lor \text{even}(n) = \text{false})\]

is an inductive theorem of OE-DELTA. We can do so by standard induction on \(n\) (which is a special case of the GSI rule in \(^5\) [26]). The Base Case can be proved automatically by the Equality Predicate Simplification rule (EPS) in [26]. For the Induction Step we have induction hypotheses: \(\text{odd}(\overline{k}) = \text{true} \lor \text{odd}(\overline{k}) = \text{false}\) and \(\text{even}(\overline{k}) = \text{true} \lor \text{even}(\overline{k}) = \text{false}\), and need to prove the conjunction:

\[(\text{odd}(s(\overline{k})) = \text{true} \lor \text{odd}(s(\overline{k})) = \text{false}) \land (\text{even}(s(\overline{k})) = \text{true} \lor \text{even}(s(\overline{k})) = \text{false})\]

where \(\overline{k}\) is a fresh constant of sort Nat. Using the EPS simplification rule this goal reduces to:

\[\neg(\text{even}(\overline{k})) = \text{true} \lor \neg(\text{even}(\overline{k})) = \text{false}) \land \neg(\text{odd}(\overline{k})) = \text{true} \lor \neg(\text{odd}(\overline{k})) = \text{false})\]

Applying the Split rule (SP) in [26] with the induction hypothesis disjunction \(\text{even}(\overline{k}) = \text{true} \lor \text{even}(\overline{k}) = \text{false}\) we get subgoals:

\(\text{even}(\overline{k}) = \text{true} \rightarrow\)

\[\neg(\text{even}(\overline{k})) = \text{true} \lor \neg(\text{even}(\overline{k})) = \text{false}) \land \neg(\text{odd}(\overline{k})) = \text{true} \lor \neg(\text{odd}(\overline{k})) = \text{false})\]

and

\(\text{even}(\overline{k}) = \text{false} \rightarrow\)

\(^5\) The inference rules in [26] have been extended in work submitted for publication. In the examples in the section some rules will be used in their extended form.
Applying the Inductive Contextual Rewriting (ICC) rule in [26], both of these subgoals automatically simplify to the goal:

\[ \neg (\text{odd}(\bar{k})) = \text{true} \lor \neg (\text{odd}(\bar{k})) = \text{false} \]

After applying the SP rule to this goal with the disjunction hypothesis \( \text{odd}(\bar{k}) = \text{true} \lor \text{odd}(\bar{k}) = \text{false} \) we again get two goals that can be automatically discharged by means of the ICC rule. This finishes the sufficient completeness proof for \( \mathcal{OE} \).

I next present a Presburger arithmetic specification for which I am not aware of any proof of sufficient completeness by any other method than the one here.

Example 4. (Presburger Arithmetic) Consider the following specification of Presburger Arithmetic for the naturals:

```plaintext
fmod NAT-PRESBURGER is protecting TRUTH-VALUE .
  sorts NzNat Nat .
  subsort NzNat < Nat .
  op 0 : -> Nat [ctor] .
  op 1 : -> NzNat [ctor] .
  op _>_ : Nat Nat -> Bool .
  vars n m : Nat . var k : NzNat .
  eq n + 0 = n [variant] .
  eq k > 0 = true [variant] .
  eq n + k > n = true [variant] .
  eq 0 > m = false [variant] .
  eq n > n = false [variant] .
  eq n > n + m = false [variant] .
endfm
```

The equations are size-decreasing and therefore clearly terminating. It is easy to check that they are also confluent using Maude’s Church-Rosser Checker [10]. Furthermore, these equations enjoy the finite variant property [7, 12]; this can be easily checked in Maude by the method proposed in [4]. Therefore, \( E \cup B \)-unification in this theory is finitary [12]. The [variant] attribute allows Maude to use this knowledge to compute \( E \cup B \)-unifiers. Note that the lefthand sides of the second, fourth and fifth rules for \( > \) are non-linear. This places the sufficient completeness checking problem outside the scope of equational tree automata techniques such as those used in [19]. That the equations defining \( > \) are sufficiently complete is intuitively clear; but giving a formal proof in a suitable
inference system is a different matter. Here is where the new inductive methodology is helpful. Of course, calling $\vec{E}$ the entire theory, and $\vec{E}_0$ the theory obtained by dropping the $\cdot$ operator, we get a subtheory $\vec{E}_0$ that is convergent and sufficiently complete. This is intuitively clear, since any term of sort Nat is either 0, and thus a constructor, or is of the form $1 + \cdot + 1$, and therefore a constructor term. This can be automatically checked with Maude’s SCC tool [19]. Our theory $\vec{E}^\Delta$ is then:

```plaintext
fmod NAT-PRESBURGER-DELTA is protecting TRUTH-VALUE .

sorts NzNat Nat .
subsort NzNat < Nat .

op 0 : -> Nat [ctor] .
op 1 : -> NzNat [ctor] .
op _+_ : [Nat] [Nat] -> [Nat] [assoc comm] .

vars n m : Nat . var k : NzNat .

eq n + 0 = n [variant] .
eq k > 0 = true [variant] .
eq n + k > n = true [variant] .
eq 0 > m = false [variant] .
eq n > n = false [variant] .
eq n > n + m = false [variant] .
endfm
```

where $\vec{E}^\Delta$ can easily be checked to be terminating and convergent. Since the sort Bool is finite, we can use Corollary 1 to reduce proving the sufficient completeness of $\vec{E}$ to proving in $\vec{E}^\Delta$ the inductive theorem:

$$x > y = true \lor x > y = false$$

with $x, y$ of sort Nat. Let us prove this theorem using the inference system proposed in [26]. Using the generator set \{0, 1, + k\} for sort Nat, where $k$ has sort NzNat, we can induct on, say, $x$ and apply the Generator Set Induction rule (GSI) to get three sub goals: (1) $0 > y = true \lor 0 > y = false$, (2) $1 > y = true \lor 1 > y = false$, and (3) $\overline{k} + 1 > y = true \lor \overline{k} + 1 > y = false$ with induction hypothesis $\overline{k} > y = true \lor \overline{k} > y = false$, where $\overline{k}$ is a fresh constant of sort NzNat.

By applying the Case rule (CAS) (which is similar to GSI but does not add induction hypotheses) to $y$ with the same generator set for Nat, both goals (1) and (2) split into three subgoals each. For example, goal (1) generates subgoals: (1.1) $0 > 0 = true \lor 0 > 0 = false$, (1.2) $0 > 1 = true \lor 0 > 1 = false$, (1.3) $0 > k' + 1 = true \lor 0 > k' + 1 = false$, with $k'$ a fresh variable of sort NzNat. All such subgoals can be automatically discharged by the Equality Predicate...
The key point to observe is that both goals are clauses whose respective conditions are equations between terms in a theory $\mathcal{E}^A$ enjoying the finite variant property, and therefore having a finitary $\mathcal{E}^A$-unification algorithm. The “only” problem is that, to perform such a variant unification, we would like to replace the fresh constant $k$ by fresh variables $k$ of the same sort. In fact, using an extended version of the Constructor Variant Unification Left rule (CVUL) — which allows and justifies this replacement of fresh constants by fresh variables— we can do just that.

Applying CVUL to (3.1) involves solving the premise equation $k > y = true$, which yields two constructor variant unifiers: $\alpha_1 = \{y \mapsto 0\}$, $\alpha_2 = \{k \mapsto p + q, y \mapsto p\}$, where $p,q$ have of sort $\mathbb{N}z$Nat. Since the “magic” of the CVUL rule involves converting the original and resulting variables that correspond to constants (in this case, $k$, $p$ and $q$) back into fresh constants, we then get the following instantiated conclusions as subgoals:

\[(3.1.1) \quad \overline{k} + 1 > 0 = true \lor \overline{k} + 1 > 0 = false \]
\[(3.1.2) \quad \overline{p} + \overline{q} > \overline{p} = true \lor \overline{p} + \overline{q} > \overline{p} = false \]

We can likewise apply CVUL to (3.2) by solving the premise equation $k > y = false$, which yields the single constructor variant unifier $\beta = \{y \mapsto k + z\}$, where $z$ has sort $\text{Nat}$. We then get the instantiated conclusion:

\[(3.2.1) \quad \overline{k} + 1 > \overline{k} + z = true \lor \overline{k} + 1 > \overline{k} + z = false \]

which can be discharged by applying the CAS rule to variable $z$ —with the same generator set for sort $\text{Nat}$— followed by EPS simplification.

This finishes the proof of sufficient completeness for the above Presburger arithmetic specification. This formal proof is important because, together with the fact that the constructors are free modulo associativity and commutativity, it ensures that we can use this specification to decide quantifier-free Presburger arithmetic formulas using the variant satisfiability algorithm in [29].

The previous example has illustrated the fact that proving the sufficient completeness of unconditional theories when lefthand sides of rules are nonlinear can be nontrivial. The case of conditional theories can be even harder. The following example, besides illustrating the application of these method to conditional
theories for a function whose result sort is infinite, does also illustrate that the
hierarchical framework used here is the same as the one used in the companion
document [25], where a theory of multisets of numbers was used as a running
equation. The example below illustrates that the method for proving sufficient
completeness in [25] — based on constrained patterns — and the one presented here
complement each other: one can use either method. In [25], sufficient completeness
of the intersection function \( \cdot \cap \cdot \) which is specified by conditional rules, was
proved. Here I prove it instead by induction, counting on the ground convergence
of the corresponding theory \( \mathcal{E}^\Delta \) already proved in [25]. The Maude specification
for \( \mathcal{E}^\Delta \) is given below. That for the original theory \( \mathcal{E} \) is given in Appendix A.

Example 5. (Multisets). The theory \( \mathcal{E}^\Delta \) for multisets is as follows:

```plaintext
fmod MULTISET-ALGEBRA-DELTA is
  protecting TRUTH-VALUE .
  sorts Nat NeMult Mult .
  subsort Nat < NeMult < Mult .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op mt : -> Mult [ctor] .
  op _,_ : Mult Mult -> Mult [assoc comm] .
  op _,_ : NeMult NeMult -> NeMult [ctor assoc comm] .
  op _,in_, : Nat Mult -> Bool .
  op _, \=, _ : Mult Mult -> Mult .
  op s : [Mult] -> [Mult] [ctor] .
  op _,_ : [Mult] [Mult] -> [Mult] [assoc comm] .
  op _,=_, : [Mult] [Mult] -> [Bool] [comm] .
  op _, \=, _ : [Mult] [Mult] -> [Mult] .
  vars n m k : Nat .
  vars U V W : Mult .
  eq U,mt = U .
  eq n .=. n = true .
  eq 0 .=. s(n) = false .
  eq n in mt = false .
  eq n in n = true .
  eq n in n = false if n .=. m = false .
  eq n in (n,U) = true .
  eq n in (m,U) = false if (n .=. m) = false /
    (n in U) = false .
  eq mt \ U = mt .
```

115
In this case, the subtheory $E_0$ is everything except the kinds, the operator $\land$ and its defining equations. To show that $\land$ is sufficiently complete, we need to prove in $\vec{E} : \Delta$ the inductive theorem:

$$U \land V : \text{Mult} = \text{tt}$$

We apply the GSI rule to variable $U$ with generator set $\{\emptyset, x, (y, Q)\}$ with $x, y$ of sort $\text{Nat}$ and $Q$ of sort $\text{NeMult}$ and get subgoals: (1) $\emptyset \cap V : \text{Mult} = \text{tt}$, which is automatically discharged by $\text{EPS}$ simplification, (2) $x \cap V : \text{Mult} = \text{tt}$ and (3) $(\emptyset, \emptyset) \cap V : \text{Mult} = \text{tt}$, with induction hypotheses $\emptyset \cap V : \text{Mult} = \text{tt}$ and $\emptyset \cap V : \text{Mult} = \text{tt}$. We can now apply to goal (2) the Split rule (SP) with predicate term $x \in V$ to get subgoals:

$$\begin{align*}
(2.1) & \quad x \in V = \text{true} \rightarrow x \land V : \text{Mult} = \text{tt} \\
(2.2) & \quad x \in V = \text{false} \rightarrow x \land V : \text{Mult} = \text{tt}
\end{align*}$$

which can both be automatically discharged using the ICC simplification rule. Likewise, we can apply SP to goal (3) with predicate term $y \in V$ to get subgoals:

$$\begin{align*}
(3.1) & \quad y \in V = \text{true} \rightarrow (\emptyset, \emptyset) \cap V : \text{Mult} = \text{tt} \\
(3.2) & \quad y \in V = \text{false} \rightarrow (\emptyset, \emptyset) \cap V : \text{Mult} = \text{tt}
\end{align*}$$

which can both be automatically discharged using the ICC simplification rule. The reason why this is so is worth pointing out. The ICC rule can use the next-to-last conditional equation defining $\land$ to simplify the consequent of (3.1) to $\emptyset, (\emptyset) \cap (V \setminus \emptyset) : \text{Mult} = \text{tt}$; but then it can use the induction hypothesis $\emptyset \cap V : \text{Mult} = \text{tt}$ and the rules $X, Y : \text{Mult} \rightarrow \text{tt}$ if $X : \text{Mult} = \text{tt} \land X : \text{Mult} = \text{tt}$, $X : \text{Mult} \rightarrow \text{tt}$ if $X : \text{Nat} = \text{tt}$, and $x : \text{Nat} \rightarrow \text{tt}$ in $\vec{E} : \Delta$, where $x$ has sort $\text{Nat}$ and $X, Y$ have sort $\text{[Mult]}$, to further simplify the conclusion to $\text{tt} = \text{tt}$, thus discharging the goal. In the case of subgoal (3.2), the ICC rule just uses the last conditional equation defining $\land$ and then the induction hypothesis $\emptyset \cap V : \text{Mult} = \text{tt}$. This finishes the proof of the sufficient completeness of $\land$. 

116
6 Related Work and Conclusions

Research on sufficient completeness goes back to Guttag’s thesis in the 1970’s and includes, e.g., \[16, 33, 6, 20, 21, 2, 17, 19, 18, 1, 31, 30, 22, 32, 25\]. Four papers most closely related to this work, because all of them deal with order-sorted theories, are \[31\], \[1\], \[17\] and \[25\]. The work in \[31\] provides some useful methods for proving sufficient completeness of order-sorted CafeOBJ specifications and shares with this work the use of module hierarchies; however, the methods used in \[31\] do not seem to support rewriting modulo axioms. The work in \[1\] has several relevant similarities with the present work: (i) it supports conditional order-sorted theories; and (ii) it emphasizes that proofs of sufficient completeness and of ground confluence help each other. However, \[1\] does not support rewriting modulo axioms. The paper \[17\] did not support rewriting modulo axioms either; but it has two relevant similarities with the present work: (i) it could analyze specifications in membership equational logic \[28, 3\], which is also supported by Maude and is more general than order-sorted equational logic; and (ii) it used a previous version of the Maude Inductive Theorem prover to discharge verification conditions. In comparison with \[17\], the implicit similarity is that the $E \rightarrow E'$ transformation in this work endows $E$ with membership equational logic reasoning capabilities, while remaining within the simpler order-sorted framework. The relation with the companion paper \[25\] has already been discussed in the body of the paper: they complement each other.

In conclusion, I have presented a new hierarchical methodology to verify sufficient completeness by inductive theorem proving and have illustrated it with three examples. Since order-sorted specifications contain many-sorted and unsorted ones as special cases, the sufficient completeness proof methods presented here apply in particular to many-sorted and unsorted specifications in any equational language supporting them. In fact, Example 1 is many-sorted. It would be highly desirable to combine the inference systems for proving ground convergence in \[11\] with those in \[25\] and in \[11\] within a tool that would also support verification of sufficient completeness by the hierarchical methods in \[25\] and in this paper. Such a tool would use as a backend the new Maude Inductive Theorem Prover under construction — which supports an extension of the inference system in \[26\] illustrated in the examples of Section 5.

Acknowledgements. I cordially thank the anonymous referees for their excellent suggestions for improving the paper.

References


## A Multiset Theory

```ocaml
fmod MULTISET-VALUE is
  protecting TRUTH-VALUE .
sorts Nat NeMult Mult .
subsort Nat < NeMult < Mult .
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor] .
op mt : -> Mult [ctor] .
op _+_ : Mult Mult -> Mult [assoc comm] .
op _+_ : NeMult NeMult -> NeMult [ctor assoc comm] .
op _\in\_ : Nat Mult -> Bool .
op _\-\_ : Mult Mult -> Mult .
op _\&\_ : Mult Mult -> Mult .
vars n m k : Nat . vars U V W : Mult .
```

119
B Proofs

Proof of Theorem 1. We can extend the RPO order on function symbols making \( E \) operationally terminating modulo \( B \) by adding, for \((S,\prec)\) the sort poset in \( \tilde{E} \), the new ordered pairs \( :s \succ tt \) for each sort \( s \in S \), as well as the pairs \( :s \succ :s' \succ tt \) for each \( s,s' \in S \) such that \( s \succ s' \). It is easy to check that this order makes the new added rules in \( \tilde{E}_S \) operationally terminating. Note that this crucially depends on \( B \) containing only associativity and/or commutativity axioms by assumption, since any axioms for an identity element for a binary function symbol \( f \) could make the rule \( f:p:x_1,x_2:q:s \succ tt \) if \( x_1:s_1:tt \) and \( x_2:s_2:tt \) non-terminating. \( \square \)

Proof of Theorem 2. All terms in \( \tilde{E} \) can be rewritten with \( \rightarrow_{\tilde{E}} \) iff they can be rewritten with \( \rightarrow_{\tilde{E}} \), so we only need to prove ground convergence for ground terms of sort \( Pred \) in \( \tilde{E} \), which are either \( tt \), which is in canonical form, or ground terms of the form \( t:s \) for some sort \( s \in S \). Any rewrite sequence \( t:s \rightarrow_{\tilde{E}} w \) must be either of the form: (i) \( t:s \rightarrow_{E} t':s \), or (ii) of the form \( t:s \rightarrow_{E} t':s \rightarrow_{\tilde{E}_S:B} tt \). In particular, this applies to terminating sequences, so that all canonical forms of ground terms of sort \( Pred \) must be of the from (a) \( tt \), or of the form (b) \( t!_{\tilde{E}}:s \).
\(\vec{E}\): will be ground convergent if we can prove that any ground predicate term \(t:s\) has a unique canonical form. This follows from the following lemma, whose easy proof by structural induction on the term structure of \(t\) is left to the reader:

**Lemma 1.** For any ground term \(t\) in \(\vec{E}\), \(t:s \rightarrow_{\vec{E},\Sigma,B} tt\) iff \(ls(t) \leq s\).

Uniqueness (up to \(B\)-equality) of the canonical normal form of a ground predicate term \(t:s\) then follows easily from the fact that, since \(\vec{E}\) is ground convergent, then it is also sort-decreasing, so that for any rewrite sequence \(t:s \rightarrow^* \vec{E} t':s \rightarrow^* \vec{E} t\) we must have \(ls(t') \geq ls(t)\). Therefore, either (a) \(ls(t) \leq s\) and the canonical form of \(t:s\) is \(tt\), or (b) \(ls(t) \geq s\) and the canonical form of \(t:s\) must be \(t\) itself, so that the two equivalences stated in the theorem hold. \(\square\)

**Proof of Theorem 3.** For any \(f \in \Delta\), maximal typing \(f : s_1, \ldots, s_n \rightarrow s\) and ground constructor substitution \(\rho\), we need to show that the term \(f(x_1, \ldots, x_n)\rho\) is \(\vec{E}\)-irreducible. Since \(\vec{E} : \Delta\) is ground confluent and \(tt\) is irreducible, we then must have \(f(x_1, \ldots, x_n)\rho : s \rightarrow_{\vec{E}} tt\). But since \(f(x_1, \ldots, x_n)\rho\) is irreducible, this can only happen if \(f(x_1, \ldots, x_n)\rho : s \rightarrow_{\vec{E}} tt\). But since \(ls(f(x_1, \ldots, x_n)\rho) = [s]\) and \([s] > s\), this is impossible by Lemma 1. \(\square\)

**Proof of Corollary 1.** Since \(\vec{E}\) is assumed ground convergent, if a maximal typing in \(\Delta\), say, \(f : s_1, \ldots, s_n \rightarrow s\) is such that \(T_{\Sigma,B,s}\) is a finite set, say, \(T_{\Sigma,B,s} = \{[u_1], \ldots, [u_m]\}\), we need to show that if

\[
(\dagger) \quad T_{\vec{E},\Delta} = \bigvee_{1 \leq i \leq m} f(x_1, \ldots, x_n) = u_i,
\]

with \(x_j\) of sort \(s_j\), \(1 \leq j \leq n\), holds, then replacing the conjunction \(f(x_1, \ldots, x_n):s = tt\) by the disjunction \(\bigvee_{1 \leq i \leq m} f(x_1, \ldots, x_n) = u_i\) in the conjunction of Theorem 3, the theorem still holds. We reason by contradiction assuming that the resulting conjunction after this replacement is an inductive theorem of \(T_{\vec{E},\Delta}\) but Theorem 3 fails. In particular, all other conjunctions for the remaining maximal typings of \(\Delta\) are inductive theorems of \(T_{\vec{E},\Delta}\), so that, by the above proof of Theorem 3, all corresponding ground instances for those operators and their typings are reducible. Therefore, the theorem can only fail if there is a ground constructor substitution \(\rho\) such that the term \(f(x_1, \ldots, x_n)\rho\) is \(\vec{E}\)-irreducible. But since \((\dagger)\) holds, then by the ground convergence of \(\vec{E}\) there must be \([u_k] \in \{[u_1], \ldots, [u_m]\}\) such that \((f(x_1, \ldots, x_n)\rho)_{\vec{E},\Delta} = u_k\). But since \(f(x_1, \ldots, x_n)\rho\) is \(\vec{E}\)-irreducible, this is impossible. \(\square\)
On Ground Convergence and Completeness of Conditional Equational Program Hierarchies

José Meseguer and Stephen Skeirik

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

Abstract. Both complete definition of functions by equations and determinism (i.e., evaluation to a unique result), are fundamental correctness properties of equational programs. But for expressive functional languages supporting conditional equations, types and subtypes and rewriting modulo axioms, proof methods for verifying such properties under general conditions are currently quite limited. This work proposes a hierarchical proof methodology where both properties are simultaneously verified in a hierarchical manner under termination assumptions.

1 Introduction

Equational programs define functions by means of equations. Such programs may be faulty in various ways. A common problem is lack of complete definition, that is, for a specific concrete input the program cannot be evaluated to concrete data. This is called lack of sufficient completeness. Maude [7] color codes unevaluated function symbols in the result of an evaluation to alert the user when this happens. A more subtle problem is lack of determinism. For any input, a terminating equational program should evaluate to a unique result. Lack of determinism may not be detected at runtime, since a functional expression will evaluate to some result following a given evaluation strategy that assumes such determinism.

These two properties, completeness and determinism, are fundamental for program correctness and are assumed when proving many other properties, for example by inductive theorem proving. For expressive equational languages supporting conditional equations, types and subtypes and rewriting modulo axioms like associativity and/or commutativity and/or identity such as OBJ [19], CafeOBJ [15] and Maude [7], decision procedures ensuring these properties only exist for restricted program classes of unconditional, terminating programs not involving associative but noncommutative axioms. For them: (i) joinability of critical pairs (see, e.g., [9]) ensures determinism; and (ii) equational tree automata methods can check sufficient completeness if lefthand sides of equational definitions are left-linear [25]. Even under those restrictions, checking (i) is only a sufficient condition for determinism. This is because only a weaker condition, namely, ground joinability of critical pairs is needed, since an equational program only executes concrete inputs, i.e., ground terms. There are many perfectly fine
equational programs whose critical pairs are ground joinable but not joinable. For a good example, see the equational definition of finitary set theory in [12]. Alas, ground joinability of critical pairs is an undecidable property [28]. Checking completeness and determinism of function definitions becomes much harder for conditional specifications, where both checks are undecidable. But this does not make the need to ensure these fundamental properties any less pressing.

Ground joinability of critical pairs is called ground confluence, and, under the assumption of operational termination, ground convergence. As further discussed in Section 4, a lot of work has been done on proof methods for both sufficient completeness and ground convergence of such programs. For ground convergence it further advances the ideas in [12]. Two key features of our proof methodology are that: (i) sufficient completeness and ground convergence proofs help and depend on each other; and (ii) the proof methods are hierarchical and therefore incremental: proofs are obtained by climbing up a tower of theory inclusions.

2 Preliminaries

We assume familiarity with the notions of an order-sorted signature $\Sigma$ on a poset of sorts $(S, \leq)$, an order-sorted $\Sigma$-algebra $A$, and the term $\Sigma$-algebras $T_{\Sigma}$ and $T_{\Sigma}(X)$ for $X$ an $S$-sorted set of variables. We also assume familiarity with the notions of: (i) $\Sigma$-homomorphism $h : A \rightarrow B$ between $\Sigma$-algebras $A$ and $B$, so that $\Sigma$-algebras and $\Sigma$-homomorphisms form a category $\text{OSAlg}_{\Sigma}$; (ii) order-sorted (i.e., sort-preserving) substitution $\theta$, its domain $\text{dom}(\theta)$ and range $\text{ran}(\theta)$, and its application $\theta t$ to a term $t$; (iii) preregular order-sorted signature $\Sigma$, i.e., a signature such that each term $t$ has a least sort, denoted $\text{ls}(t)$; (iv) the set $\hat{S} = S/(\geq \cup \leq)^+$ of connected components of a poset $(S, \leq)$ viewed as a DAG; and (v) for $A$ a $\Sigma$-algebra, the set $A_s$ of its elements of sort $s \in S$, and the set $A_{[s]} = \bigcup_{s \in [s]} A_s$ of all elements in a connected component $[s] \in \hat{S}$. We furthermore assume that all signatures $\Sigma$ have non-empty sorts, i.e., $T_{\Sigma,s} \neq \emptyset$ for each $s \in S, [A \rightarrow B]$ denotes the $S$-sorted functions from $A$ to $B$. These notions are explained in [36, 18]. The material below is adapted from [37, 34].

**Order-Sorted Algebra and $E$-Unification.** An OS equational theory is a pair $T = (\Sigma, E)$, with $E$ a set of (possibly conditional) $\Sigma$-equations. $\text{OSAlg}_{(\Sigma, E)}$ denotes the full subcategory of $\text{OSAlg}_\Sigma$ with objects those $A \in \text{OSAlg}_\Sigma$ such that $A \models E$, called the $(\Sigma, E)$-algebras. $\text{OSAlg}_{(\Sigma, E)}$ has an initial algebra $T_{\Sigma|E}$ [36]. If $E = (\Sigma, E)$, $T_E$ abbreviates $T_{\Sigma|E}$. For $\Sigma$ an OS-signature, $\text{Form}(\Sigma)$ denotes the set of its first-order formulas, whose atoms are $\Sigma$-equations. Given $T = (\Sigma, E)$ and $\varphi \in \text{Form}(\Sigma)$, we call $\varphi$ $T$-valid, written $E \models \varphi$, iff $A \models \varphi$ for all $A \in \text{OSAlg}_{(\Sigma, E)}$. We call $\varphi$ $T$-satisfiable iff there exists $A \in \text{OSAlg}_{(\Sigma, E)}$ with $\varphi$ satisfiable in $A$; that is, there exists an assignment $a$, i.e., and $S$-sorted function
a ∈ [fv(ϕ) → A] with fv(ϕ) the free variables of ϕ, such that A, a |= ϕ. Note that ϕ is T-valid iff ¬ϕ is T-unsatisfiable. The inference system in [36] is sound and complete for OS equational deduction, i.e., for any OS equational theory (Σ, E), and Σ-equation u = v we have an equivalence E |= u = v ⇔ E |= u = v. Deducibility E |= u = v is abbreviated as u =E v, called E-equality. An E-unifier of a system of Σ-equations, i.e., of a conjunction ϕ = u₁ = v₁ ∧ . . . ∧ uᵦ = vᵦ of Σ-equations, is a substitution σ such that uσ =E vᵦσ, 1 ≤ i ≤ n. An E-unification algorithm for (Σ, E) is an algorithm generating a complete set of E-unifiers Unif_E(ϕ) for any system of Σ equations ϕ, where “complete” means that for any E-unifier σ of ϕ there is a τ ∈ Unif_E(ϕ) and a substitution ρ such that σ =E (τρ)|dom(τ)\dom(σ), where =E here means that for any variable x we have xσ =E x(τρ)|dom(σ)\dom(τ). The algorithm is finitary if it always terminates with a finite set Unif_E(ϕ) for any ϕ. Given a set of equations B used for deduction modulo B, a preregular OS signature Σ is called B-preregular if for each u = v ∈ B and substitutions ρ, ls(up) = ls(vp).

Convergent Theories and Sufficient Completeness. Given an order-sorted equational theory E = (Σ, E ∪ B), where B is a collection of associativity and/or commutativity and/or identity axioms and Σ is B-preregular, we can associate to it a corresponding rewrite theory [35] E = (Σ, B, E) by orienting the equations E as left-to-right rewrite rules. That is, each (u = v) ∈ E is transformed into a rewrite rule u → v. For simplicity we recall here the case of unconditional equations. Since in this work we will consider conditional theories E, we refer to [32] for full details on the general definition of convergent theory. The main purpose of the rewrite theory E is to reduce the complex bidirectional reasoning with equations to the much simpler unidirectional reasoning with rules under suitable assumptions. We assume familiarity with the notion of subterm t|p of t at a term position p and of term replacement t[wp] of t by w at position p (see, e.g., [9]). The rewrite relation t →*E,B t' holds iff there is a subterm t|p of t, a rule (u → v) ∈ E and a substitution σ such that uσ =E t[σ|p], and t' = t[σ|p]. We denote by →*E,B the reflexive-transitive closure of →*E,B. For E unconditional, the convergence requirements are as follows (see [32] for E conditional): (i) vars(v) ⊆ vars(u); (ii) sort-decreasingness: for each substitution θ, ls(uθ) ≥ ls(vθ); (iii) strict B-coherence: if t₁ →*E,B t'₁ and t₁ =B t₂ then there exists t₂ →*E,B t'₂ with t'₁ =B t'₂; (iv) confluence (resp. ground confluence) modulo B: for each term t (resp. ground term t) if t →*E,B v₁ and t →*E,B v₂, then there exist rewrite sequences v₁ →*E,B w₁ and v₂ →*E,B w₂ such that u₁ =B w₁; (v) termination: the relation →*E,B is well-founded (for E conditional, we require operational termination [32]). If E satisfies conditions (i)–(v) (resp. the same, but

---

1 If B = B₀ w U, with B₀ associativity and/or commutativity axioms, and U identity axioms, the B-preregularity notion can be broadened by requiring only that: (i) Σ is B₀-preregular in the standard sense that ls(up) = ls(vp) for all u = v ∈ B₀ and substitutions ρ; and (ii) the axioms U oriented as rules U are sort-decreasing in the sense explained below.
(iv) weakened to ground confluence modulo $B$), then it is called convergent (resp. ground convergent). The key point is that then, given a term (resp. ground term) $t$, all terminating rewrite sequences $t \rightarrow^*_E w$ end in a term $w$, denoted $t_E$, that is unique up to $B$-equality, and its called $t$’s canonical form. Ground convergence implies three major results: (1) for any ground terms $t, t'$ we have $t =_{E,B} t'$ iff $t_E = B t'_E$, (2) the $B$-equivalence classes of canonical forms are the elements of the canonical term algebra $C_{\Sigma,B}$, where for each $f : s_1 \ldots s_n \rightarrow s$ in $\Sigma$ and $B$-equivalence classes of canonical terms $[t_1], \ldots, [t_n]$ with $ts(t_i \in s_i$, the operation $f_{C_{\Sigma,B}}$ is defined by the identity: $f_{C_{\Sigma,B}}([t_1] \ldots [t_n]) = [f(t_1 \ldots t_n)]_E$, and (3) we have an isomorphism $T_E \cong C_{\Sigma,B}$.

A ground convergent rewrite theory $\tilde{\mathcal{E}} = (\Sigma, B, \tilde{E})$ is called sufficiently complete with respect to a subsignature $\Omega$, whose operators are then called constructors, iff for each ground $\Sigma$-term $t$, $t_{\tilde{E}} \in T$. Furthermore, for $\tilde{\mathcal{E}} = (\Sigma, B, \tilde{E})$ sufficiently complete w.r.t. $\Omega$, a ground convergent rewrite subtheory $(\Omega, B_\Omega, \tilde{E}_\Omega) \subseteq (\Sigma, B, \tilde{E})$ is called a constructor subspecification iff $T_{\tilde{E}}|_{B} \cong T_{\Omega|B, B_\Omega}$. If $E_\Omega = \emptyset$, then $\tilde{\mathcal{E}}$ is called a signature of free constructors modulo axioms $B_\Omega$. Note that $\tilde{\mathcal{E}} = (\Sigma, B, \tilde{E})$ is sufficiently complete with respect to $\Omega$ iff each ground $\Sigma$-term $f(u_1, \ldots, u_n)$ with $f \in \Sigma \setminus \Omega$ and $u_i \in T$, $1 \leq i \leq n$, is $\tilde{E}, B$-reducible, i.e., $f(u_1, \ldots, u_n) \rightarrow_{\tilde{E}, B} t$ for some $t \in T_{\Sigma}$.

**Generator Sets.** Generator sets generalize standard structural induction on the constructors of a sort. They are particularly useful for inductive reasoning when constructors obey structural axioms $B$ including associativity and/or associativity-commutativity for which structural induction may be ill-suited.

A generator set for a sort $s$ is a set of constructor terms of sort $s$ or smaller such that any ground constructor term of sort $s$ is a ground substitution instance of one of the patterns in the generator set. Here is the general definition (identity axioms are not needed thanks to the theory transformation $\tilde{\mathcal{E}}_U \rightarrow \tilde{\mathcal{E}}_U$ in [10]):

**Definition 1.** For an order-sorted signature of constructors $\Omega$ —which may satisfy axioms $B$ of associativity and/or commutativity— and a sort in $\Omega$, a $B$-generator set for sort $s$ is a finite set of terms $\{u_1, \ldots, u_k\} \subseteq T(\Omega)^s$ such that

$$T_{\Omega|B,s} = \bigcup_{1 \leq i \leq k} \{[u_i, \rho] \in T_{\Omega|B,s} \mid \rho \in [X \rightarrow T]\}.$$

**Checking the Correctness of Generator Sets.** How do we know that a proposed generator set $\{u_1, \ldots, u_k\}$ it truly one modulo axioms $B$ for a given sort $s$ and constructors $\Omega$? Assuming that the terms $u_1, \ldots, u_k$ are all linear, i.e., have no repeated variables —which is the usual case for generator sets— this check can be reduced to an automatic sufficient completeness check with Maude’s Sufficient Completeness Checker (SCC) tool [25], which is based on tree automata decision procedures modulo axioms $B$. The reduction is extremely simple: define a new unary predicate $s : s \rightarrow \text{Bool}$ with equations $s(u_i) = \text{true}$, $1 \leq i \leq k$. Then, $\{u_1, \ldots, u_k\}$ is a correct generator set for sort $s$ modulo $B$ for
the constructor signature \( \Omega \) iff the predicate \( s \) is sufficiently complete, which can be automatically checked by the SCC tool. Furthermore, if \( \{ u_1, \ldots, u_k \} \) is not a generator set for sort \( s \), the SCC tool will output a useful counterexample.

3 Proving Ground Convergence and Sufficient Completeness Hierarchically

Basic Assumptions about \( \vec{E} \). We assume throughout a conditional equational theory \( \mathcal{E} = (\Sigma, \vec{E} \cup U \cup B) \) such that: (i) \( \Sigma \) decomposes as a disjoint union \( \Sigma = \Delta \cup \Omega \), where \( \Omega \) are the intended but not yet proved to be constructor symbols, that furthermore are free modulo \( B \), and \( \Delta \) are the intended defined symbols. (ii) \( B \) is any combination of associativity and/or commutativity axioms, but any binary \( f \in \Delta \) may not satisfy any axioms except commutativity.\(^2\) (iii) \( \vec{U} \) are sort-decreasing unit axiom rules of the form \( c(e, x) \rightarrow x \) or \( c(x, e) \rightarrow x \), where \( c \) is a constructor name and \( e \) is an \( \Omega \)-term. However, \( c(x, e) \) is not an \( \Omega \)-term because some of \( c \)'s type declarations do not belong to \( \Omega \) (see Example 1). This makes it possible for constructors to be free modulo \( B \) in spite of such unit rules. (iv) \( \mathcal{E} = \bigcup_{f \in \Delta} \vec{E}_f \), where for each \( f \in \Delta \), its associated rewrite rules \( \vec{E}_f \) are sort-decreasing and have the form: \( \vec{E}_f = \{ [i] : f(\bar{u}_i) \rightarrow r_i \text{ if } \Gamma_i \} \subseteq I \) such that: (a) the \( \bar{u}_i \) are \( \Omega \)-terms; and (b) for each \( i \in I \), \( \Gamma_i = \bigwedge_{j \in J} w_j = w_j' \) and \( \text{vars}(f(\bar{u}_i)) \supseteq \text{vars}(r_i) \cup \text{vars}(\Gamma_i) \). (v) There is a \( B \)-compatible recursive path order (RPO) \( > \) (see [9]) such that for each \( i \in I \), \( f(\bar{u}_i) > r_i \), and \( f(\bar{u}_i) > w_j, w_j' \).

\(^2\) Furthermore, for any \( f \) that is commutative we always assume a top typing \( f : s s \rightarrow s_0 \) with all other typings of the form \( f : s' s' \rightarrow s'_0 \), with \( s \leq s' \), \( s_0 \leq s'_0 \). Regarding the absence of unit element axioms, they are precisely the equations \( \vec{U} \) that will be used as rules \( \vec{U} \) (see, e.g., Example 1). The point is that, for both confluence and termination purposes, if \( \vec{G} \) has axioms \( B \cup U \), with \( B \) associative and/or commutative axioms and \( U \) unit element axioms, then the axioms \( U \) can be eliminated by turning them into rules \( \vec{U} \) thanks to the semantics-preserving theory transformation \( \vec{G} \rightarrow \vec{G}_U \) defined in [10], so that the axioms of the semantically equivalent \( \vec{G}_U \) are just \( B \). Therefore, Our results apply as well to theories \( \vec{G} \) with axioms \( B \cup U \) such that \( \vec{G}_U \) has the properties (i)-(vi) listed in what follows.

\(^3\) Since axioms \( B \) are primarily used to specify constructor data structures, in actual practice, limiting axioms for defined symbols to just commutativity is a mild restriction. Furthermore, this restriction can be removed \textit{a posteriori} in the following sense. After \( \vec{E} \) has been shown ground convergent and sufficiently complete, if we can prove by inductive theorem proving that the initial algebra \( T_E \) does satisfy additional associativity and/or commutativity axioms for some binary \( f \in \Delta \), then we can add to \( \vec{E} \): (a) those extra axioms for \( f \), and (b) the \( A \)-res. \( \text{AC-extensions} \) (see [40]) of the rules \( \vec{E}_f \) in the sense of (iv) below (to ensure \( \text{B-coherence} \)). One can then show that the theory thus extended is also ground convergent and sufficiently complete if its rules remain operationally terminating modulo the extended axioms. For example, in the \textsc{Multiset-Algebra} module of Example 1, we can prove the associativity and commutativity of the intersection operator \( \cap \) as inductive theorems and then add those properties as axioms of \( \cap \) (the \( \text{AC-extensions} \) of \( \vec{E}_c \) do not need to be added explicitly: they are added automatically by Maude).
\[ j \in J, \text{ which makes the rules } \vec{E} \cup \vec{U} \text{ operationally terminating modulo } B. \] (vi) The rules \( \vec{E} \cup \vec{U} \) are strictly \( B \)-coherent.

The main goal of this paper is to develop hierarchical methods to prove that a theory \( \vec{E} \) enjoying properties (i)-(vi) above is ground convergent. As it turns out, such hierarchical methods will also allow us to prove that \( \vec{E} \) is sufficiently complete with respect to its hypothesized constructor signature \( \Omega \). As we shall see, hierarchical proofs of ground convergence and of sufficient completeness will help each other. Since the rules in \( \vec{E} \) can be conditional, they are generally outside the scope of equational tree-automata methods for checking sufficient completeness supported by tools like Maude’s Sufficient Completeness Checker (SCC) [25], which assume unconditional and left-linear rules: new proof methods are needed. Sufficient completeness is important both for program correctness and because it allows constructor-based inductive reasoning.

**Calling Graph and Theory Hierarchies.** We assume that all function symbols in \( \Delta \) are **subsort-overloaded**, i.e., for any \( f: s_1 \ldots s_n \rightarrow s \) and \( f: s'_1 \ldots s'_n \rightarrow s' \) we have \([s] = [s']\), and \([s_i] = [s'_i]\), \( 1 \leq i \leq n \). This can always be achieved by renaming \( \Delta \). Let \( F_\Delta \) be the set of **names** for the function symbols in \( \Delta \), disregarding their typing. The **calling relation** is a binary relation \( C \) on \( F_\Delta \), where for each \( f, g \in F_\Delta \), \((f, g) \in C\) iff there exists a rule \( f(\vec{u}) \rightarrow r_1 \) if \( \Gamma_i \) in \( \vec{E}_1 \) such that the function symbol \( g \) occurs in either \( r_1 \) or in \( \Gamma_i \). Let \( C^* \) denote the reflexive-transitive closure of \( C \), and \( =_C \) the equivalence relation on \( F_\Delta \) defined by the equivalence: \( f =_C g \) iff \( f C^* g \) and \( g C^* f \). Then, the quotient set \( F_\Delta / \equiv_C \) has a partial order defined by the equivalence \([f] \geq [g] \leftrightarrow f C^* g\).

The hierarchical method we propose is based on a **hierarchy of theory inclusions** chosen as follows. Given our theory \( \vec{E} \) we: (i) identify a subtheory \( \vec{E}_0 \) having subsignature \( \Delta_0 \cup \Omega \) containing all the subsort-overloaded typings of any \( f \in \Delta_0 \) and having rules \( \vec{U} \cup \vec{E}_0 \), with \( \vec{E}_0 = \bigcup_{f \in \Delta_0} \vec{E}_f \), and axioms \( B_0 = B_{\Delta_0 \cup \Omega} = \bigcup_{f \in \Delta_0 \cup \Omega} B_f \), where \( B_f \) are the associative and/or commutative axioms, if any, for \( f \in B \), and such that \( \vec{E}_0 \cup \vec{U} \) is sufficiently complete with respect to \( \Omega \) and ground convergent. Of course, we should choose \( \vec{E}_0 \) as big as possible; in the worse case we may have \( \vec{E}_0 = \emptyset \) and keep only \( \vec{U} \). We furthermore assume that we can find a sequence of theory inclusions:

\[
\vec{E}_0 \subset \vec{E}_1 \subset \ldots \vec{E}_{n-1} \subset \vec{E}_n
\]

such that: (a) \( \vec{E}_n = \vec{E} \), (b) each \( \vec{E}_k \) has signature \( \Delta_k \cup \Omega \) containing all subsort-overloaded typings of any \( f \in \Delta_k \) and having rules \( \vec{U} \cup \vec{E}_k \), with \( \vec{E}_k = \bigcup_{f \in \Delta_k} \vec{E}_f \), and, besides \( \vec{U} \), rules \( \vec{E}_k = \bigcup_{f \in \Delta_k} \vec{E}_f \), where for each \( k \geq 1 \) and each rule \( \{i\} : f(\vec{u}) \rightarrow r_i \) if \( \Gamma_i \) in \( \vec{E}_j \) the condition \( \Gamma_i \) is a conjunction of \( \vec{E}_{k-1} \)-equalities, (c) for each \( 0 \leq k < k + 1 \leq n \), there exists a function symbol \( g \in F_{\Delta_{k+1}} \setminus F_{\Delta_k} \) such that \( F_{\Delta_{k+1}} = F_{\Delta_k} \cup [g] \); that is, we add all the symbols in a new \( \equiv_C \)-equivalence class \([g]\) to climb up each step in the theory hierarchy. Let us see an example.

**Example 1.** (Multisets of Natural Numbers). As a running example we use a theory of **multisets of natural numbers** with number equality \( _= \) and multiset
membership \_ \in \_ predicates, multiset difference \setminus, intersection \setcap, and union \_ \cup \_. Its Maude specification (with self-explanatory syntax for \( (\Sigma, E \cup U \cup B) \)) is:

\[
\text{fmod MULTISET-ALGEBRA is}
\]

\[
\begin{align*}
& \text{protection TRUTH-VALUE} . \\
& \text{sorts Nat Mult} . \\
& \text{subsort Nat < NeMult < Mult} . \\
& \text{op 0 : } \to \text{Nat [ctor]} . \\
& \text{op s : Nat } \to \text{Nat [ctor]} . \\
& \text{op mt : } \to \text{Mult [ctor]} . \quad \text{[ctor]} \quad \text{[ctor]} \quad \text{[ctor]} \\
& \text{op _,_ : Mult Mult } \to \text{Mult [assoc comm] } . \\
& \text{op _,_ : NeMult NeMult } \to \text{NeMult [ctor assoc comm] } . \\
& \text{op _,._ : Nat Nat } \to \text{Bool [comm] } . \\
& \text{op _,in_ : Nat Mult } \to \text{Bool } . \\
& \text{op _\setminus_ : Mult Mult } \to \text{Mult } . \\
& \text{op _\setcap_ : Mult Mult } \to \text{Mult } . \\

\end{align*}
\]

\[
\begin{align*}
& \text{vars n m k : Nat} . \\
& \text{vars U V W : Mult} . \\
& \text{eq U,mt = U} . \\
& \text{eq n .=. n = true} . \\
& \text{eq 0 .=. s(n) = false} . \\
& \text{ceq s(n) .=. s(m) = false if n .=. m = false} . \\
& \text{eq n in mt = false} . \\
& \text{eq n in n = true} . \\
& \text{ceq n in m = false if (n .=. m) = false} . \\
& \text{eq n in (n,U) = true} . \\
& \text{ceq n in (m,U) = false} \\
& \quad \text{if (n .=. m) = false } \setcap \text{ (n in U) = false} . \\
& \text{eq mt \setminus U = mt} . \\
& \text{eq U \setminus mt = U} . \\
& \text{eq m \setminus m = mt} . \\
& \text{ceq m \setminus n = m if n .=. m = false} . \\
& \text{eq (m,U) \setminus n = U} . \\
& \text{ceq (m,U) \setminus n = m,(U \setminus n) if n .=. m = false} . \\
& \text{eq U \setminus (n,V) = (U \setminus n) \setminus V} . \\
& \text{eq mt \setcap V = mt} . \\
& \text{ceq n \setcap V = n if (n in V) = true} . \\
& \text{ceq n \setcap V = mt if (n in V) = false} . \\
& \text{ceq (n,U) \setcap V = n,(U \setcap (V \setminus n)) if (n in V) = true} . \\
& \text{ceq (n,U) \setcap V = U \setcap V if (n in V) = false} . \\
\end{align*}
\]

endfm

Its calling graph is described in Figure 1.

Let \( \hat{E} \) denote the above theory of multisets of natural numbers, with the defined operations \( \_ = \_ \), \( \_ \in \_ \), \( \_ \setminus \_ \), \( \_ \setcap \_ \), and \( \_ \cup \_ \) and the usual Boolean operations. Its

128
constructors \( \Omega \) are: 0, \( s \), \( J \), and \( K \), \( H \), and the union operator \( \cup \) for non-empty multisets. \( \mathcal{E} \) satisfies properties (i)–(vi). Furthermore, we have a tower of theories:

\[
\mathcal{E}_{s} \subset \mathcal{E}_{s,\cup} \subset \mathcal{E}
\]

where each theory contains the Boolean values, plus the constructors, plus the mentioned operations, plus the rules defining such operations. \( \mathcal{E}_s \) is both convergent and sufficiently complete, so we just have two steps to climb up the tower to prove \( \mathcal{E} \) both ground convergent and sufficiently complete.

The Hierarchical Proof Method. All now boils down to finding proof methods to climb up the theory hierarchy one step at a time. We then repeat this method \( n \) times, with \( n \) the length of the chain of theory inclusions. That is, we focus on a single theory inclusion \( \mathcal{E}_0 \subset \mathcal{E} \), where \( \mathcal{E}_0 \) has already been proved ground convergent and sufficiently complete with respect to the constructor signature \( \Omega \), and then prove that \( \mathcal{E} \) is also ground convergent and sufficiently complete as follows. First of all, we define a new theory \( \mathcal{E}_\Delta \), with the same rules \( \mathcal{E} \cup \mathcal{U} \) as in \( \mathcal{E} \), and having also a theory inclusion \( \mathcal{E}_0 \subset \mathcal{E}_\Delta \), but where, if \( \Sigma_0 \) and \( \Sigma \) are the respective signatures of \( \mathcal{E}_0 \) and \( \mathcal{E} \), and \( \Delta = \Sigma \setminus \Sigma_0 \), then \( \mathcal{E}_\Delta \) has a signature \( \Sigma_\Delta \) that extends \( \Sigma_0 \) by: (i) adding to each connected component \([s] \in \hat{S}\) of the poset of sorts \((\hat{S}, \leq)\) of \( \Sigma_0 \) \([s]\) itself as a new “kind” top sort, i.e., \( \forall s' \in [s], s' < [s] \), and (ii) lifting to the kind levels the \( f \in \Sigma \). That is, we extend \( \Sigma_0 \) by adding to \( \Sigma_\Delta \) a function symbol \( f : [s_1] \ldots [s_n] \rightarrow [s] \) for each \( f : s_1 \ldots s_n \rightarrow s, n \geq 1, \) in \( \Sigma \). In \( \mathcal{E}_\Delta \) the axioms \( B \) are lifted to kinds. For example, for the theory inclusion \( \mathcal{E}_{s,\cup} \subset \mathcal{E} \) in our running example, \( \Sigma_0, \Sigma \) and \( \Sigma_\Delta \) can be depicted as follows (subsort inclusions in vertical lines):
Note that $\Sigma^\Delta$ adds no new terms to the original sorts $S$, i.e., $T_{\Sigma^\Delta}(X)_s = T_{\Sigma_0}(X)_s, s \in S$. The hierarchical proof methodology then proceeds as follows:

1. We first prove that $E^\Delta$ is ground convergent.

2. We then prove that for any $f \in \Delta$ typing $f : s_1, \ldots, s_n \rightarrow s$ maximal in the subsort order, and ground constructor substitution $\rho$, $f(x_1, \ldots, x_n)\rho$, with $x_i$ of sort $s_i, 1 \leq i \leq n$, can be rewritten with some rule in $E^\Delta$.

3. By Theorem 1, (1) and (2) actually prove that $E$ is both ground convergent and sufficiently complete with respect to $\Omega$.

All we now need to do is to give a sound inference system that will allow us to carry out the proofs for (1) and (2), and then prove Theorem 1. The inference system for proving (1) and (2) works in the context of the theory inclusion $E_0 \subset E^\Delta$, where $E_0$ has equations $U \cup E_0 \cup B_0$. Properties are specified as constrained properties of the form $p \mid \varphi$, where $p$ is a property and $\varphi$ is a conjunction of $\Sigma_0$-equations. Semantically, $p \mid \varphi$ describes the set $[p \mid \varphi]$ of ground constructor instances of property $p$ that satisfy $\varphi$. More precisely:

$$[p \mid \varphi] = \{p\theta \mid \theta \in [X \rightarrow T_\Omega] \land E_0 \vdash \varphi\theta\}.$$

The hierarchical inference system uses two kinds of constrained properties:
1. For proving that all instances of a term \( f(x_1, \ldots, x_n) \) by a ground substitution \( \rho \) are \( \vec{E}_f, B \)-reducible, where \( f \in \Delta \), we use constrained properties of the form \( \text{red}(f(u_1, \ldots, u_n)) \mid \varphi \), where the \( u_i \) are \( \Omega \)-terms. By definition, \( \text{red}(f(u_1, \ldots, u_n)) \mid \varphi \) holds iff for each ground constructor substitution \( \rho \) such that \( \varphi \rho \) holds in \( \mathcal{E}_0 \), the term \( f(u_1, \ldots, u_n)\rho \) is \( \vec{E}_f, B \)-reducible.

2. For proving the ground joinability of a conditional critical pair\(^4\) (CCP) of the form: \( \varphi \Rightarrow t = t' \) for the theory \( \vec{E}^\Delta \), we represent this property as the constrained property \( t \downarrow t' \mid \varphi \) and think of \( \downarrow \) as a binary predicate. We always assume that sorts of all variables in \( t \downarrow t' \mid \varphi \) are in \( S \) and \( \varphi \) is a conjunction of \( \mathcal{E}_0 \)-equalities, which will always be the case for any CCP of \( \vec{E}^\Delta \).

By definition, \( t \downarrow t' \mid \varphi \) holds iff for each ground constructor substitution\(^5\) \( \rho \) such that \( \mathcal{E}_0 \vdash \varphi \rho \) there exist \( u, v \) such that \( t\rho \rightarrow^* u =_B v \Rightarrow t'\rho \).

---

**The Shared Hierarchical Inference System.** We first introduce the inference rules applicable to both joinability and reducibility goals, i.e., goals either of the form (i) \( t \downarrow t' \mid \varphi \), with \( t, t' \Sigma^\Delta \)-terms; or of the form (ii) \( \text{red}(f(u_1, \ldots, u_n)) \mid \varphi \), with \( f \in \Delta \) and the \( u_i \) \( \Omega \)-terms. The key feature shared by both kinds of goals is that their constraint \( \varphi \) is a conjunction of \( \Sigma_0 \)-equations. By assumption, \( \mathcal{E}_0 \) is ground convergent, sufficiently complete with respect to \( \Omega \), and the constructors \( \Omega \) are free modulo \( B_0 \), with \( B_0 \) associative and/or commutative axioms.

In the shared inference system we assume constrained terms \( p \mid \varphi \) of the form (i) or (ii). The shared inference rules are the following:

**Narrowing the Condition (NA)**

\[
\frac{\{ (p \mid \Gamma_i \land \varphi[r_i]p)_{i \in I, r_i \in \text{Unif}_B(f(\vec{v}) = f(\vec{u}_i))} \}}{p \mid \varphi[f(\vec{v})]p}
\]

where the \( \vec{v} \) are \( \Omega \)-terms, \( f \in \Sigma_0 \setminus \Omega \) is defined by rules \( \{ [i] : f(\vec{u}_i) \rightarrow r_i \mid \Gamma_i \}_{i \in I} \) whose variables are always assumed disjoint of those in \( p \mid \varphi \), and where \( \text{Unif}_B(f(\vec{v}) = f(\vec{u}_i)) \) denotes the set of \( B \)-unifiers of the equation \( f(\vec{v}) = f(\vec{u}_i) \).

**Unification (UN)**

\[
\frac{\{ (p \mid \varphi[\psi])_{\theta \in \text{Unif}_U \cup \Sigma_0 \cup B_0(\psi)} \}}{p \mid \varphi \land \psi}
\]

where \( \land \) is assumed \( AC \), and \( \psi \) is a set of \( \Sigma_0 \)-equations such that \( \psi \) has a finite set of most general \( U \cup \Sigma_0 \cup B_0 \)-unifiers. This is guaranteed to happen if either \( \psi \) is a conjunction of \( \Omega \)-equations or, more generally, a conjunction of equations in a protected subtheory of \( \mathcal{E}_0 \) that has the finite variant property [13].

---

\(^4\) For a detailed definition of CCPs in an order-sorted setting see [11].

\(^5\) The ground joinability of the CCP \( \varphi \Rightarrow t = t' \) is normally stated as the joinability \( t_\alpha \downarrow t'_\alpha \) for all ground substitution \( \alpha \) such that \( \mathcal{E}_0 \vdash \varphi \alpha \). However, since, by ground convergence and sufficient completeness of \( \mathcal{E}_0 \) and the sort of all variables being in \( S \), any such \( \alpha \) can be normalized to a ground constructor substitution \( \alpha' \mathcal{E}_0 \), it can easily be shown that the CCP is ground joinable iff the property \( t \downarrow t' \mid \varphi \) holds.
Equality Simplification (ES)

\[ p \mid \varphi \vdash \bar{\varphi}_{\bar{E}_0} \]

\[ p \mid \varphi \]

where \( \varphi \vdash \bar{\varphi}_{\bar{E}_0} \) denotes the canonical form of \( \varphi \) in the ground convergent theory \( \bar{E}_0 \) extending \( E_0 \) with equality predicates defined in [21], which allows formula simplifications such as, e.g., \((T = \bot \land \psi) \vdash \bar{\varphi}_{\bar{E}_0} = \bot\), and \((0 = s(u) \land \psi) \vdash \bar{\varphi}_{\bar{E}_0} = \bot\).

Case (CA)

\[ \{(p \mid \varphi)\{x \mapsto v_i\}\}_{i \in I} \]

\[ p \mid \varphi \]

with \( x \in \text{vars}(p) \) a variable of sort \( s \) and \( \{v_i\}_{i \in I} \) a generator set for sort \( s \) in \( \Omega \), where all variables in \( \{v_i\}_{i \in I} \) are assumed fresh.

Split (SP)

\[ \{p \mid \varphi \land \psi_i\}_{i \in I} \]

\[ p \mid \varphi \]

where \( T_{\Sigma_{\psi} / E_0 \cup B_0} \models \bigwedge_{i \in I} \psi_i \) and \( \text{vars}(\bigwedge_{i \in I} \psi_i) \subseteq \text{vars}(u \mid \varphi) \).

Generalization (GN)

\[ p' \mid \psi \]

\[ p \mid \varphi \]

where \( 3\theta \ p' \theta = B \ p, \) and \( T_{\Sigma_{\psi} / E_0 \cup U \cup B_0} \models \varphi \Rightarrow (\psi \theta) \).

Empty Goal (\( \bot \))

\[ p \mid \bot \]

Ground Joinability Inference System. The goals to be proved are those associated to the conditional critical pairs in \( \bar{E} \Delta \) that are not in \( \bar{E}_0 \). The additional inference rules include: (i) constrained versions of the ground joinability rules in [12] (not needed for our running example); and (ii) the following two rules:

Join (JN)

\[ u \downarrow v \mid \varphi \quad \text{if} \quad u =_B v \]

Contextual Rewriting (CR)

\[ w \downarrow w' \mid \varphi \quad \pi \rightarrow^* \bar{\varphi} \quad u \downarrow v \mid \varphi \]

where (i) \( \bar{t} \) denotes the ground term obtained by replacing the variables in \( t \) by corresponding fresh constants, (ii) \( \bar{\varphi} \) is the rewrite condition associated to \( \varphi \) as explained in [11] (for example a condition \( x \in U = \top \land x \ast y = 0 \) yields the rewrite condition \( x \in U \rightarrow \top \land x \ast y \rightarrow 0 \), and (ii) \( \bar{\varphi} \) denotes the set of ground rewrite rules \( \pi \rightarrow \bar{\pi} \) such that \( u_1 \rightarrow u_2 \) is a conjunct in \( \bar{\varphi} \).
Example 2. For the theory inclusion $\vec{E}_{\Delta} \subseteq \vec{E}$ in our running example, the ground joinability goals associated to the theory $\vec{E}_{\Delta}$ are:

$$y, (x \cap (V \setminus y)) \downarrow x, (y \cap (V \setminus x)) \mid y \in V = \top \land x \in V = \top$$

$$y, ((x, W) \cap (V \setminus y)) \downarrow x, ((y, W) \cap (V \setminus x)) \mid y \in V = \top \land x \in V = \top$$

corresponding to the only two CCPs not already proved joinable for $\vec{E}_0$ that cannot be proved joinable by Maude’s Church-Rosser Checker tool. The proof of the first goal is shown below; we leave proving the second goal as an exercise. Note that, due to width constraints, the proof is broken up into named fragments. As is usual for proof trees, we place the root of the proof tree, fragment $P_0$, at the bottom and the remaining proof branch fragments ascending vertically.

Ground Reducibility Inference System. To show $\vec{E}_{\Delta}$ sufficiently complete with respect to $\Omega$ we need to prove goals of the form $\text{red}(f(x_1, \ldots, x_n)) \mid \top$. 

133
with \( f \in \Delta, f : s_1 \ldots s_n \rightarrow s \) a maximal typing for \( f \) in \( \vec{E} \), and \( x_i \) of sort \( s_i, 1 \leq i \leq n \). The inference rules that can be applied to prove such goals include all the shared rules plus the rule:

**Rewrite (RW)**

\[
\frac{\text{red}(f(\vec{v})) | \psi}{\text{red}(f(\vec{\tilde{v}})) | \psi}
\]

where \( f \in \Delta \) and there is a rule \((f(\vec{u}) \rightarrow r \mid \Gamma) \in \vec{E}_f \) and a substitution \( \theta \) such that \( f(\vec{\tilde{u}}) =_B f(\vec{u})\theta \) and \( T_{\Sigma_B/E_0 \cup U \cup B_0} \models \psi \Rightarrow (\Gamma \theta) \).

**Example 3.** In our running example we leave as an exercise for the reader the proofs of ground convergence and sufficient completeness for the first theory inclusion \( \vec{E}_1 < \vec{E}_2 \Delta \) as well as the RPO-modulo-based proof of operational termination of \( \vec{E}_1 \) (for which the MTA tool [20] can be used), and prove in detail both properties for the second theory inclusion \( \vec{E}_1 < \vec{E}_2 \Delta \). The proof of sufficient completeness needs to prove the single goal \( U \in V \mid \top \), with \( U, V \) of sort \( \text{MSet} \). Using the generating set \( \{ \emptyset, x, (y, W) \} \) for sort \( \text{MSet} \), we get the proof tree (rendered in a table due to width constraints as was done previously):

\[
\begin{array}{c}
P_3 \quad \text{red}((y, W) \cap V) \mid y \in V = \top \quad \text{RW} \quad \text{red}((y, W) \cap V) \mid y \in V = \bot \quad \text{RW} \quad \text{SP} \\
P_2 \quad \text{red}(x \cap V) \mid x \in V = \top \quad \text{RW} \quad \text{red}(x \cap V) \mid x \in V = \bot \quad \text{RW} \quad \text{SP} \\
P_1 \quad \text{red}(\emptyset \cap V) \mid \top \quad \text{RW} \\
P_0 \quad \text{red}(U \cap V) \mid \top \quad \text{CA} \\
\end{array}
\]

Therefore, assuming that the remaining proof obligations left as exercises for the reader have already been discharged, we have proved for our running example that: (1) the theory \( \vec{E}_1 \Delta \) for multisets of natural numbers is ground convergent, and (2) all instances of the term \( U \cap V \) by a ground constructor substitution \( \rho \) are \( \vec{E}_1 \Delta \)-reducible, i.e., \( \vec{E}_1 \Delta \) is sufficiently complete with respect to \( \Omega \). Thanks to our methodology, we can conclude that (3) \( \vec{E} \) itself is also ground convergent and sufficiently complete with respect to \( \Omega \), thus illustrating the entire hierarchical methodology. Fact (3) is a consequence of the following general theorem, whose proof can be found in Appendix A:

**Theorem 1.** Under the already-stated assumptions on a theory inclusion \( \vec{E}_0 \subset \vec{E} \), if \( \vec{E}_1 \Delta \) is ground convergent and sufficiently complete with respect to \( \Omega \), then \( \vec{E} \) is also ground convergent and sufficiently complete with respect to \( \Omega \).
Of course, the correctness of the hierarchical proof methodology crucially depends on the soundness of its inference system, i.e., on the following theorem, whose proof can also be found in Appendix A:

**Theorem 2. (Soundness Theorem).** Under the stated assumptions for the theory inclusion $\vec{E}_0 \subset \vec{E}^\Delta$ and for the joinability and reducibility goals, if the inference system proves a joinability goal of the form $t \downarrow t' \mid \varphi$, then $t \downarrow t' \mid \varphi$ holds in $\vec{E}^\Delta$. Likewise, if the inference system proves a reducibility goal of the form $\text{red}(f(u_1, \ldots, u_n)) \mid \varphi$, then $\text{red}(f(u_1, \ldots, u_n)) \mid \varphi$ holds in $\vec{E}^\Delta$.

### 4 Related Work and Conclusions

Research on sufficient completeness goes back to Guttag’s thesis in the 1970’s and includes, e.g., [22, 44, 8, 27, 29, 5, 23, 25, 24, 39, 38, 30, 42].

Early papers on methods to prove ground confluence appeared in the 1980s, including [46] and [41]. Subsequent work includes, e.g., [28, 16, 14, 17, 3, 4, 1, 12]. Since confluence implies ground confluence, work on methods and tools to prove confluence, e.g., [2, 43, 45, 39, 26] is also relevant. However, there are many ground confluent specifications that are not confluent.

Both sufficient completeness (even for unconditional theories as soon as a symbol is associative or associative-commutative) and ground confluence (again, even for conditional theories) are undecidable properties (see, respectively, [29] and [28]). This is not surprising, since both are inductive properties.

On sufficient completeness, two papers most closely related to this work, because both deal with order-sorted theories, are [39] and [4]. The work in [39] provides some useful methods for proving sufficient completeness of order-sorted CafeOBJ specifications and shares with our work the feature of exploiting module hierarchies; however, the methods used in [39] do not seem to support rewriting modulo axioms. The work in [4] shares a number of important ideas with the present work, including: (i) it supports conditional order-sorted theories; and (ii) it emphasizes that proofs of sufficient completeness and of ground confluence help each other, and, like us, it provides an inference system to prove both properties. Differences from [4] include that it does not support rewriting modulo axioms and that—as in the SPIKE prover [6], whose implementation it extends—instantiation of variables by terms in a generating set are favored over unification and narrowing-based approaches like ours. In that sense, our work also bears some loose similarities to an extensive body of work on ground confluence proof methods originating in the “inductionless induction” approach to inductive theorem proving, including, [16, 14, 17, 3, 1], all of which use narrowing and unification in their inference rules. However, besides the fact that the inference systems in that body of work are quite different from ours, none of that work supports order-sorted theories or rewriting modulo axioms. The work on ground confluence that is most closely related to ours is the one in [12]. In fact, the present work should be seen as further progress along the lines initiated in [12]. Specifically: (i) as pointed out in Section 3, our inference rules
for ground confluence include and extend those in [12]; and (ii) our hierarchical methods for proving ground confluence of conditional order-sorted theories extend and complement those presented in [12]. A key improvement in terms of greater applicability is that the theory inclusions $E_0 \subset E$ allowed in [12] had to obey the fairly restrictive assumption that some chosen sorts in $E_0$ could not have any extra terms in $E$. Our use of the theory inclusion $E_0 \subset E_\Delta$, for which that assumption holds by construction, completely obviates this restriction.

In conclusion, we have presented a new hierarchical methodology to prove conditional equational programs sufficiently complete and ground convergent. We have illustrated how the inference system works with the help of a running example. More inference rules can be added. For example, the Unfeasibility rule in [11] is an obvious addition. Also, a tool combining the inference systems in [12, 33] and in this work would allow further experimentation and would be quite useful in many verification efforts, not just for Maude, but also for the less general cases of many-sorted or unsorted equational programs in any language.

Acknowledgements. We cordially thank the anonymous referees for their very helpful suggestions, that have helped us improve the manuscript.

References


A Proofs

Proof of the Soundness Theorem 2.
Proof. For each inference rule we must show that if the premises of the rule hold, then the conclusion follows. We do so for each inference rule. Recall that in all applications, i.e., to prove either a ground joinability or a ground reducibility property in $E^A$, the meaning of $p \mid \varphi$ holding is that it does so for all its ground constructor substitutions $\rho$ such that $\varphi\rho$ holds in $E_0$.

Shared Inference Rules. Except for rule GN, all these rules correspond to equivalences. That is, the premises hold iff the conclusion does. Let us consider each inference rule.

- NA. For any ground constructor substitution $\rho$, at position $p$ in $\varphi$ the term $f(\bar{v}\rho)$ has constructor term arguments. Therefore, by sufficient completeness of $E_0$, there is a rewrite rule in $E_0$, say rule [1] whose lefthand side $f(\bar{u}_i)$ is $B$-matched by $f(\bar{v}\rho)$ with a ground constructor substitution $\gamma$, i.e., $f(\bar{v})\rho = f(\bar{u}_i)\gamma$, and whose condition instance $\Gamma\gamma$ holds in $E_0$. Therefore, we can rewrite $f(\bar{v}\rho)$ to the instance $r_i\gamma$ of its righthand side. Therefore, there is a $B$-unifier $\alpha_{i,j}$ of the equation $f(\bar{v}) = f(\bar{u}_i)$ and a ground constructor substitution $\delta$ such that $\rho \wedge \gamma = \alpha_{i,j}\delta$. Therefore $\varphi\rho$ holds in $E_0$ iff $(\Gamma_i \cup \varphi[r_i]_\rho)\alpha_{i,j}\delta$ holds, and of course $up = _B \omega_{i,j}\delta$. In brief, the equivalence summarizes symbolically (by narrowing) all the possible ways in which all ground constructor instances of condition $\varphi$ can be rewritten in one step at position $p$.

- UN. If $(\varphi \land \psi)\rho$ holds in $E_0$, then $\psi\rho$ does, i.e., $\rho$ is a $U \cup E_0 \cup B_0$-unifier of $\psi$. Therefore, there must be a $U \cup E_0 \cup B_0$-unifier $\theta$ of $\psi$ and a ground constructor substitution $\gamma$ such that $\rho = _{U \cup E_0 \cup B_0} \theta\gamma$. The equivalence follows naturally from this fact.

- ES. The main result about equality predicates in [21] is that for any Boolean formula $\varphi$ and ground constructor substitution $\rho$, $\varphi\rho$ holds in ground convergent $E_0$ iff $\varphi\rho \triangleleft E_0 \rho$ does. In particular, this equivalence holds when $\varphi$ is a conjunction of equalities.

- CA. The equivalence follows from the definition of a generating set for the sort $s$ of $x$, since for any ground constructor substitution $\rho$, $\rho(x)$ must be such that $\rho(x) = _B v_i\gamma$ for some $v_i$ in such a set and ground constructor substitution $\gamma$.

- SP. The equivalence between the premises and the conclusion follows from the semantic equivalence $T_{\Sigma_0/E_0\cup B_0} \models \varphi \Leftrightarrow \bigvee_{i \in I} \psi_i \land \varphi$, plus the Boolean equivalence $(A \lor B) \Rightarrow C \equiv (A \Rightarrow C) \land (B \Rightarrow C)$.

- GN. This is the only shared rule tat is not an equivalence, i.e., where the premise implies the consequence but need not be equivalent to it. The property $p'\rho$ must hold (i.e., $p'\rho$’s ground reducibility, or $p'\rho$’s ground joinability, depending on $p$) whenever $\psi\rho$ does. In particular, if $\varphi\gamma$ holds, then $\psi\theta\gamma$ does, and therefore $p'\theta\gamma$ does. That is, $p'\theta \mid \varphi$ holds. But $p'\theta = _B p$. The result then follows from the fact that for either ground reducibility or ground joinability
properties \( q, q' \) such that \( q = B q' \), \( q \mid \varphi \) holds iff \( q' \mid \varphi \) does. This follows in either case from the assumption that the rules \( \hat{U} \cup \hat{E} \) are strictly \( B \)-coherent.

- \( \emptyset \). Since no ground substitution can satisfy \( \bot, u \mid \bot \) holds trivially.

**Ground Joinability Inference System.** The proof of the constrained version of the ground confluence inference rules in [12] follows easily from that of the unconstrained inference rules in [12]. The soundness of rule \( JN \) holds trivially from the very notion of joinability. A proof of soundness for the \( CR \) inference rule can be found in [11].

**Ground Reducibility Inference System.** The only inference rule is \( RW \). Suppose that \( \psi \rho \) holds in \( \hat{E}_0 \). Then, \( \Gamma \theta \rho \) does; and by the rule's assumptions \( f(\overline{v})\rho \) is reducible, as desired.

This finishes the proof of the Soundness Theorem. \( \square \)

**Proof of Theorem 1.**

*Proof. First of all, note that, considering \( T_\Sigma(X) \) and \( T_{\Sigma,A}(X) \) as sets, i.e., disregarding sorts, we have an inclusion \( T_\Sigma(X) \subseteq T_{\Sigma,A}(X) \). Also, for each \( s \in S \) we have a set equality \( T_{\Sigma_0,s}(X) = T_{\Sigma,A,s}(X) \). In particular, \( T_\Sigma \subseteq T_{\Sigma,A} \), and \( T_{\Sigma_0,s} = T_{\Sigma,A,s} \) for each \( s \in S \).

Second, \( \hat{E} \) and \( \hat{E}^A \) have the exact same CCP’s. To begin with, in both cases the rules not in \( \hat{E}_0 \) are the same, namely \( \hat{E}_\Delta \). Furthermore, in both cases, the only CCP’s that do not come from \( \hat{E}_0 \) can be of only two kinds: (i) between a unit rule in \( \hat{U} \) and a rule in \( \hat{E}_f \) for some \( f \in \Delta \), where the unit rule’s lefthand side unifies with a constructor subterm of the lefthand side of one of \( f \)’s constructor arguments; or (ii) between two, not necessarily different, rules in \( \hat{E}_f \) for some \( f \in \Delta \).

In case (i), the unifier generating the CCP must be a constructor unifier so that the resulting CCP is the same in both \( \hat{E} \) and \( \hat{E}^A \), and its condition is a \( \Sigma_0 \)-condition. In case (ii), the CCP comes from two —not necessarily different, but variable-renamed if \( i = j \) to ensure disjoint variables— rules \( [i] : f(\overline{u}_i) \rightarrow r_i \), if \( \Gamma_i \), and \( [j] : f(\overline{u}_j) \rightarrow r_j \) if \( \Gamma_j \) and its associated order-sorted unifier (in either \( \hat{E} \) or \( \hat{E}^A \)) solves the equation \( f(\overline{u}_i) = f(\overline{u}_j) \). We claim that the order-sorted unifiers of the equation \( f(\overline{u}_i) = f(\overline{u}_j) \) are the same in \( \hat{E} \) and in \( \hat{E}^A \). Recall that, by assumption, \( B_f \) is either empty or a commutativity axiom. If \( B_f = \emptyset \), then \( \alpha \) is a unifier of \( f(\overline{u}_i) = f(\overline{u}_j) \) iff it is a unifier of the system of equations \( u_{i,1} = u_{j,1} \land \ldots \land u_{i,k} = u_{j,k} \), where \( k \) is the number of arguments of \( f \). If \( f \) is commutative, the only difference is that in \( \hat{E}^A \) the axiom \( f(x_1,x_2) = f(x_2,x_1) \) is such that \( x_1, x_2 \) have sort \( s \) for \( f : s \rightarrow s_0 \) the maximal typing of \( f \), whereas in \( \hat{E} \) \( x_1, x_2 \) have kind \( [s] \). This, however, makes no difference, since, by the Decomposition inference rule for a commutative symbol of order-sorted unification (see [31] and [7] §15.1), \( \alpha \) is a unifier of \( f(u_{i,1}, u_{i,2}) = f(u_{j,1}, u_{j,2}) \) iff it is a unifier of the disjunction of systems of equations \( (u_{i,1} = u_{j,1} \land u_{i,2} = u_{j,2}) \lor (u_{i,1} = u_{j,2} \land u_{i,2} = u_{j,1}) \). Therefore, the CCPs are the same and the unifiers are constructor unifiers, so that the CCP’s condition is a \( \Sigma_0 \)-condition.
Third, for ground terms we have proper inclusions of rewrite relations,
\[ \to_{\vec{E}} \subseteq \to_{\vec{E}}^\Delta \subseteq \to_{\vec{E}} \subseteq T_{\vec{E}}^\Delta \times T_{\vec{E}}^\Delta. \]

The first inclusion is proper because there are terms in \( T_{\vec{E}} \setminus T_{\vec{E}_0} \) that can be rewritten with \( \to_{\vec{E}}^\Delta \). The second inclusion is proper because, by the definition of \( \Sigma^\Delta \), a rule in the theory \( \vec{E}_\Delta \), say, \([i]: f(\vec{u}) \mapsto r_i \) if \( \Gamma_i \), can, only be enabled to rewrite a term \( f(\vec{v}) \) if the terms \( \vec{v} \) are \( \Sigma^\Delta \)-terms. That is, \( \to_{\vec{E}}^\Delta \) performs rewritings exactly like \( \to_{\vec{E}} \), but only in a "weakly innermost" manner ("weakly" because the \( \Sigma^\Delta \)-terms \( \vec{v} \) need not be constructors).

Fourth, for any \( \bar{t} \in T_{\vec{E}} \), \( \bar{t} \to_{\vec{E}} \bar{u}^\Delta \) is a constructor term. Suppose not, i.e., there is a \( \bar{t} \in T_{\vec{E}} \) such that \( \bar{t} \to_{\vec{E}} \bar{u}^\Delta \) is not a constructor term. But since we have an inclusion of rewrite relations \( \to_{\vec{E}} \subseteq \to_{\vec{E}}^\Delta \) and \( \vec{E}_0 \) is sufficiently complete, this means that \( \bar{t} \to_{\vec{E}} \bar{u}^\Delta \) must contain a subterm of minimal size of the form \( f(\vec{u}) \) with \( f \in \Delta \) and the terms \( \vec{u} \) constructor terms. But this is impossible, since all such terms have been proved \( \vec{E}_\Delta \)-reducible.

Fifth, for any \( \bar{t} \in T_{\vec{E}} \), if \( \bar{t} \to_{\vec{E}}^\Delta \bar{v} \) and \( \bar{v} \) is in \( \vec{E} \)-canonical form, then \( \bar{v} \) is a constructor term. This follows from the containments of rewrite relations \( \to_{\vec{E}} \subseteq \to_{\vec{E}}^\Delta \) and \( \vec{E}_0 \) is the fourth property above, and the sufficient completeness of \( \vec{E}_0 \).

Finally, we are now ready to prove that \( \vec{E} \) is ground convergent. Note that, by the fifth property above, \( \vec{E} \) is then also sufficiently complete with respect to \( \Omega \). Since we have the containment of ground rewrite relations \( \to_{\vec{E}} \subseteq \to_{\vec{E}}^\Delta \), the ground convergence of \( \vec{E} \) will follow from the fourth and fifth properties above if we can prove that for each \( \bar{t} \in T_{\vec{E}} \) and each ground constructor term \( \bar{v} \) such that \( \bar{t} \to_{\vec{E}} \bar{v} \) we have \( \bar{v} = B_\bar{t} \bar{t} \to_{\vec{E}} \bar{u}^\Delta \).

**Lemma 1.** For each \( \bar{t} \in T_{\vec{E}} \), if \( \bar{t} \to_{\vec{E}}^\Delta \bar{u} \) and \( \bar{u} \) is a constructor term, then \( \bar{u} = B_\bar{t} \bar{t} \to_{\vec{E}}^\Delta \).

**Proof.** Suppose not. Let us choose a term \( \bar{t} \in T_{\vec{E}} \) such that: (i) \( \bar{t} \to_{\vec{E}}^\Delta \bar{u} \) and \( \bar{u} \) is a constructor term, and (ii) \( \bar{t} \neq B_\bar{t} \bar{t} \to_{\vec{E}}^\Delta \). Therefore, by the RPO order modulo proving \( \vec{E} \) operationally terminating, \( \bar{t} \) is a minimal element among the set of terms in \( T_{\vec{E}} \) such that (i) holds. This can only happen if \( \bar{t} \) is not a constructor term. Therefore, we have \( \bar{t} \to_{\vec{E}} \bar{v} \to_{\vec{E}} \bar{u} \). Note that \( \bar{t} > \bar{v} \). Therefore, by the minimality assumption for \( \bar{t} \), we must have \( u = B_\bar{t} \bar{t} \to_{\vec{E}}^\Delta \). Let us now consider the one-step rewrite \( \bar{t} \to_{\vec{E}} \bar{v} \). This means that there is a rule \( f(\bar{v}) \to_{\vec{E}} \bar{r} \) if \( \Gamma \) in \( \vec{U} \cup \vec{E} \) with the \( \bar{u} \) constructor terms (rules in \( \vec{U} \), though unconditional, also have this form), a ground substitution \( \alpha \) and a term position \( p \) such that \( t|_p = B f(\bar{u}) \alpha \), \( \Gamma \alpha \) holds in \( \vec{E} \), and \( \bar{t} = (t|_p)_{\alpha \rho} \). Since \( \bar{t} > \bar{u} \) is a \( B \)-compatible RPO order and all rules are assumed \( \to_{\vec{E}} \)-operationally-terminating, for each equality \( w = w' \in \Gamma \) we must have \( t > w \alpha, w' \alpha \). Therefore, by the minimality hypothesis on \( t \), we must have \( (w \alpha)_{\alpha \rho} = B_{\alpha \rho} (w' \alpha)_{\alpha \rho} \), so that \( \Gamma \alpha \) also holds in \( \vec{E}^\Delta \) and, for the same reason, \( \Gamma \rho \) holds in \( \vec{E}^\Delta \) for the constructor substitution \( \rho = \alpha_{\bar{u}^\Delta} \) obtained by normalizing each \( \alpha(x) \) with \( x \) in the domain of \( \alpha \). Therefore, we have a rewrite \( t(f(\bar{u})_{\rho})_{\rho} \to_{\vec{E}} \bar{u}^\Delta \). Furthermore, \( t = B t(f(\bar{u})_{\alpha \rho})_{\rho} \), and we
have rewrite sequences $t[f(\vec{u})\alpha]_p \rightarrow^*_{\vec{E}^\Delta} t[f(\vec{u})\rho]_p$, and $t[r\alpha]_p \rightarrow^*_{\vec{E}^\Delta} t[r\rho]_p$, and since $t > t' = t[r\alpha]_p$, we must have $\vec{u} =_{B_{\bar{\Delta}}} t[r\rho]_p !_{\vec{E}^\Delta}$. In summary, we have the sequence of rewrites in $\vec{E}^\Delta$,

$$t[f(\vec{u})\alpha]_p \rightarrow^*_{\vec{E}^\Delta} t[f(\vec{u})\rho]_p \rightarrow_{\vec{E}^\Delta} t[r\rho]_p \rightarrow^*_{\vec{E}^\Delta} t[r\rho]_p !_{\vec{E}^\Delta}$$

with $u =_{B_{\bar{\Delta}}} t[r\rho]_p !_{\vec{E}^\Delta}$. But by $t =_B t[f(\vec{u})\alpha]_p$ and the convergence of $\vec{E}^\Delta$ we also must have $t!_{\vec{E}^\Delta} =_{B_{\bar{\Delta}}} t[r\rho]_p !_{\vec{E}^\Delta} =_{B_{\bar{\Delta}}} u$, contradicting the assumption $u \not=_{B_{\bar{\Delta}}} t!_{\vec{E}^\Delta}$, as desired. □

This finishes the proof of Theorem 1. □
Parallel Maude-NPA for Cryptographic Protocol Analysis

Canh Minh Do, Adrián Riesco, Santiago Escobar, and Kazuhiro Ogata

1 Japan Advanced Institute of Science and Technology (JAIST)
{canhdominh,ogata}@jaist.ac.jp
2 Universidad Complutense de Madrid
ariesco@fdi.ucm.es
3 VRAIN, Universitat Politècnica de València
sesobar@upv.es

Abstract. Maude-NPA is a symbolic model checker for analyzing cryptographic protocols in the Dolev-Yao strand space model modulo an equational theory defining the cryptographic operations, which starts from an attack state to find counterexamples by performing a backward narrowing reachability analysis. Although Maude-NPA is a powerful analyzer, its running performance can be improved by taking advantage of parallel and/or distributed computing when dealing with non-trivial protocols in which the state space is huge. This paper describes a parallel version of Maude-NPA and a tool that supports it. We report on some experiments of various kinds of protocols that demonstrate that the tool can increase the running performance of Maude-NPA by 30% on average for most non-trivial case studies in which the number of states located at each layer is considerably large.

Keywords: Maude · Cryptographic Protocol Analysis · Parallel Maude-NPA.

1 Introduction

With the emergence of the Internet and network-based services, many cryptographic protocols, also called security protocols, have been developed over time.
decades to provide information security, such as confidentiality and authentication, in an insecure network. The design of cryptographic protocols, such as authentication protocols, is difficult, error-prone, and hard to detect flaws [13]. Many protocols contain flaws even a long time after they were published. For example, Lowe found an attack on the Needham-Schroeder public key authentication protocol after seventeen years [33,27]. Therefore, it is important to have automated tools to verify the properties of cryptographic protocols. There are several tools dedicated to security protocol analysis, such as Athena [34], ProVerif [6], OFMC [5], Avispa [1], Scyther [10], TAMARIN [28], Maude-NPA [16], and Verifpal [22]. Most analyzers based on model checking suffer from the notorious state space explosion problem, which prevents some model checking experiments from being carried out. Another challenge is to increase the running performance of model checking. One promising approach to this challenge is to parallelize model checking, which can make best use of multicore architectures.

Maude-NPA is a powerful symbolic model checker for analyzing cryptographic protocols modulo an equational theory that uses the Dolev-Yao strand space model [12,18], which is capable of intercepting, modifying, and injecting messages to impersonate other protocol principals by intruders. Maude-NPA uses a backward narrowing reachability analysis, which starts from a final insecure state, an attack state, to determine whether or not it is reachable from an initial state, which has no further backward steps. If that is the case, the initial state is a counterexample. The advantage of Maude-NPA is that it supports an unbounded session model and different equational theories; as a counterpart, these theories often lead to a bigger state space that requires more time to conduct model checking. Although some techniques were devised to reduce the state space, such as grammar-based techniques, giving priority to input messages in strands, early detection of inconsistent states (never reaching an initial state), a relation of transition subsumption (to discard transitions and states already being processed in another part of the search space), and the super lazy intruder (to delay the generation of substitution instances as much as possible) [14,15], the state space explosion problem is inevitable in some cases. Therefore, improving the running performance of Maude-NPA to some extent is worth doing. We are aware that Maude-NPA basically uses a breadth-first search (BFS) to explore the state space. Given a set of states in layer $l$, for each state in the set, we can perform independently the backward narrowing to obtain its successor states in layer $l + 1$, which opens an opportunity for parallelization so as to improve the running performance of Maude-NPA (time challenge). Note that an attack state is located at layer 0 and all states that reach the attack state by one-step state transition are located at layer 1.

In the present paper, we describe a parallel version of Maude-NPA and a tool that supports it, where successor states are generated in parallel at each layer. Basically, we transform the breadth-first search in Maude-NPA into a parallel breadth-first search without altering the number or form of the states in the state space. If the number of states located at each layer is considerably large, our tool can effectively improve the running performance of Maude-NPA. The tool has
been built in Maude [8] as an implementation language, which is one direct successor language of OBJ3 [19], an algebraic specification language, and based on rewriting logic [31] as its theoretical foundation. Maude-NPA is written in Maude, which supports adequate parallel facilities, making it possible to develop parallel tools, such as Parallel L + 1-DCA2L2MC [11], which is a new technique to mitigate the state explosion for leads-to model checking. Therefore, the use of Maude for the parallel development of Maude-NPA without extra conversion is superior to other programming languages. The architecture of the tool is a master-worker model where one master and multiple workers are involved. The tool uses a shared cache maintained by the master and a local cache maintained by each worker to avoid making unnecessary duplications of jobs. The tool uses a set of jobs and a queue of worker identifiers to distribute (or assign) jobs to workers in a well-balanced way. The present paper also reports on some case studies on various kinds of protocols that demonstrate that the tool can increase the running performance of Maude-NPA by 30% on average for most non-trivial case studies. The support tool is available at the webpage.4

The rest of the paper is organized as follows. Section 2 mentions some preliminaries in which narrowing is described. Section 3 describes the overview of Maude-NPA. Section 4 describes a parallel version of Maude-NPA and a tool that supports it. Section 5 reports on some experimental results. Section 6 mentions some existing work. Finally, Section 7 concludes the paper together with some future directions.

2 Preliminaries

We follow the classical notation and terminology from [21] for term rewriting and from [29,30] for rewriting logic and order-sorted notions. We assume an order-sorted signature \( \Sigma \) with a finite poset of sorts \((S, \leq)\) and a finite number of function symbols. We furthermore assume that: (i) each connected component in the poset ordering has a top sort, and for each \( s \in S \) we denote by \([s]\) the top sort in the component of \( s \); and (ii) for each operator declaration \( f : s_1 \times \ldots \times s_n \rightarrow s \) in \( \Sigma \), there is also a declaration \( f : [s_1] \times \ldots \times [s_n] \rightarrow [s] \). We assume an \( S \)-sorted family \( \mathcal{X} = \{ X_s \}_{s \in S} \) of disjoint variable sets with each \( X_s \) countably infinite. \( T_\Sigma(\mathcal{X})_s \) is the set of terms of sort \( s \), and \( T_\Sigma,s \neq \emptyset \) for every

4 https://github.com/canhminhdo/parallel-maude-npa
sort \( s \), order-sorted equational logic induces a congruence relation \( =_E \) on terms \( t, t' \in T_\Sigma(X) \) (see \[30\]). We assume that \( T_{\Sigma, s} \neq \emptyset \) for every sort \( s \). The E-subsumption order on terms \( T_\Sigma(X)_{\neq} \), written \( t \preceq_E t' \) (meaning that \( t' \) is more general than \( t \)), holds if \( \exists \sigma : t =_E \sigma(t') \). The E-renaming equivalence on term \( T_\Sigma(X)_{\neq} \), written \( t \equiv_E t' \), holds if \( t \preceq_E t' \) and \( t' \preceq_E t \). An E-unifier for two terms \( t, t' \in T_\Sigma(X) \) is a substitution \( \sigma \) such that \( \sigma(t) =_E \sigma(t') \). A complete set of \( \Sigma \)-unifiers of two terms \( t, t' \) is defined in Maude-NPA, which consists of the intruder strands, regular strands, and attack states. Maude-NPA starts from an attack state, a final insecure state, to

A rewrite rule is an oriented pair \( l \rightarrow r \), where \( l \notin \Sigma \) and \( l, r \in T_\Sigma(X)_{\neq} \) for some sorts \( s \in S \). An (unconditional) order-sorted rewrite theory is a triple \( \mathcal{R} = (\Sigma, E, R) \) with \( \Sigma \) an order-sorted signature, \( E \) a set of \( \Sigma \)-equations, and \( R \) a set of rewrite rules. A topmost rewrite theory is a rewrite theory such that for each \( l \rightarrow r \in R \), \( l \in T_\Sigma(X)_{\text{State}} \) for a top sort \( \text{State} \), \( r \notin \Sigma \), and no operator in \( \Sigma \) has \text{State} as an argument sort. The rewriting relation \( \rightarrow_R \) on \( T_\Sigma(X) \) is \( t \stackrel{\Delta}{\rightarrow}_R t' \) (or \( \rightarrow_R \)) if \( p \in \text{Pos}_\Sigma(t), l \rightarrow r \in R, l|_p = \sigma(l), \) and \( t' = t[\sigma(r)]_p \) for some \( \sigma \). The relation \( \rightarrow_{R/E} \) on \( T_\Sigma(X) \) is defined in the same way. A complete unification algorithm, the narrowing relation \( \rightarrow_{R,E} \) on \( T_\Sigma(X) \) is \( t \backsim_{R,E} t' \) (or \( \sim_{R,E} \)) if \( p \in \text{Pos}_\Sigma(t), l \rightarrow r \in R, \sigma \in CSU_E(t|_p = l), \) and \( t' = \sigma(t[r]_p) \). Assuming that \( E \) has a finitary and complete unification algorithm, the narrowing relation \( \rightarrow_{R,E} \) on \( T_\Sigma(X) \) is \( t \backsim_{R,E} t' \) (or \( \sim_{R,E} \)) if \( p \in \text{Pos}_\Sigma(t), l \rightarrow r \in R, \sigma \in CSU_E(t|_p = l), \) and \( t' = \sigma(t[r]_p) \). Note that \( \rightarrow_{R,E} \) on \( T_\Sigma(X) \) induces a relation \( \rightarrow_{R,E} \) on \( T_\Sigma/E(X) \) by \( [t]_E \rightarrow_R [t']_E \) if and only if \( t \rightarrow_{R,E} t' \).

The narrowing relation \( \rightarrow_{R,E} \) on \( T_\Sigma(X) \) is \( t \backsim_{R,E} t' \) (or \( \sim_{R,E} \)) if \( p \in \text{Pos}_\Sigma(t), l \rightarrow r \in R, \sigma \in CSU_E(t|_p = l), \) and \( t' = \sigma(t[r]_p) \). Assuming that \( E \) has a finitary and complete unification algorithm, the narrowing relation \( \rightarrow_{R,E} \) on \( T_\Sigma(X) \) is \( t \backsim_{R,E} t' \) (or \( \sim_{R,E} \)) if \( p \in \text{Pos}_\Sigma(t), l \rightarrow r \in R, \sigma \in CSU_E(t|_p = l), \) and \( t' = \sigma(t[r]_p) \). Note that \( \rightarrow_{R,E} \) on \( T_\Sigma(X) \) induces a relation \( \rightarrow_{R,E} \) on \( T_\Sigma/E(X) \) by \( [t]_E \backsim_{R,E} [t']_E \) if and only if \( \exists w \in T_\Sigma(X) : t \sim_{R,E} w \) and \( w \equiv_E t' \).

3 Maude-NPA

Maude-NPA [16] is a model checker for analyzing cryptographic protocols modulo equations, which is written in Maude with about 18,000 lines of code. This section gives an overview of Maude-NPA focusing on those pieces of code that will be used for parallelization while omitting the rest.

A protocol specification in Maude-NPA is done by overwriting the three predefined modules: PROTOCOL-EXAMPLE-SYMBOLS, PROTOCOL-EXAMPLE-ALGEBRAIC, and PROTOCOL-SPECIFICATION that specify the syntax of the protocol, which consist of sorts, subsets, and operators, the algebraic properties of the operators, which consist of equational rules (equations) and equational axioms (axioms), and the actual behaviors of the protocol using the Dolev-Yao strand space model [12,18], which consists of the intruder strands, regular strands, and attack states. Maude-NPA starts from an attack state, a final insecure state, to
perform a backward reachability analysis which determines whether or not it is reachable from an initial state, which has no further backward steps. If that is the case, the initial state is a counterexample. The backward search is performed by a backward narrowing with a symbolic execution since the attack state is a term with logical variables. Each backward narrowing step can be regarded as a state transition, such as sending or receiving a message by principals, or manipulating a message by intruders. Given a symbolic state, a backward narrowing step is performed to return a previous symbolic state in the protocol. By that we can obtain all successor states (in the backward sense) from the state. In Maude-NPA, each state found during the backward analysis is represented by six sections separated by the symbol || in the following order: (1) state id, (2) set of current protocol and intruder strands, (3) intruder knowledge, (4) sequence of messages, (5) ghost list, and (6) never pattern. For instance, the following is a state found during the backward analysis of the Needham-Schroeder public key protocol:

```plaintext
< 1 . 9 > (  
  :: nil ::  
  [ nil  
    -(n(b, #0:Fresh)),  
    +(pk(b, n(b, #0:Fresh))), nil] &  
  :: #0:Fresh ::  
  [ nil,  
    -(pk(b, #1:NNSet ; a)),  
    +(pk(a, #1:NNSet ; b * n(b, #0:Fresh))) |  
    -(pk(b, n(b, #0:Fresh))), nil]  
) ||  
pk(b, n(b, #0:Fresh)) !inI,  
n(b, #0:Fresh) inI,  
irr(pk(b, n(b, #0:Fresh))) ||  
-(n(b, #0:Fresh)),  
+(pk(b, n(b, #0:Fresh))),  
-(pk(b, n(b, #0:Fresh))) ||  
nil ||  
nil
```

The state id is a unique id assigned to each state during the backward analysis. The set of current strands represents the messages that were sent or received in the past (those messages before the symbol |) and the messages that will be sent or received in the future (those messages after the symbol |) in each strand. The strand set also indicates how to advance each strand in the execution process by partial substitutions for the messages in each strand. The intruder knowledge represents what messages the intruder knows (symbol _inI) or does not know yet (symbol _!inI) at each state. The sequence of messages denotes the actual
sequence of messages communicated to reach the state. The ghost list is extra information for optimization in the super lazy intruder technique to reduce the state space. The never pattern is used for authentication attacks.

We can divide the whole process of Maude-NPA into two main stages. In the first stage, given a protocol specification $\mathcal{P}$ and an equational theory $E_P$, Maude-NPA needs to do as follows:

- Extracting the attack state $St$ from the protocol given an attack state id file.
- Building rewrite rules $R_P$ against the behavior of the protocol specified in form of intruder and regular strands along with some pre-defined rewrite rules in the Maude-NPA specification.
- Generating grammars that represent infinite sets of states unreachable for the intruder to reduce the state space.

In the second stage, Maude-NPA performs the backward narrowing reachability analysis from the attack state $St$ using the relation $\Rightarrow_{R_P^{-1}, E_P}$ where $R_P^{-1}$ is the set of rewrite rules derived from $R_P$ by inverting its rewrite rules. Maude-NPA basically uses the breadth-first search to explore the state space. There are three main steps needed for the exploration of each layer as follows:

- The first step is to generate all successor states for the next layer given a set of states in the current layer. This step also consists of almost all techniques to reduce the state space except for the transition subsumption technique, which is used in the second step subsequently.
- The second step is to simplify the successor states by the transition subsumption technique for removing states that are subsumed by either other states in the successor states or visited states (history states).
- Ultimately, the third step will filter out states from the previous step by using history states to avoid state duplications and rule out initial states as counterexamples. The cycle continues until a depth bound is reached or no more states exist for the next layer.

The first step in the second stage actually performs the backward narrowing just by one step to obtain all successor states from a given set of states in a layer. The successor states then go through a series of optimization steps, such as giving priority to input messages in strands, early detection of inconsistent states, the super lazy intruder, and filtering states by the grammars. We are aware that this step can be executed independently for each given state from the set of states, which opens an opportunity for parallelization. Given a set of states in layer $l$, for each state in the set, we can perform the backward narrowing step independently to obtain its successor states in layer $l+1$. In the next section, we will describe a parallel version of Maude-NPA in which the successor states are generated in parallel for each layer. Note that such reduction techniques are also included in this step and the parallel version does not alter the number or form of the states in the state space. If the number of states located at each layer is considerably large, our tool can effectively improve the running performance of Maude-NPA.
In addition, the second step in the second stage plays an important role to reduce the state space in Maude-NPA, which may transform an infinite-state system into a finite one [17], and this is also time-consuming. Basically, it performs two series of transition subsumption tasks, also called the implication step throughout this paper. Firstly, for each state in the successor states obtained in the first step of the stage it will be checked whether or not the state is subsumed (implied) by another state in the successor states. If that is the case, the state is ignored. We only keep states which cannot be subsumed by other states after this process. Secondly, each state will be checked whether or not the state is subsumed (implied) by a state in history states again. If that is the case, the state is ignored. Otherwise, the state is stored. We plan to parallelize the whole process of this step in the near future as one piece of our future work.

4 Parallel Maude-NPA and Its Tool Support

The support tool is implemented in Maude to conveniently extend the implementation of what is developed in Maude-NPA. We use object-based programming that can model an object-based system, where objects can communicate to each other via message passing. In addition, Maude also supports communicating with external objects by using sockets so that objects inside an object-based system can interact with different objects inside another object-based system. We adopt such functionalities to make a parallel version of Maude-NPA based on a master-worker model, which is described in this section.

As mentioned above, we parallelize the backward narrowing step in Maude-NPA. We use a master-worker model to make a parallel version of Maude-NPA. In our tool, a master maintains a shared cache that is a set of states (history states), while each worker also maintains a local cache that is a set of states, which contains all states explored by the worker. Use of the shared cache prevents jobs that have been processed from being assigned to workers, while use of the local caches prevents jobs that have been processed from being made by workers. The very initial job is made by the master, while all the other jobs are made by workers and basically sent to the master. Jobs are assigned to workers by the master unless the jobs have been tackled.

There are two kinds of messages exchanged by the master and workers: job and getJob. A job message is in the form of a state. A job message is sent to a worker by the master, distributing (or assigning) a job to the worker. Meanwhile, a job message is sent to the master by a worker, delivering a job made by the worker to the master. A getJob message is sent to the master by a worker, asking the master to assign a job to the worker. In Maude, we can send data over sockets provided that the data must be a string. The getJob message is just literally a string “getJob,” while the job message is in the form of the state described in Section 3 in which the state will be transformed into a string before sending at the sender side and be restored to the original state from the string at the receiver side. The following functions written in Maude are used to do the transformation and restoration.
op qidListToString : QidList -> String .
op qidListToString : QidList String -> String .
eq qidListToString(QIL) = qidListToString(QIL, "") .
eq qidListToString(nil, S) = S .
eq qidListToString(Q QIL, S) = qidListToString(QIL, S + string(Q) + " ") .

op stringToQidList : String -> QidList .
op stringToQidList : String QidList -> QidList .
eq stringToQidList(S) = stringToQidList(S, nil) .
eq stringToQidList("", QIL) = QIL .
eq stringToQidList(S, QIL) = QIL qid(S) [owise] .
ceq stringToQidList(S, QIL) = stringToQidList(S', QIL qid(S'))
  if N := find(S, " ", 0)
  \ S' := substr(S, 0, N)
  \ S'' := substr(S, N + 1, length(S)) .

op state2string : IdSystemSet -> String .
eq state2string(State) = qidListToString(
  metaPrettyPrint(SM, upTerm(State), none)) .

op string2state : String -> IdSystemSet .
eq string2state(S) = downTerm(getTerm(
  metaParse(SM, stringToQidList(S), 'IdSystemSet)), errIdSystemSet) .

where State and SM are Maude variables of Module and IdSystemSet sorts, respectively. The function string2state is used to transform a state into a string by doing the following order: (1) convert the state to its meta representation by using the function upTerm, (2) convert the meta representation of the state to a list of quoted identifiers that presents the string of tokens, and (3) convert the list of quoted identifiers to a string by the function qidListToString. The function string2state is used to restore the state from the string by doing the following order: (1) convert the string to a list of quoted identifiers by the function stringToQidList, (2) parse the list of quoted identifiers in the module SM by the function metaParse, (3) get the term, the meta representation of the state, from the output of the function metaParse by the function getTerm, and (4) convert the meta representation of the state to the original state by the function downTerm. Note that some essential functions, such as upTerm, downTerm, getTerm, metaPrettyPrint, metaParse, are built-in in Maude, while the two functions qidListToString and stringToQidList are defined above.

The master is in charge of collecting all successor states (jobs) from workers, then performing the implication step to remove implied states, checking state duplications with history states, ruling out initial states as counterexamples, and distributing (or assigning) unprocessed jobs to workers. Besides, the master can stop the tool whenever counterexamples are found or there are no unprocessed jobs left or a depth bound is reached. Meanwhile, each worker is responsible for processing a job, a state, assigned to it by the master. A worker generates all successor states reachable from the state by the backward narrowing and checks them with its local cache to avoid explored states. The worker may then
Algorithm 1: Job Scheduling by a Master

**input**: P – a protocol specification
Id – an attack state id in the protocol specification
F – a filter
BStep – the maximum number of backward steps
N – a number of workers

**output**: empty or counterexamples

1. (workers, jobs, next, history) ← (empty, empty, empty, empty);
2. (M, GS, IS) ← initialize(P, Id, BStep, F);
3. jobs ← {IS}; history ← {IS};
4. while True do
5.   for k ← 1 to N do
6.     if DATA ← recv(worker_k) then
7.       if DATA = getJob then
8.         enqueue(workers, worker_k);
9.       else
10.      (IS) ← DATA;
11.      next ← insert(next, IS);
12.     while not isEmpty(workers) and not isEmpty(jobs) do
13.       worker ← dequeue(workers);
14.       IS ← dequeue(jobs);
15.       send(worker, IS);
16.     if isEmpty(jobs) and size(workers) = N then
17.       if not isEmpty(next) then
18.         IST ← simplifyByImplication(F, history, next);
19.         (INIT, IST') ← filterWithHistoryAndInit(M, history, IST);
20.         (jobs, next, BStep) ← (IST', empty, BStep - 1);
21.         history ← insert(history, IST');
22.       if not isEmpty(INIT) then
23.         closeConnection();
24.       return INIT;
25.     if BStep = 0 or isEmpty(jobs) then
26.       closeConnection();
27.     return empty;

construct new jobs and send them to the master as job messages. At last, when a worker has completed a job, the worker requests a new job by sending a getJob message to the master. The master uses a set of states and a queue of worker identifiers to distribute jobs to workers so that job distribution can be well-balanced, which means that all workers are processing jobs all the time except the beginning, the implication step, and ending of the backward narrowing.

Algorithm 1 shows the pseudo-code for job scheduling conducted by the master. workers is a queue data structure that contains worker identifiers, which are requesting jobs. jobs, history, and next are set data structures. jobs contains jobs (states) that are distributed to workers, while next contains all possible next jobs (successor states) of the next layer. history contains all states explored at
the moment. Initially, workers is set to the empty queue, while jobs, next, and jobs are set to the empty set at line 1. In the first stage, Maude-NPA needs to build rewrite rules \( R_P \) in form of a module, an attack state, and grammars from a protocol specification \( \mathcal{P} \). The module is used to perform the backward reachability analysis, the attack state is used as the beginning state, and the grammars are used to remove unreachable states for intruders. This stage is proceeded by initialize function at line 2 with \( \mathcal{P}, Id, BStep, \) and \( F \) parameters that are a protocol specification, the id of an attack state in the specification, the maximum number of backward steps, and a filter which is +parallel as default denoting the parallelization mode, respectively. The result of the function is deconstructed and stored in \( M, GS, \) and \( IS \), which stand for the module, the grammar, and the attack state, respectively. jobs and history are updated to contain only the attack state at line 3.

For each worker, whenever the master receives DATA from worker, where DATA is one of the two kinds of messages described above, it checks whether DATA is getJob, meaning that the worker is requesting a job. If so, workers is enqueued to workers so that a job can be assigned to worker subsequently. When DATA is a job that has been made and sent from worker, the master deconstructs DATA into a state IS at line 10 and then inserts it to the set of successor states next at line 11. Note that if the state already exists in the set, it is ignored. Otherwise, it is added to the set. The code fragment at lines 12-15 checks whether workers and jobs are not empty. If that is the case, the master dequeues workers and jobs to obtain a job and a worker identifier and assigns the job to the worker by sending a job message to the worker. The code fragment at lines 16-27 checks whether there are neither unprocessed jobs left nor jobs being processed by workers. If that is the case, the master continuously checks if next is not empty. If that is the case, we need to process all successor states next before moving to explore the next layer. Firstly, the successor states in next are simplified with the implication step by simplifyByImplication function at line 18 in which states implied by other states in either next or history are ignored. The output is a new set of states IST that is filtered by using history states history and rule out initial states as counterexamples by filterWithHistoryAndInit function at line 19. Ultimately, the final successor states IST' and initial states INIT are obtained. We assign jobs to IST', reset next to empty, and decrease BStep by one. Note that if BStep is unbounded, it is unchanged regardless of the subtraction, history is also updated by inserting IST' at line 21. If INIT contains some initial states, we close all connections and return INIT as counterexamples at lines 22-24. After preparing jobs for the next layer, if either BStep is 0 or jobs is empty, the tool terminates and returns empty meaning that there is no counterexample up to the depth BStep given at the beginning. Note that the three functions initialize, simplifyByImplication, and filterWithHistoryAndInit are based on existing functions provided in Maude-NPA.

Algorithm 2 shows the pseudo-code for job handling conducted by workers. Each worker maintains a set of states history to avoid sending explored states
Algorithm 2: Job Handling by Workers

\textbf{input} : \( P \) – a protocol specification
\hspace{2em} \( \text{Id} \) – an attack state id in the protocol specification
\hspace{2em} \( F \) – a filter
\hspace{2em} \( B\text{Step} \) – the maximum number of backward steps
\hspace{2em} \( N \) – a number of workers

1 \( (M,GS,IS) \leftarrow \text{initialize}(P, \text{Id}, B\text{Step}, F); \)
2 \( \text{history} \leftarrow \{IS\}; \)
3 \( \text{send}(\text{server}, \text{getJob}); \)
4 \textbf{while} isOpen() \textbf{do}
\hspace{2em} 5 \textbf{if} DATA \leftarrow \text{recv}(\text{server}) \textbf{then}
\hspace{1em} 6 \hspace{1em} IS \leftarrow \text{DATA};
\hspace{1em} 7 \hspace{1em} IST \leftarrow \text{nextBackNarrowForParallel}(M, GS, F, IS);
\hspace{1em} 8 \hspace{1em} IST' \leftarrow \text{filterWithHistory}(M, \text{history}, IST);
\hspace{1em} 9 \hspace{1em} \text{history} \leftarrow \text{insert}(\text{history}, IST');
\hspace{1em} 10 \hspace{1em} \textbf{forall} IS' \in IST' \textbf{do}
\hspace{1em} 11 \hspace{2em} JOB \leftarrow IS';
\hspace{1em} 12 \hspace{2em} \text{send}(\text{server}, JOB);
\hspace{1em} 13 \hspace{2em} \text{send}(\text{server}, \text{getJob});

by the worker to the master. Initially, we need to call \texttt{initialize} function at line 1 as the same mentioned above for the master. \texttt{history} is initially set to contain only the attack state \( IS \) at line 2. Each worker starts the job handling by sending a getJob message to the master to request a job at line 3. While the connection is open, whenever a worker receives DATA from the master, which must be a job, the worker deconstructs it into the state \( IS \) at line 6. Given \( M, GS, F \), and \( IS \), \texttt{nextBackNarrowForParallel} function performs the backward narrowing step to obtain successor states reachable from \( IS \) at line 7, which is the main task that the worker needs to do. \( IST \) is then filtered with the local cache \texttt{history} by using \texttt{filterStateWithHistory} function, which returns a new set of states \( IST' \) at line 8. \texttt{history} is then updated by inserting \( IST' \) at line 9. For each state in \( IST' \), we produce a new job and send it to the master at lines 10-12. We intend to send each job one by one to the master because it achieves the best running performance in our experiments. Once all jobs are sent to the master, the worker sends a getJob message to request a job. Note that the workers terminate if and only if the master closes all connections. The two functions \texttt{initialize} and \texttt{filterWithHistory} are based on existing functions provided in Maude-NPA.

5 Experiments

We have used a MacPro computer that carries a 2.5 GHz microprocessor with 28 cores and 1.5 TB memory to conduct experiments. We use Maude-NPA and the parallel version of Maude-NPA in our case studies. The tool and the case studies
for the experiments are publicly available at the webpage in the Footnote 4. Besides, the original source code of the cases studies and more protocols are listed at the webpage.5

We have conducted experiments on various kinds of protocols to confirm the usefulness of our parallel version of Maude-NPA such as Symmetric Key Protocols, Homomorphism Protocols, Exclusive OR Protocols, API Protocols, PKCS Protocols, Choice Protocols, and Distance-Bounding Protocols. The experimental data are shown in Tables 1–2. The first, second, and third columns denote the name of the protocols, the attack state id used in protocol specifications, and the depth bound, respectively. The fourth and fifth columns denote the verification time excluding the time taken to generate the grammars for protocols when conducting model checking with Maude-NPA and Parallel Maude-NPA, respectively. In a row, the bold value is either in the fourth column or the fifth column denoting the corresponding winning tool. The sixth column denotes the percentage of improvement when using Parallel Maude-NPA. If the value is a positive number, namely $X$, it means that the parallel Maude-NPA is $X$% faster than Maude-NPA. Conversely, if the value is a negative number, namely $-X$, it means that Maude-NPA is $X$% faster than the parallel Maude-NPA. The last column denotes the average number of states at each layer for each worker to handle, respectively. Furthermore, we inspect the number of states located at each layer for each protocol shown in Appendix A. Model checking experiments terminate as soon as counterexamples are found or the depth bound is reached.

The tool uses sockets to communicate between the master and the workers so that we can flexibly choose to use a shared-memory machine or a distributed environment. For all experiments, we use a master and eight workers with a shared-memory machine, the MacPro computer. The experimental data says that for simple case studies (24 experiments) in which the verification time is less than 40 seconds, Maude-NPA is obviously faster than the parallel Maude-NPA because the number of states located at each layer is very small and the verification time is so short that the cost of communication between the master and workers becomes burdensome. However, the parallel Maude-NPA still can finish in a reasonably short amount of time. For non-trivial case studies (35 experiments) in which the number of states located at each depth is larger, the parallel Maude-NPA has a very good performance that is 30% faster than Maude-NPA on average, demonstrating its potential. For the Diffie Hellman protocol, the percentage of improvement can be up to 49%. The average number of states at each layer for a worker is measured to let us know how busy the worker is, which reflects the number of states located at each layer. The more busy workers are and the deeper the depth bound is, the more benefit we may gain from the use of parallelization.

We select three protocols whose verification time is the largest among all protocols to conduct extra experiments with different numbers of workers used in our tool. The experimental data are shown in Table 3. The sixth column shows the number of workers used in the experiments. We can see that the

5 http://personales.upv.es/sanesro/Maude-NPA_Protocols/index.html
<table>
<thead>
<tr>
<th>Protocol</th>
<th>Attack State</th>
<th>Depth</th>
<th>Maude-NPA (seconds)</th>
<th>Parallel Maude-NPA (seconds)</th>
<th>P(%)</th>
<th>States/Layer/Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Symmetric Key Protocols</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amended Needham Schroeder</td>
<td>0</td>
<td>7</td>
<td>4588.61</td>
<td>2821.933</td>
<td>39</td>
<td>11.11</td>
</tr>
<tr>
<td>Carlen Secret Key Initiator</td>
<td>0</td>
<td>5</td>
<td>224.175</td>
<td>148.732</td>
<td>34</td>
<td>3.73</td>
</tr>
<tr>
<td>Denning Sacco</td>
<td>0</td>
<td>11</td>
<td>35.243</td>
<td>35.986</td>
<td>4</td>
<td>0.43</td>
</tr>
<tr>
<td>Diffie Hellman</td>
<td>0</td>
<td>11</td>
<td>284.211</td>
<td>158.882</td>
<td>44</td>
<td>1.56</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>286.663</td>
<td>145.412</td>
<td>49</td>
<td>1.45</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>35.371</td>
<td>21.48</td>
<td>39</td>
<td>0.32</td>
<td></td>
</tr>
<tr>
<td>ISO-5 Pass Authentication</td>
<td>0</td>
<td>5</td>
<td>101.649</td>
<td>64.306</td>
<td>37</td>
<td>2.1</td>
</tr>
<tr>
<td>Kao-Chow RA</td>
<td>0</td>
<td>4</td>
<td>52.235</td>
<td>32.904</td>
<td>37</td>
<td>2</td>
</tr>
<tr>
<td>Kao-Chow RAHK</td>
<td>0</td>
<td>4</td>
<td>4.027</td>
<td>12.151</td>
<td>-67</td>
<td>0.19</td>
</tr>
<tr>
<td>Kao-Chow RAH</td>
<td>0</td>
<td>4</td>
<td>114.414</td>
<td>77.436</td>
<td>32</td>
<td>1.94</td>
</tr>
<tr>
<td>Otway-Rees</td>
<td>0</td>
<td>4</td>
<td>72.516</td>
<td>44.91</td>
<td>38</td>
<td>2.16</td>
</tr>
<tr>
<td>Secret 06</td>
<td>0</td>
<td>2</td>
<td>1.732</td>
<td>1.874</td>
<td>-64</td>
<td>0.38</td>
</tr>
<tr>
<td>Secret 07</td>
<td>0</td>
<td>4</td>
<td>2.589</td>
<td>6.662</td>
<td>-61</td>
<td>0.28</td>
</tr>
<tr>
<td>Wide Mouthed Frog</td>
<td>0</td>
<td>3</td>
<td>16.11</td>
<td>15.563</td>
<td>3</td>
<td>1.92</td>
</tr>
<tr>
<td>Woo and Lam Authentication</td>
<td>0</td>
<td>4</td>
<td>1.371</td>
<td>5.83</td>
<td>-76</td>
<td>0.28</td>
</tr>
<tr>
<td>Yahalom</td>
<td>0</td>
<td>4</td>
<td>25.216</td>
<td>28.838</td>
<td>36</td>
<td>1.91</td>
</tr>
<tr>
<td>2. Homomorphism Protocols</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Needham Schroeder Lowe ECB</td>
<td>0</td>
<td>7</td>
<td>73.869</td>
<td>54.355</td>
<td>26</td>
<td>1.11</td>
</tr>
<tr>
<td>3. Exclusive OR Protocols</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Needham Schroeder Lowe XOR</td>
<td>0</td>
<td>5</td>
<td>10.22</td>
<td>14.771</td>
<td>-31</td>
<td>0.31</td>
</tr>
<tr>
<td>SK3</td>
<td>0</td>
<td>3</td>
<td>4.162</td>
<td>10.171</td>
<td>-59</td>
<td>0.17</td>
</tr>
<tr>
<td>TMN ltv-F-tmn-asy</td>
<td>0</td>
<td>5</td>
<td>157.442</td>
<td>110.264</td>
<td>30</td>
<td>0.88</td>
</tr>
<tr>
<td>WIRED ltv-C-wep-asy</td>
<td>0</td>
<td>5</td>
<td>14.392</td>
<td>21.335</td>
<td>-33</td>
<td>0.15</td>
</tr>
<tr>
<td>WIRED ltv-C-wep-variant</td>
<td>0</td>
<td>5</td>
<td>15.571</td>
<td>22.781</td>
<td>-32</td>
<td>0.15</td>
</tr>
<tr>
<td>4. API Protocols</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>YubiKey</td>
<td>0</td>
<td>9</td>
<td>3.487</td>
<td>10.551</td>
<td>-67</td>
<td>0.17</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>9828.475</td>
<td>65294.633</td>
<td>30</td>
<td>5.13</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>8</td>
<td>341.529</td>
<td>228.15</td>
<td>33</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>13092.864</td>
<td>10208.363</td>
<td>22</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>YubiHSM attack(d)</td>
<td>0</td>
<td>9</td>
<td>843.388</td>
<td>598.361</td>
<td>29</td>
<td>2.38</td>
</tr>
</tbody>
</table>

The average number of states at each layer for a worker is subject to the number of workers used in the experiments. When the average number of states at each layer for a worker is high, we have a chance to increase the number of workers to improve the running performance of our tool for the first two protocols, Amended Needham Schroeder and YukiKey. Up to a certain point, the more workers are used, the less busy workers are and the more burden the master needs to handle and communicate with workers that may not improve the running performance and even become worse as in the third case study, TLS attack. In addition, as mentioned above we parallelize only the first step in the second stage, but not the second step in the second stage. Hence, there is a limitation point for improvement of the first step by parallelization. Even if the number of workers is increased, it will not improve the running performance and may make the...
### Table 2: Results of Maude-NPA and Parallel Maude-NPA

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Attack State</th>
<th>Depth</th>
<th>Maude-NPA (seconds)</th>
<th>Parallel Maude-NPA (seconds)</th>
<th>P(%) States/Layer/Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>5. PKCS Protocols</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PKCS11 a1-noComp</td>
<td>0</td>
<td>4</td>
<td>24.815</td>
<td>21.489</td>
<td>13.81</td>
</tr>
<tr>
<td>PKCS11 a2-noComp</td>
<td>0</td>
<td>6</td>
<td>69.532</td>
<td>46.842</td>
<td>33.75</td>
</tr>
<tr>
<td>PKCS11 a3-noComp</td>
<td>0</td>
<td>6</td>
<td>296.424</td>
<td>201.218</td>
<td>32.16</td>
</tr>
<tr>
<td>PKCS11 a4-noComp</td>
<td>0</td>
<td>7</td>
<td>62.886</td>
<td>45.154</td>
<td>28.08</td>
</tr>
<tr>
<td>PKCS11 a5-noComp</td>
<td>0</td>
<td>9</td>
<td>382.498</td>
<td>271.263</td>
<td>29.12</td>
</tr>
<tr>
<td><strong>6. Choice Protocols</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>encryption mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>3.164</td>
<td>8.634</td>
<td>-63</td>
<td>0.28</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>8.587</td>
<td>9.855</td>
<td>-13</td>
<td>0.78</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>67.941</td>
<td>43.627</td>
<td>36</td>
<td>1.10</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>136.958</td>
<td>88.037</td>
<td>36</td>
<td>1.61</td>
</tr>
<tr>
<td>rock paper scissors</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>9</td>
<td>125.615</td>
<td>76.972</td>
<td>39</td>
<td>1.81</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.389</td>
<td>4.602</td>
<td>-92</td>
<td>0.13</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1.003</td>
<td>4.285</td>
<td>-77</td>
<td>0.38</td>
</tr>
<tr>
<td>TLS regular</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>6.727</td>
<td>15.02</td>
<td>-55</td>
<td>0.17</td>
</tr>
<tr>
<td>TLS attack</td>
<td>11</td>
<td>8695.211</td>
<td>6925.944</td>
<td>20</td>
<td>3.15</td>
</tr>
<tr>
<td><strong>7. Distance-Bounding Protocols</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>brands chaum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6.236</td>
<td>8.817</td>
<td>-29</td>
<td>0.25</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>16.186</td>
<td>16.088</td>
<td>1</td>
<td>0.29</td>
</tr>
<tr>
<td>CRCS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>766.746</td>
<td>515.919</td>
<td>33</td>
<td>0.75</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>121.77</td>
<td>101.815</td>
<td>16</td>
<td>0.42</td>
</tr>
<tr>
<td>H&amp;K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>16.792</td>
<td>14.108</td>
<td>16</td>
<td>0.35</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1.174</td>
<td>4.388</td>
<td>-73</td>
<td>0.13</td>
</tr>
<tr>
<td>MAD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>175.382</td>
<td>126.641</td>
<td>28</td>
<td>0.67</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>967.156</td>
<td>693.66</td>
<td>28</td>
<td>2.42</td>
</tr>
<tr>
<td>Meadows v1-DH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1.646</td>
<td>6.281</td>
<td>-74</td>
<td>0.13</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>32.153</td>
<td>35.344</td>
<td>-9</td>
<td>0.28</td>
</tr>
<tr>
<td>Meadows v2-DH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1.694</td>
<td>6.396</td>
<td>-74</td>
<td>0.14</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2.464</td>
<td>5.858</td>
<td>-58</td>
<td>0.17</td>
</tr>
<tr>
<td>Munilla</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>186.24</td>
<td>96.735</td>
<td>48</td>
<td>1.45</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6.279</td>
<td>9.017</td>
<td>-30</td>
<td>0.19</td>
</tr>
<tr>
<td>Swiss Knife</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6.69</td>
<td>9.384</td>
<td>-29</td>
<td>0.25</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>28.862</td>
<td>24.539</td>
<td>9</td>
<td>0.38</td>
</tr>
<tr>
<td>TREAD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6.444</td>
<td>8.962</td>
<td>-28</td>
<td>0.25</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5.166</td>
<td>8.236</td>
<td>-37</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Master busier to handle many workers at the same time. Note that there are no workers handling jobs when the second step is performed. Hence, it is significant to parallelize the second step as one piece of our future work.

In summary, the parallel version of Maude-NPA can improve the running performance of Maude-NPA effectively when dealing with most non-trivial case studies in which the number of states located at each layer is considerably large. The more states located at each layer and the deeper the search space is, the more improvement may be obtained by parallelization. For simple case studies, whose verification time is very small, for example, less than 40 seconds in our
Table 3: Parallel Maude-NPA with various numbers of workers

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Attack Depth</th>
<th>State</th>
<th>Maude-NPA (seconds)</th>
<th>#Workers</th>
<th>Parallel Maude-NPA (seconds)</th>
<th>P(%)</th>
<th>States/Layer/Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amended Needham Schroeder</td>
<td>0</td>
<td>7</td>
<td>4588.61</td>
<td>8</td>
<td>2631.933</td>
<td>.89</td>
<td>14.11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>2651.974</td>
<td>.55</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24</td>
<td>2646.479</td>
<td>.42</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td>2622.72</td>
<td>.43</td>
<td>2.78</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td>2627.64</td>
<td>.43</td>
<td>2.22</td>
</tr>
<tr>
<td>YubiKey</td>
<td>1</td>
<td>7</td>
<td>93824.875</td>
<td>8</td>
<td>65294.633</td>
<td>.30</td>
<td>1.13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>61365.349</td>
<td>.35</td>
<td>2.56</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24</td>
<td>60937.287</td>
<td>.35</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td>59652.934</td>
<td>.36</td>
<td>1.28</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td>60683.891</td>
<td>.36</td>
<td>1.03</td>
</tr>
<tr>
<td>TLS attack</td>
<td>0</td>
<td>11</td>
<td>8695.211</td>
<td>8</td>
<td>6997.392</td>
<td>.20</td>
<td>1.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>6960.781</td>
<td>.20</td>
<td>1.57</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24</td>
<td>7193.625</td>
<td>.17</td>
<td>1.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td>7220.197</td>
<td>.17</td>
<td>0.79</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td>7586.980</td>
<td>.13</td>
<td>0.63</td>
</tr>
</tbody>
</table>

In case studies, we do not need to use the parallel version of Maude-NPA, although we still can use it to obtain a reasonable result. We can see that the verification time for simple case studies is very small and so the use of the parallel version of Maude-NPA is not much different compared with Maude-NPA. Hence, it is sufficient to use solely the tool in the present paper for cryptographic protocol analysis with Maude-NPA.

6 Related Work

Our work is very close to parallel Breadth-first search algorithms. There are various parallel BFS algorithms that have been intensively studied [2,25,35,26,7]. Some of these algorithms work efficiently compared to the classical serial BFS algorithm [9, Section 22.2]. PBFS [26] uses a multiset data structure called a bag instead of a queue (FIFO). The bag supports insertion essentially as fast as FIFO and can be split and combined efficiently. In addition, for efficient implementation, PBFS contains a benign race condition in their algorithm and uses a bag reducer that allows updating concurrently to a shared variable or data structure at the same time. A bag is a collection of pennants in which each pennant is a tree of $2^k$ nodes, where $k$ is a non-negative integer. Each node in this tree contains two pointers to denote its left and right children. The bag is a crucial data structure in PBFS that is implemented efficiently in C++, while we use a set data structure that can be defined in Maude. Both Maude-NPA and Parallel Maude-NPA are written in Maude, a specification language, which is not flexible to adapt various data structures able to be implemented efficiently in the low-level, however, the idea to parallelize BFS is shared. Furthermore, we have demonstrated that the breadth-first search in Maude-NPA can be reasonably parallelized.
In addition to Maude-NPA, there are several cryptographic security analysis tools, such as Athena [34], ProVerif [6], OFMC [5], Avispa [1], Scyther [10], Verifpal [22], and TAMARIN [28]. Among them, TAMARIN, which is a prover for the symbolic analysis of security protocols, is the closest to Maude-NPA that generalizes the backward search used by the Scyther tool to support the unbounded session model, reasoning modulo equational theories, and modeling complex control flow and mutable global state. In TAMARIN, protocol specification is specified in multiset rewriting rules, while property specification is written in a guarded fragment of first-order logic. Each protocol trace corresponds to a multiset rewriting derivation that is the sequences of the labels of the applied rules. TAMARIN performs an exhaustive backward search to look for a trace that does not satisfy the property and returns a counterexample as an attack. If no rule can be applied anymore and no counterexample is found, then the protocol satisfies the property. To the best of our knowledge, our tool is the first attempt to parallelize a dedicated cryptographic security tool, Maude-NPA. Although, there are many parallel model checking algorithms for LTL [3], such as DiVinE 3.0 [4], Garakabu2 [24,23], a multicore extension of SPIN [20], and Parallel L + 1-DCA2L2MC [11].

7 Conclusion

The paper has described a parallel version of Maude-NPA and a tool that supports it. The tool has been implemented in Maude by using a master-worker model with socket communication. The paper has also reported on some experiments of various kinds of protocols in which the tool can increase the running performance of Maude-NPA by 30% on average for most non-trivial case studies where one master and eight workers are used.

There are several lines of future work as mentioned in the paper. Furthermore, we would like to use the new meta-interpreter feature in Maude rather than sockets to reduce the time taken in verifying protocols by our tool to some extent. Meta-interpreters can be run in a separate process to handle jobs independently and processes can communicate to each other by using filesystem objects on the same host instead of sockets with the TCP/IP protocol. For our experiments, if we increase the number of jobs that will be sent simultaneously between workers to the master, the running performance becomes worse because each state in Maude-NPA carries more information after each state transition for optimizing and tracing back to the initial state from the state and each state is converted to a string before sending over sockets. Hence, using sockets to convey considerably large data between workers and the master in Maude is not a good way. Therefore, the use of processes in a shared memory machine may be better than sockets in terms of communication cost. Finally, we should conduct more case studies and use various numbers of workers with the tool to demonstrate its usefulness.
References


A The Number of States Located at Each Layer

The fourth column in Tables 4–5 shows the number of states located at each layer starting from depth zero up to the depth bound for each protocol, which is a list of natural numbers separated by commas. If the last value in the list is $X$, it means that there are $X$ states located at the depth bound. Especially, if $X$ is zero, it means that there is no state for the layer. If the last value in the list is of the form $X + Y$, it means that there are $X + Y$ states located at the depth bound while $Y$ is the number of initial states (counterexamples).
Table 4: The number of states located at each layer

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Attack Depth State</th>
<th>States located at layers $(0, ..., i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Symmetric Key Protocols</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amended Needham Schroeder</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Carlsen Secret Key Initiator</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Denning Sacco</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>Diffie Hellman</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>ISO-5 Pass Authentication</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Kao-Chow RA</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Kao-Chow RAHK</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Kao-Chow RAT</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Otway-Rees</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Secret 06</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Secret 07</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Wide Mouthed Frog</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Woo and Lam Authentication</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Yahalom</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2. Homomorphism Protocols</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Needham Schroeder Lowe ECB</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>3. Exclusive OR Protocols</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Needham Schroeder Lowe XOR</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>SK3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>TMN ltv-F-tmn-asy</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>WIRED ltv-C-wep-asy</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>WIRED ltv-C-wep-variant</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>4. API Protocols</td>
<td></td>
<td></td>
</tr>
<tr>
<td>YubiKey</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>YubiHSM attack(d)</td>
<td>0</td>
<td>9</td>
</tr>
</tbody>
</table>
Table 5: The number of states located at each layer

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Attack State Depth</th>
<th>States located at layers (0, ..., i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. PKCS Protocols</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PKCS11 a1-noComp</td>
<td>0</td>
<td>1, 3, 5, 7, 9 + 1</td>
</tr>
<tr>
<td>PKCS11 a2-noComp</td>
<td>0</td>
<td>1, 2, 4, 11, 11, 4 + 1</td>
</tr>
<tr>
<td>PKCS11 a3-noComp</td>
<td>0</td>
<td>1, 3, 6, 13, 20, 21, 12 + 1</td>
</tr>
<tr>
<td>PKCS11 a4-noComp</td>
<td>0</td>
<td>1, 3, 7, 10, 10, 8, 6, 3 + 1</td>
</tr>
<tr>
<td>PKCS11 a5-noComp</td>
<td>0</td>
<td>1, 4, 11, 22, 31, 31, 15, 9, 5, 1 + 1</td>
</tr>
<tr>
<td>6. Choice Protocols</td>
<td></td>
<td></td>
</tr>
<tr>
<td>encryption mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>1, 1, 1, 2, 3 + 1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1, 2, 4, 8, 9 + 1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>1, 4, 9, 12, 15, 16, 13, 10, 6, 2, 0</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>1, 4, 10, 12, 22, 24, 21, 18, 14, 8, 2, 0</td>
</tr>
<tr>
<td>rock paper scissors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>9</td>
<td>1, 8, 16, 24, 27, 24, 18, 1, 3, 0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1, 0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1, 5, 0</td>
</tr>
<tr>
<td>TLS regular</td>
<td>0</td>
<td>1, 1, 0</td>
</tr>
<tr>
<td>TLS attack</td>
<td>0</td>
<td>1, 4, 7, 10, 14, 18, 20, 24, 29, 35, 46, 69</td>
</tr>
<tr>
<td>7. Distance-Bounding Protocols</td>
<td></td>
<td></td>
</tr>
<tr>
<td>brands chaum</td>
<td>1</td>
<td>1, 2, 3, 2, 0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1, 3, 4, 3, 1, 1, 0 + 1</td>
</tr>
<tr>
<td>CRCS</td>
<td>1</td>
<td>1, 3, 8, 16, 26, 35, 28, 14, 4, 0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1, 3, 3, 3, 6, 6, 3, 1, 0 + 1</td>
</tr>
<tr>
<td>H&amp;K</td>
<td>1</td>
<td>1, 2, 4, 5, 2, 0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1, 1, 0</td>
</tr>
<tr>
<td>MAD</td>
<td>1</td>
<td>1, 3, 7, 10, 10, 8, 3, 1, 0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1, 5, 10, 14, 18, 27, 40 + 1</td>
</tr>
<tr>
<td>Meadows v1-DH</td>
<td>1</td>
<td>1, 1, 1, 0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1, 2, 2, 3, 3, 3, 1, 0</td>
</tr>
<tr>
<td>Meadows v2-DH</td>
<td>1</td>
<td>1, 1, 1, 0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1, 1, 0 + 1</td>
</tr>
<tr>
<td>Munilla</td>
<td>1</td>
<td>1, 4, 7, 12, 22, 25, 10, 0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1, 2, 2, 1, 0</td>
</tr>
<tr>
<td>Swiss Knife</td>
<td>1</td>
<td>1, 2, 3, 2, 0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1, 4, 5, 2, 0</td>
</tr>
<tr>
<td>TREAD</td>
<td>1</td>
<td>1, 3, 2, 1, 0 + 1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1, 3, 2, 1, 0 + 1</td>
</tr>
</tbody>
</table>
Automating Safety Proofs about Cyber-Physical Systems using Rewriting Modulo SMT

Vivek Nigam2,3 and Carolyn Talcott1

1 SRI International, Menlo Park, USA, clt@csl.sri.com
2 Federal University of Paraíba, João Pessoa, Brazil, vivek.nigam@gmail.com
3 Huawei Munich Research Center, Germany

Abstract. Cyber-Physical Systems, such as Autonomous Vehicles (AVs), are operating with high-levels of autonomy allowing them to carry out safety-critical missions with limited human supervision. To ensure that these systems do not cause harm, their safety has to be rigorously verified. Existing works focus mostly on using simulation-based methods which execute simulations on concrete instances of logical scenarios in which systems are expected to function. The level of assurance obtained by these methods is, therefore, limited by the number of simulations that can be carried out. A complementary approach is to produce, instead, proofs that vehicles are safe for all instances of logical scenarios. This paper investigates how Rewriting modulo SMT applied to Soft Agents, a rewriting framework for the specification and verification of Cyber-Physical system, can be used to generate such proofs in an automated fashion. In particular, rewrite rules specify the executable semantics of systems on logical scenarios instead of concrete scenarios. This is accomplished by generating at each execution step a set of (non-linear) constraints whose satisfiability are checked by using SMT-solvers. Intuitively, a model of such set of constraints corresponds to a concrete execution on an instance of the corresponding logical scenario. We demonstrate how to specify and verify scenarios in this framework using an example involving a vehicle platoon. Finally, we investigate the trade-offs between how much of the verification is delegated to search engines (namely Maude) and how much is delegated to SMT-solvers (e.g., Z3).

1 Introduction

Autonomous Vehicles (AVs) are expected to soon reach higher-levels of autonomy, being able to drive through complex environments with no or little human supervision. To achieve this, however, it is necessary to produce a rigorous safety assurance argument [11]. An assurance strategy based on collecting data by running AVs on the streets is not feasible [12] as it would require billions of miles of data for achieving confidence in the results. Symbolic methods based on formal models have been advocated [22] as a means for safety assurance.

A safety assurance strategy begins by first identifying abstract scenarios, called logical scenarios [18], such as lane changing or platooning or pedestrian crossing, in which AVs have to avoid harm. These logical scenarios contain details
about the situations in which a vehicle shall be able to safely operate,\footnote{Also called Operational Design Domain (ODD).} such as which types and number of actors, e.g., vehicles, pedestrians, operating assumptions, e.g., range of speeds, and road topology, e.g., number of lanes. The system safety is then verified with respect to each scenario. The challenge, however, is that there are infinitely many instances for any given logical scenario.

To overcome this challenge, existing work can be divided into two different approaches. The first approach [8,15,4] is to use simulation-based methods that run a sufficiently large number of simulations using vehicle simulators [7]. A limitation of this approach is that there is possibly a large number of simulations need to be generated for each logical scenario, and even then critical situations may be missed. The second approach is to use algorithms [21,1] that are proved to generate safe trajectories under the assumption that the remaining agents behave correctly. These safe planners can then be integrated with advanced (high-performance, but not safe) controllers as fall-back options whenever safety assurance is low [6]. There are two limitations with this approach. The first limitation is that safety proofs have to be constructed manually. The second limitation is that these proofs consider only planning and not other aspects such as sensing, knowledge bases, and communication channels that are used in AV applications [4,15].

This paper’s main goal is to address the limitations of these two types of approaches by proposing a rewriting framework, based on Soft Agents [23], that enables the automated construction of vehicle level safety proofs, i.e., produce proofs that AVs are safe for all instances of a logical scenario. Such safety proofs provide greater confidence on the safety of AVs, complementing other verification evidence such as simulation-based verification techniques.

Towards achieving this goal, we make the following contributions:

- **Soft Agents Framework with Rewriting Modulo SMT:** We propose an executable symbolic Soft Agents framework [23] where instead of considering concrete values for attributes such as agent’s speed, position and acceleration, it represents these values as symbols whose possible values are specified by a set of (real non-linear) constraints. This is accomplished by extending the current Soft Agents framework with Rewriting Modulo SMT [20]. Soft Agent specifications can be executed by using Maude extensions with SMT [13]. In contrast to existing frameworks that can execute only instances of logical scenarios, symbolic soft agents can execute logical scenarios producing symbolic traces, each denoting a possibly infinite number of concrete executions of the logical scenario.

- **Vehicle Platooning Specification:** We demonstrate the Soft Agent framework by using a simple, but realistic vehicle platooning application. We illustrate how vehicle behavior and safety properties can be specified in Soft Agents, explaining how design choices may affect verification performance.

- **Verification Trade-off between Rewriting and Constraint Solving:** For the verification of systems, Soft Agents make uses of rewriting (through Maude [3]) and of SMT-solvers (through Z3 [5]). In particular, rewriting cap-

\begin{itemize}
\item \textbf{Soft Agents Framework with Rewriting Modulo SMT:} We propose an executable symbolic Soft Agents framework [23] where instead of considering concrete values for attributes such as agent’s speed, position and acceleration, it represents these values as symbols whose possible values are specified by a set of (real non-linear) constraints. This is accomplished by extending the current Soft Agents framework with Rewriting Modulo SMT [20]. Soft Agent specifications can be executed by using Maude extensions with SMT [13]. In contrast to existing frameworks that can execute only instances of logical scenarios, symbolic soft agents can execute logical scenarios producing symbolic traces, each denoting a possibly infinite number of concrete executions of the logical scenario.

\item \textbf{Vehicle Platooning Specification:} We demonstrate the Soft Agent framework by using a simple, but realistic vehicle platooning application. We illustrate how vehicle behavior and safety properties can be specified in Soft Agents, explaining how design choices may affect verification performance.

\item \textbf{Verification Trade-off between Rewriting and Constraint Solving:} For the verification of systems, Soft Agents make uses of rewriting (through Maude [3]) and of SMT-solvers (through Z3 [5]). In particular, rewriting cap-
\end{itemize}
Fig. 1. Platooning Logical Scenario: The follower vehicle $veh_f$ and $veh_l$ are in a straight lane with respectively velocities and accelerations $v_f, \alpha_f$ and $v_l, \alpha_l$. $pos_f$ is the position of front of $veh_f$ and $pos_l$ is the position of the back of the $veh_l$. We consider vehicle positions to be only the x-component increasing with as one follows to the right direction of the road. The distance between the vehicles $dist = pos_l - pos_f$.

The constraint-solver, on the other hand, generates proofs that a property is satisfied or that a property is unsatisfiable. We investigate in this paper the trade-offs between how much of verification is delegated to rewriting and how much to the constraint-solver. On the one hand, the more fine grained is the rewriting, e.g., searching with more constrained system evolutions, the greater is the number of states the search engine has to traverse leading to a greater number of calls to the SMT-solver, but the simpler are the problems that the solver has to solve. On the other hand, the more coarse is the rewriting, e.g., searching with less constrained system evolutions, the fewer are the calls to the SMT-solver, but the larger are problems that the constraint solver has to solve. Our experiments indicate that these trade-offs need to be considered in order to verify more challenging properties.

Plan. We start in Section 2 by describing a motivating example: a logical scenario from a vehicle platooning case study, which is used as running example. Section 3 introduces symbolic rewriting, then recalls the soft agents framework and its generalization to to symbolic rewriting with SMT solving. Section 4 presents key elements of the symbolic vehicle platooning logical scenario, including control decisions, safety properties, and search patterns for reachability analysis. Section 5 presents experiments evaluating trade-offs between size of search space and complexity of constraints to solve. We conclude by discussing related work in Section 6 and future work in Section 7.

2 Motivating Example

Our motivating example is a platooning scenario which is a typical Level 3 autonomy\(^5\) use-case. This scenario takes place in a highway as illustrated by Figure 1. The vehicle $veh_f$, called follower vehicle, follows autonomously, i.e.,

\(^5\) For the levels of autonomy, see the SAE classification described in [10]
only with human supervision, vehicle $\text{veh}_l$, called leader vehicle. The vehicles are driving in a highway lane and therefore are expected to have a speed within some given range of values normally obtained by considering legal speeds and the vehicle's capabilities, e.g., speeds between 60 km/h and 130 km/h. Moreover, the acceleration (and deceleration) capabilities of the vehicles are also bounded, typically between $-8 m/s^2$ and $2 m/s^2$.

The goal of the follower vehicle is to maintain a safe distance to the leader vehicle, but still be close enough to profit from the wind shadow of the leader vehicle yielding up to 17% of fuel savings [25]. Since the speed of the vehicles may vary, it is not appropriate to define a safe distance as an absolute quantity, but in terms of time to react. That is, the distance will depend on the relative speeds of the vehicles.

As an example, building on ideas from [6], we define the following three properties for the platooning logical scenario:

\[
P_{\text{safer}} := \text{dist} \geq v_f \times (1[s] + \text{gap}_\text{safer}) - v_l \times 1[s],
\]

\[
P_{\text{safe}} := v_f \times (1[s] + \text{gap}_\text{safer}) - v_l \times 1[s] > \text{dist} \geq v_f \times (1[s] + \text{gap}_\text{safe}) - v_l \times 1[s],
\]

\[
P_{\text{unsafe}} := \text{dist} < v_f \times (1[s] + \text{gap}_\text{safe}) - v_l \times 1[s]
\]

Intuitively, their satisfaction is conditional on the distance (dist) between the vehicles; their speeds ($v_l$ and $v_f$); and the parameters $\text{gap}_\text{safer}$ and $\text{gap}_\text{safe}$ which are time to react parameters, typically a few seconds. Moreover, $\text{gap}_\text{safer} > \text{gap}_\text{safe}$, which means that the instance of a logical scenario satisfies $P_{\text{safer}}$ (or simply safer) if the vehicles $\text{veh}_l$ and $\text{veh}_f$ have a greater distance between them. Finally, an instance of a logical scenario satisfies $P_{\text{unsafe}}$ (or simply unsafe) if distance is too small to satisfy $P_{\text{safe}}$ or $P_{\text{safer}}$.

A description of the function of a vehicle, such as platooning, using formal notations and ranges of parameters is called a logical scenario [14]. The objective is to prove that an implementation of a controller for the platooning function is safe, that is either $P_{\text{safer}}$ or $P_{\text{safe}}$ is satisfied for all concrete instances of this logical scenario. This is challenging as there are infinitely many such instances.

3 Symbolic Soft Agents Framework

We begin with an overview of challenges in modeling cyber-physical systems (CPSs), then recall the main features of soft agent specifications, and then briefly discuss the generalization to symbolic form.

3.1 Overview

A soft agent (SA) model of a CPS makes explicit both discrete changes (cyber actions, control settings) and continuous change (in the physical environment). Following ideas developed in Real Time Maude [17], soft agent models have instantaneous rules that specify agents decision processes that generate actions
such as communication or setting control parameters; and a timeStep rule that models the passage of some interval of time, updating the state according to a model of the time-dependent aspects of the state.

In contrast to the usual realtime specifications, soft agent CPS specifications involve variables, such as speed, distance, etc, that are dense and their evolutions over time are not discrete events. Moreover, system properties, such as safety properties, are expressed using these variables, e.g., keeping a given distance to the vehicle ahead rather than timing properties such as network delay or execution time. Verification of safety properties for CPS specifications involves reasoning about possibly infinitely many states and properties whose parameters may change continuously over time.

Two challenges for safety analysis of CPS specifications are (1) soundness of discrete time sampling execution; and (2) checking for reachability of unsafe states from a possibly infinite set of instances of a logical scenario. Challenge (1) includes choosing the timestep intervals small enough so that no unsafe situations are missed, while not being so fine grained that the state space becomes unmanageable. This is a design time concern, for example choosing the frequency with which sensors are read and control settings are updated. The latter challenge (2) involves the coverage and state space management with time properties.

Real Time Maude addresses (1) in [16], defining conditions on a timed rewrite theory that guarantee soundness and completeness of model checking based on maximal time elapsed discrete time sampling. Unfortunately, soft agent analysis problems generally do not meet these conditions. Narrowing is one approach to checking reachability from a possibly infinite initial set of system states. Maude supports narrowing modulo a rich collection of equational theories, but narrowing using conditional rules is not supported [3], and soft agent relies heavily on conditional rules.

New ideas are needed to address the verification challenges. We propose a form of symbolic rewriting that combines rewriting and constraint solving.

1. We represent logical scenarios as symbolic system states, representing a set of concrete states. A logical scenario consists of a pattern (a term with pattern variables called symbols) together with a set of constraints on values of the symbols.\(^6\)

2. A symbolic rewrite rule introduces new symbols and additional constraints representing new values of the pattern variables. The resulting logical scenario represents the instances reachable from instances of the starting pattern using the rewrite rule.

3. Symbolic rule conditions use symbolic function evaluation to generate new symbols and their constraints.

The point of symbolic analysis is to check properties of concrete systems represented by concrete scenarios. Thus we want to connect symbolic executions to concrete executions. The concrete executions may be obtained from a concrete

---

\(^6\) Mathematically, a logical scenario is a term with variables. To be able to rewrite logical scenarios in Maude, we replace variables by symbols, which formally are uninterpreted constants.
form of the rewrite rules, or simply using the symbolic rules with grounding constraints of the form, \( \text{sym} == \text{ground term} \).

To describe the desired symbolic-concrete connection, we need a little notation. The basic idea is analogous to that presented in [19]. We assume a rewrite theory \( \mathcal{T} = (\Sigma, B \cup E, R) \) with signature \( \Sigma \), axioms \( B \), equations \( E \), and rules \( R \). Assume further an equational subtheory \( \mathcal{T}_0 \) of \( \mathcal{T} \) axiomatizing the theory in which the constraints are solved by the SMT solver. We use \( sS, sS_0, sS_1, \ldots \) to denote logical scenarios (symbolic states) and \( cS, cS_0, cS_1, \ldots \) to denote concrete states (ground states with no symbols). Let \( \sigma, \sigma_0, \sigma_1, \ldots \) denote substitutions mapping symbols to concrete terms (values). A logical scenario is structured as a pair \( (sP, sC) \) consisting of a pattern, \( sP \), and a constraint, \( sC \), on the symbols of \( sP \). \( sC \) represents a quantifier free formula in the language of \( \mathcal{T}_0 \).

Application of a substitution, \( \sigma \), to a logical scenario, \( sS = (sP, sC) \) (written \( (\sigma sP) \)), gives an instance of \( sS \) if the domain of \( \sigma \) contains all the symbols of \( sS \) and \( \sigma \) satisfies \( sC (\mathcal{T}_0 = sC\sigma) \). We say \( \sigma_1 \) extends \( \sigma_0 \), written \( \sigma_1 \gg \sigma_0 \) if the domain of \( \sigma_1 \) contains the domain of \( \sigma_0 \) and \( \sigma_0(v) = \sigma_1(v) \) (wrt. \( \mathcal{T} \)) for \( v \) in the domain of \( \sigma_0 \). Finally, we let \( \rightarrow_c \) denote the concrete rewrite relation induced by \( \mathcal{T} \), and \( \rightarrow_s \) denote the symbolic rewrite relation induced by \( \mathcal{T} \). Then the desired connection between the rewrite relations is given by the following Soundness and Completeness properties. These correspond to Theorems 1 and 2 of [19] and can be proved by analogous arguments.

**Soundness.** If \( sS_0 \rightarrow_c sS_1 \) and \( \sigma_0 \) gives an instance of \( sS_0 \), then there exists \( \sigma_1 \gg \sigma_0 \) such that \( cS_1 \) is equivalent (in \( \mathcal{T}_0 \)) to \( sP_1\sigma_1 \) and and \( \sigma_0(sP_0) \rightarrow_c cS_1 \).

**Completeness.** If \( \sigma_0 \) gives an instance of \( sS_0 \) and \( \sigma_0(sP_0) \rightarrow_c cS_1 \) then there exists \( sS_1 \), and \( \sigma_1 \gg \sigma_0 \) such that \( \sigma_1 \) gives an instance of \( sS_1 \) with \( cS_1 \) equivalent to \( \sigma_1(sP_1) \) and \( sS_0 \rightarrow_s sS_1 \) where \( \sigma_1 \) gives an instance of \( sS_1 \).

### 3.2 The structure of Soft Agent Rewriting

In soft agents, a system state consists of a set of agent terms together with a unique environment term. Abstractly an agent term has the form \( A(id, attrs) \) where \( id \) is the agent identifier, and \( attrs \) is a set of named attributes including the agents local knowledge base (local KB), and a set of pending tasks and actions each labeled by the time until ready for execution. An environment term has the form \( E(ekb) \) where \( ekb \) is a knowledge base representing the physical state of the system and contextual information such as location of features or bounds on location.

There are two rewrite rules: **doTask** and **timeStep**. The **doTask** rule has the form

\[
\text{crl[doTask]}: A(id, attrs) E(ekb) \Rightarrow A(id, attrs') E(ekb) \text{ if taskConds}
\]

where **taskConds** has clauses for reading sensors from the environment, evaluating possible actions, and updating the local KB, pending tasks, and actions. The **timeStep** rule has the form
crl[timeStep]: A(id1,attrs1) ... A(idk,attrsk) E(ekb) =>
A(id1,attrs1') ... A(idk,attrsk') E(ekb') if stepConds

where stepConds has a clause to execute ready actions (with time delay 0) and
update time-dependent symbols to capture the passing of time. There are also
clauses to update time parameters (clocks, delays...), transmit messages, and
share knowledge amongst the agents. Executing actions affects parameters that
control how the physical state evolves (change of acceleration, direction, on/off
switches ...). Passing time lets the physical model run for the specified interval
of time, updating the physical state (position, energy level, ...) according to
laws parameterized by the control settings.

3.3 Symbolic Soft Agent Rewriting

To enable symbolic execution of soft agent specifications we abstract system
states as terms of the form $SA[uu] \ SE[vv]$ where $SA$ is a pattern with symbols
$uu$ whose structure captures the state aspects that are not changed during ex-
ecution, for example the number of agents, their ids, attribute names, and any
persistent structure in attribute values. Similarly, $SE[vv]$ is a pattern, with sym-
 bols $vv$ capturing the persistent structure in the environment knowledge base.
$uu$ and $vv$ are disjoint lists of symbols. For example, in a platooning scenario,
symbols in $vv$ would represent values including the position, acceleration, and
velocity of each vehicle. Mathematically, we represent the symbolic constraint
as a separate state component. In practice, we represent it as an element of the
environment knowledge base.

Intuitively, the execution of a logical scenario constructs new constraints con-
taining fresh symbols representing new values of the system’s physical attributes.
As for (concrete) soft agents, there are two rewrite rules for symbolic soft agents.
At the framework level, the symbolic rules are obtained by replacing the clauses
in the rule conditions of concrete rules by symbolic versions that refer to sym-
 bolic versions of the functions involved. It is the job of the specifier to define
these symbolic functions and their symbolic evaluation equations. In the vehicle
platooning case, symbolic functions were obtained by systematically transform-
ing the original concrete versions. In the next section we give examples of key
elements of the symbolic vehicle platooning system.

4 Vehicle Specifications

This section details how one can specify logical scenarios including safety prop-
erties by specifying the vehicle platooning example described in Section 2. While
the specifications below are declarative, i.e., closely resemble textbook formulas,
we do assume that the reader is familiar with the Maude syntax [3]. Our start-
ing point is a concrete specification of the vehicle platooning described in [4]. It
contains several features, such as vehicle controllers and communication proto-
col specifications, which have been ported to the symbolic machinery described
below. The complete code can be found at https://github.com/SRI-CSL/VCPublic.git in the folder symbolic-platooning. To execute this code you will need the Maude integration with Z3 which can be found at [13].

4.1 Basic Symbolic Sorts

RealSym is the sort of real values. It contains concrete values, i.e., real numbers, or symbols of the form vv(i) or vv(i,str) where i is a Nat uniquely identifying a symbol and str is a string describing the intuitive meaning of the symbol, used for improved readability. The term mkNuVar(i,id,str) evaluates to a (fresh) symbol with identifiers id, str, where id is an agent identifier and str is a string with a short description of the fresh symbol.

Example 1. The following symbols represent the initial conditions for the follower ag1, namely, its position, speed, maximum acceleration, maximum deceleration, and initial acceleration.

\[
\begin{align*}
\text{eq } v1posx &= vv(2, "ag1-positionX") . \\
\text{eq } v1posy &= vv(3, "ag1-positionY") . \\
\text{eq } v1vel &= vv(5, "ag1-speed") . \\
\text{eq } maxacc1 &= vv(9, "ag1-maxAcc") . \\
\text{eq } maxdec1 &= vv(10, "ag1-maxDec") . \\
\text{eq } acc1 &= vv(32, "ag1-acc") .
\end{align*}
\]

SymTerm is the sort of symbolic terms containing arithmetic expressions constructed inductively using basic arithmetic operators (e.g., addition, subtraction, division, multiplication) and elements of RealSym. They are used to specify constraints of sort Boolean involving symbols.

Example 2. The following constraint using the symbols in Example 1 specifies that ag1’s acceleration is bounded by the maximum acceleration and deceleration: \((\text{acc1} \leq \text{maxacc1}) \land (\text{acc1} \geq \text{maxdec1})\)

4.2 Knowledge Specifications

Cyber-physical systems reason using knowledge about their locations, speeds, direction, and accelerations and of the surrounding objects. Such knowledge is represented using a sort Info. Knowledge base elements are of the form info @ t where t is a logical time, i.e., the number of time steps since the beginning.

Vehicle locations are two-dimensional, speeds are real values, and directions are vectors specified using two locations and a magnitude:

\[
\begin{align*}
\text{op } \text{loc} &: \text{SymTerm} \times \text{SymTerm} \to \text{Loc} . \\
\text{op } \text{speed} &: \text{Id} \times \text{RealSym} \to \text{Info} . \\
\text{op } \text{dir} &: \text{Id} \times \text{Loc} \times \text{Loc} \times \text{SymTerm} \to \text{Info} .
\end{align*}
\]

Example 3. The agent ag1’s initial knowledge base, that is, at logical tick 0, contains the following terms, specifying its initial position, speed, acceleration and direction:

\[
\begin{align*}
\text{(at}(\text{ag1}, \text{loc}(v1posx,v1posy)) @ 0) \quad \text{(speed}(\text{ag1},v1vel) @ 0) \\
\text{(accel}(\text{ag1},\text{acc1}) @ 0) \\
\text{(dir}(v(1),\text{loc}(v1ix,v1iy),\text{loc}(vltx,vlty),\text{v1mag}) @ 0)
\end{align*}
\]
Based on the above notation, we can specify symbolically typical definitions, such as the distance between two locations:

```
eq ldist(i,loc(x0,y0),loc(x1,y1))
   = {s(i),vv(i,"dist"), (vv(i,"dist") >= 0/1) and
       vv(i,"dist") * vv(i,"dist") === ((y1 - y0) * (y1 - y0) +
       (x1 - x0) * (x1 - x0)) } .
```

This definition creates a fresh symbol, `vv(i,"dist")` together with the constraint specifying the Euclidean distance. Notice that we need to specify that the distance is a non-negative value. Similar specifications can be made for other distance measures, such as, Manhattan distance.

The following operator specifies how an agent’s location, `loc(x,y)`, is updated to `loc(nuVarX,nuVarY)` given an (average) speed, `spd`, and a direction.

```
op upVLoc : Nat Id Loc SymTerm Info -> NatLocBoolean .
ceq upVLoc(i,id,loc(x,y),spd,dir(id,loc(x0,y0),loc(x1,y1),mag))
   = {i + 2,loc(nuVarX,nuVarY),cond}
if nuVarX := mkNuVar(i,id,"-positionX")
   \ con1 := (x0 === x1) and (not (y0 === y1)) and
           (nuVarX === x) and (nuVarY === y + spd)
   \ con2 := (not (x0 === x1)) and (y0 === y1) and
           (nuVarX === x + spd) and (nuVarY === y)
   \ con3 := (not (x0 === x1)) and (not (y0 === y1)) and
           (nuVarX === (x + spd * (x1 - x0) / mag)) and
           (nuVarY === (y + spd * (y1 - y0) / mag))
   \ con := con1 or con2 or con3 .
```

We made some design choices in this definition. The first design choice is to split it into three different cases. The first case (`con1`) is when the agent is moving vertically, the second case (`con2`) horizontally, and the third case (`con3`) when it is moving in the quadrant. In this way we help the constraint solver to avoid to solve the harder non-linear constraint involved in the third case whenever the agent is moving only along the x-axis and only along the y-axis. The second design choice was to include the magnitude in the definition of `dir` which may seem redundant as it can be specified from the two associated locations. However, by doing so, we avoid the need to generate fresh symbols and new constraints whenever the magnitude is needed as in the third case of `upVLoc`.

Finally, we also capture symbolically the fact that the physical system is continuous while the cyber part of the system works in logical ticks. The size of the tick is specified by the term `tickSize(dt)`, where `dt` is symbol denoting the size of the tick. Typically it is fixed during the whole execution by using a constraints, e.g., `dt === 1/10`, specifying a tick duration of 100ms. We assume here for simplicity that all agents use the same tick duration. However, agents with different tick duration can also be specified. When the soft agent machinery updates the agent’s positions using `upVLoc` it scales accordingly the speed to the tick size by multiplying the speed with `dt`. 

172
4.3 Soft-Constraint Controller

Agents decide which action to take based on their local knowledge base, which is updated by reading their sensors, and taking into account different concerns, such as safety and efficiency. For vehicle platooning, as described in detail in [4], there are two main concerns, \textit{safety}, i.e., maintaining a safe distance between vehicles, and \textit{fuel-efficiency}, i.e., maintaining a distance between vehicles that is not too great.

The controller is specified in a similar way to the knowledge functions described above by using existing symbols, creating new symbols, and using constraints to determine its possible values.

The following equation specifies the controller evaluation to rank the possible actions that the vehicle can take from a safety perspective. In particular, it takes as input $i$, for creating fresh symbols, $v_{\text{min}}, v_{\text{max}}$, respectively, the minimum and maximum speeds that the vehicle is allowed to use, $v_{\text{minD}}, v_{\text{maxD}}$, the minimum and maximum desired speeds according to the safety parameters ($\text{gap}_{\text{safe}}, \text{gap}_{\text{safety}}$), and the constraints $\text{cond}$ on the existing symbols. It then returns a range of speeds that are safe specified by the interval between the fresh symbols $v_{\text{i}}$ and $v_{\text{i} + 1}$. However, the concrete values for $v_{\text{i}}$, $v_{\text{i} + 1}$ depend on the relation between the possible speeds ($v_{\text{min}}, v_{\text{max}}$) and the desired speeds $v_{\text{minD}}, v_{\text{maxD}}$ as detailed by the constraints $\text{cond11}, \text{cond21}, \ldots, \text{cond61}$.

\begin{verbatim}
4.3 Soft-Constraint Controller

Agents decide which action to take based on their local knowledge base, which is updated by reading their sensors, and taking into account different concerns, such as safety and efficiency. For vehicle platooning, as described in detail in [4], there are two main concerns, \textit{safety}, i.e., maintaining a safe distance between vehicles, and \textit{fuel-efficiency}, i.e., maintaining a distance between vehicles that is not too great.

The controller is specified in a similar way to the knowledge functions described above by using existing symbols, creating new symbols, and using constraints to determine its possible values.

The following equation specifies the controller evaluation to rank the possible actions that the vehicle can take from a safety perspective. In particular, it takes as input $i$, for creating fresh symbols, $v_{\text{min}}, v_{\text{max}}$, respectively, the minimum and maximum speeds that the vehicle is allowed to use, $v_{\text{minD}}, v_{\text{maxD}}$, the minimum and maximum desired speeds according to the safety parameters ($\text{gap}_{\text{safe}}, \text{gap}_{\text{safety}}$), and the constraints $\text{cond}$ on the existing symbols. It then returns a range of speeds that are safe specified by the interval between the fresh symbols $v_{\text{i}}$ and $v_{\text{i} + 1}$. However, the concrete values for $v_{\text{i}}, v_{\text{i} + 1}$ depend on the relation between the possible speeds ($v_{\text{min}}, v_{\text{max}}$) and the desired speeds $v_{\text{minD}}, v_{\text{maxD}}$ as detailed by the constraints $\text{cond11}, \text{cond21}, \ldots, \text{cond61}$.

\begin{verbatim}
\begin{align*}
\text{ceq symValSpeedRed}(i, \text{str}, v_{\text{min}}, v_{\text{max}}, v_{\text{minD}}, v_{\text{maxD}}, \text{cond}) &=
\{i + 2, [v_{\text{i}}, v_{\text{i} + 1}], \text{nuCond and cond}\}
\text{if cond1} := v_{\text{min}} >= v_{\text{maxD}}
\text{\quad \wedge cond11} := v_{\text{i}} === v_{\text{min}} \text{ and } v_{\text{i} + 1} \text{ is safe}
\text{\quad \wedge cond2} := v_{\text{max}} <= v_{\text{minD}}
\text{\quad \wedge cond21} := v_{\text{i}} === (v_{\text{min}} + v_{\text{max}}) / 2\text{ and } v_{\text{i} + 1} === v_{\text{max}}
\text{\quad \wedge nuCond} := (\text{cond11 or cond21 or cond31 or cond41 or cond51 or cond61}).
\end{align*}
\end{verbatim}

In the definition above, the effort of determining which condition applies is delegated to the constraint solver. As we will investigate in Section 5, this will lead to great performance penalties.

An alternative way to expressing the same controller is to return six possibilities as specified by the following equation, rather than the single disjunction $\text{nuCond}$:

\begin{verbatim}
\begin{align*}
\text{ceq symValSpeedRed-Split}(i, \text{str}, v_{\text{min}}, v_{\text{max}}, v_{\text{minD}}, v_{\text{maxD}}, \text{cond}) &=
\{i + 2, [v_{\text{i}}, v_{\text{i} + 1}], \text{cond11 and cond}\}
\{i + 2, [v_{\text{i}}, v_{\text{i} + 1}], \text{cond21 and cond}\}
\ldots
\{i + 2, [v_{\text{i}}, v_{\text{i} + 1}], \text{cond61 and cond}\}
\text{if cond1} := v_{\text{min}} >= v_{\text{maxD}}
\text{\quad \wedge cond61} := \text{nuCond}
\end{align*}
\end{verbatim}

In the definition above, the effort of determining which condition applies is delegated to the constraint solver. As we will investigate in Section 5, this will lead to great performance penalties.

An alternative way to expressing the same controller is to return six possibilities as specified by the following equation, rather than the single disjunction $\text{nuCond}$:

\begin{verbatim}
\begin{align*}
\text{ceq symValSpeedRed-Split}(i, \text{str}, v_{\text{min}}, v_{\text{max}}, v_{\text{minD}}, v_{\text{maxD}}, \text{cond}) &=
\{i + 2, [v_{\text{i}}, v_{\text{i} + 1}], \text{cond11 and cond}\}
\{i + 2, [v_{\text{i}}, v_{\text{i} + 1}], \text{cond21 and cond}\}
\ldots
\{i + 2, [v_{\text{i}}, v_{\text{i} + 1}], \text{cond61 and cond}\}
\text{if cond1} := v_{\text{min}} >= v_{\text{maxD}}
\text{\quad \wedge cond61} := \text{nuCond}
\end{align*}
\end{verbatim}

173
With this new definition the choice of which condition is applicable is left to the search engine, i.e., Maude.

A similar choice occurs when specifying how the time advancement affects agent’s speeds. Several cases occur due to the fact that logical scenarios assume that vehicle’s speeds are bounded. For example, depending on the tick duration, current speed and maximum acceleration, an agent’s speed may reach the maximum speed or not before completing a logical tick. For analyzing the impact of delegating such enumeration of cases to the SMT-solver or to the search engine, we implemented two versions of time advancement: `timestep` that returns one output with a constraint with a disjunct for each case, as in `symValSpeedRed`; and `timestep-split` that returns several outputs, one for each possible case as `symValSpeedRed-split`.

### 4.4 System Configurations

As described in Section 3, a system configuration of sort `ASystem` is a collection of agent configurations and an environment configuration.

An agent configuration has the form `[id : class | attrs ]`, where `id` is the agent’s unique identifier, `class` is its class, e.g., vehicle, and `attrs` are its attributes which include its local knowledge base written `lkb : kb`, where `lkb` is a label and `kb` is the local knowledge base contents.

An environment configuration has the form `[eId | ekb]` where `ekb` is the environment knowledge base which specifies state of the world. The environment knowledge base contains the knowledge item `constraints(i,cond)` where `i` is the current index of fresh variables, and `cond` is the constraints (accumulated) on the existing symbols.

**Example 4.** The initial configuration of a platooning scenario described in Section 2 is as follows:

\[
\text{asysI} = \{ \text{[eId | (kb constraint(i,condI))] } \\
\text{[v(0) : veh | lkb : kb0 ] [v(1) : veh | lkb : kb1 ] } \}
\]

where `kb` is the environment knowledge base specifying among other things, the vehicle’s actual locations and speeds, while `kb0` and `kb1` are the vehicle `v(0)` and `v(1)`’s local knowledge bases. The constraint `condI` contains the constraints on these values as per the logical scenario. It contains for example constraints on the acceleration of vehicles (see Example 2) and the following constraints:

\[
(v1vel >= vellb1) and (v1vel <= velub1) and (v0posy > v1posy)
\]

which specify that the follower vehicle’s speed is bound within the bounds `vellb1` and `velub1`. Moreover, the following vehicle `v(1)` is behind the leader `v(0)`.

Notice that such a symbolic system configuration may correspond to infinitely many concrete system configuration, i.e., concrete instances of the specified platooning scenario.
4.5 Safety Properties

We are interested in generating proofs regarding the safety of logical scenarios, such as the one specified in Example 4. The specification of safety property is formalized using the operator:

\[
\text{op mkSPCond : SP ASystem} \rightarrow \text{SPSpec}.
\]

This function takes a property (an identifier in \(\text{SP}\)) and a system configuration, and returns a safety property of sort \(\text{SPSpec}\) of the form:

\[
\text{op \{_,_,_,\} : Nat SymTerms Boolean Boolean} \rightarrow \text{SPSpec}.
\]

The first element is the new symbol index, the second is the new (auxiliary) symbols created for specifying the property, which are then constrained by the third element. The last element specifies the safety property based on the auxiliary symbols and the previously existing symbols in the given system configuration.

For example, the first safety property in Eq 1b is specified as follows:

\[
\begin{align*}
\text{ceq mkSPCond(saferSP, \{ conf env \})} &= \{ k + 1, \text{dis,cond00,nucond} \} \\
\text{if [id0 | kb] := env} \\
&/\ (\text{atloc(v(0),10) @ t0}) (\text{atloc(v(1),11) @ t1}) \\
& (\text{speed(v(0),v0) @ t2}) (\text{speed(v(1),v1) @ t3}) \\
& (\text{gapSafety(v(1),gapSafer,gapSafe)}) (\text{constraint(n,cond)}) \text{ kb1 := kb} \\
&/\ \{k,\text{dis,cond00}\} := \text{ldist(n,11,10)} \\
&/\ \text{nucond := (dis >= ((1/1 + gapSafer) * v1) - v0)}.
\end{align*}
\]

Notice the use of the function \(\text{ldist}\) that creates the auxiliary fresh symbol \(\text{dis}\).

Using \(\text{mkSPCond}\), we specify an operator (definition elided)

\[
\text{op enforceSP : SP ASystem} \rightarrow \text{ASystem}.
\]

For example, \(\text{enforce(saferSP,asysI)}\) returns a configuration in which the conditions (\text{cond00} and \text{nucond} from \(\text{mkSPCond}\)) are added to the set of constraints. This means that the resulting configuration will only have instances \(\text{asysI}\) that satisfy the \(\text{saferSP}\). The term \(\text{IsSatModel(enforce(saferSP,asysI))}\) calls the SMT-Solver and returns an assignment for \(\text{asysI}\) symbols:

\[
\begin{align*}
\text{ag0-positionX} & \rightarrow (0/1).\text{Real}, \quad \text{ag0-positionY} \rightarrow (1/1).\text{Real} \\
\text{ag1-positionX} & \rightarrow (0/1).\text{Real}, \quad \text{ag1-positionY} \rightarrow (0/1).\text{Real}, \\
\text{ag0-speed} & \rightarrow (7/1).\text{Real}, \quad \text{ag1-speed} \rightarrow (2/1).\text{Real}, \\
\text{ag1-safer} & \rightarrow (3/1).\text{Real}
\end{align*}
\]

This state satisfies the \(\text{saferSP}\) property for a \(\text{gap_{safer}}\) of value 3.

4.6 Verifying Logical Scenarios

We can now use Rewriting Modulo SMT [20] to verify and effectively generate safety proofs of the specifications above in an automated fashion. Consider the following search:
search enforceSP(safeSP,setStopTime(asysI,2)) =>*
asys such that checkSP(unsafeSP,asys) .
No solution. states: 63 rewrites: 394686 in 20134ms

It attempts to find any instance of system configuration asys that satisfies unsafeSP (see Eq. 1c) starting from any instance of asysI that satisfies property safeSP. Moreover, the term setStopTime(asysI,2) specifies that the search is bound to two logical ticks, i.e., search stops after two tick rules. The search engine combined with the SMT-solver can generate proofs that no instance of reachable states are unsafe. However, as shown Section 5, the complexity of the problem greatly increases when considering larger logical tick bounds.

5 Trade-offs Between Rewriting and Constraint Solving

The verification of logical scenario involves rewriting and constraint solving. Rewriting enumerates possible system states while the constraint solver attempts to check the satisfiability of constraints. As demonstrated in Section 4.3, how much of verification is delegated to rewriting and how much to the constraint solver can be adjusted by leaving the non-determinism in the constraints, e.g., by placing disjunctions in the constraints, or to the rewriting, e.g., returning instead for each disjunct an output, a rewriting choice.

Delegating verification to the rewriting engine means that the search tree is larger leading to more calls to the SMT-solver, but each call involves simpler constraints to solve, i.e., with less disjunctions and therefore less cases to consider. Delegating verification to the constraint solver, on the other hand, means a smaller search space traversed by the rewriting engine leading to less calls to the constraint solver, but with more complex constraints.

To demonstrate this, we considered three cases according to the specifications described in Section 4.3:

- **More SMT Less Search:** This case uses symValSpeedRed for the controller and timestep for the time step evolution. This means that all cases are specified as disjunctions in the constraint that will need to be solved by the solver.

- **Less SMT More Search:** This case uses symValSpeedRed-split for the controller and timestep-split for the time step evolution. This means that all cases are specified as different outputs that need to be traversed by the rewriting engine.

- **Balanced:** This case uses symValSpeedRed for the controller and the specification timestep-split for the time step evolution. This means that some cases are specified as constraints and others as outputs.

To evaluate the different cases, we executed the command:

search enforceSP(safeSP,setStopTime(asysI,Bound)) =>! asys
such that isSat(asys) .

which enumerates all the reachable symbolic configurations that are satisfiable exactly in Bound time ticks, i.e., number of applications of the timeStep rule.
A second dimension that we investigated was on the way we can prune the search tree. We considered the following cases:

- **All Pruning:** At each rewrite rule for doTask, which evaluates an agent’s actions, and tick, which applies the agent’s actions, we placed a check whether the resulting configuration is satisfiable. This means that the search tree has only satisfiable configurations with the price of calling the SMT-Solver at each step.

- **No Pruning:** As opposed to the All Pruning case, rewrites doTask nor tick did not check the satisfiability of the resulting configuration. The check was made only at the configuration resulting from applying the number to ticks specified by the bound. This means that the search tree is not pruned, and therefore, more states are traversed.

- **Tick Pruning:** The third case does a check on the configuration resulting from timeStep rewrites, but not on doTask. In this way, we still prune the search tree without calling the SMT-solver at each rewriting step.

Table 1 summarizes our experiments with these scenarios using bounds of two and three cs. The best case was not pruning the tree and delegating verification to the search tree when considering greater time bounds. The balanced case had better results when considering lower time bounds.

Interestingly, pruning the tree, while had a great effect on number of states, it did not improve the time required to traverse the tree. We believe that this can be further improved if the search engine uses the SMT-solver in a more clever way, in particular, using its incremental solving features. This would allow the solver to re-use work done in previous calls.

### 6 Related Work

Existing work for the verification of autonomous cyber-physical systems can be divided into three different approaches.
The first approach [8] is to use simulation-based methods that run a sufficiently large number of simulations using simulators [7]. A main advantage of this approach is that it can be used to verify the actual artifacts, e.g., machine learning artifacts, used in applications and rely on vehicle simulators to generate very complicated and high-fidelity scenarios. However, as already mentioned, as each simulation is run using a concrete instance of a logical scenario, a limitation of this approach is that possibly a large number of simulations need to be generated for each logical scenario. Our work complements this work by enabling the specification and verification of vehicle behavior using symbolic methods covering all instances of a logical scenario, and enables early verification of designs before expensive artifacts are built.

The second approach is to use safe controllers [21,1] that are guaranteed to generate safe trajectories under the assumption that the remaining agents behave correctly. A limitation of this type of work is that it focuses only on individual functions, typically control algorithms without taking into account other functions needed for AVs, e.g., sensing, knowledge bases, and communication channels. As shown in [6], safe controllers can be integrated with advanced (high-performance, but not safe) controllers as fall-back options whenever safety assurance is low. In particular, a formal framework for Run Time Assurance (RTA) is presented, and conditions are given that, if satisfied by a safe controller and associated monitor, guarantee that integration with an untrusted control maintains safe operation. The paper leaves open methods to verify that a controller satisfies its RTA requirements. Our work has been greatly inspired by [6] and the result is complimentary. Symbolic rewriting combined with SMT solving provides automated methods to verify correctness of time sampling mechanisms and safety requirements.

The third approach [15,24,17], similar to the non-symbolic Soft Agents, are formal frameworks that enable the specification and verification of other functions, besides trajectory planning [9,4]. However, as with the first approach, the evidence that can be produced by these frameworks is based on running simulations or model checking concrete scenario instances. Therefore, it also suffers the limitation that a large number of simulations need to be carried out, or a large sample of scenario instances must be model checked.

The Soft Agent execution strategy is based on the Real Time Maude maximal time elapse (MTE) execution strategy for real time theories [17]. In [16] two conditions for soundness and completeness of model checking Real Time Maude specifications based on the MTE execution strategy are given. The first condition, time robustness, is a property of the rewrite theory. It requires that timesteps of any duration are allowed, and a timestep can be subdivided without changing the end result. The second condition requires that atomic propositions are stable with respect to time: at most one change during a time step. These conditions hold for a wide range of Real Time Maude specifications, timing of protocols, network performance, or discrete events used for defining system behavior of, e.g., manufacturing plants. SA specifications are concerned with physical properties of a system such as bounds on distance, change of position,
use of resources to express both safety and goal satisfaction properties. SA specifications are time robust, but the properties of interest are generally not stable with respect to time. Thus, we can not directly use the Real Time Maude results. Work is in progress to define an analog to stability for system properties that evolve over time.

A formal mathematical foundation for symbolic rewriting modulo SMT is presented in [19]. Our work is essentially a mapping of these ideas to be executable in Maude with an integrated SMT solver. The soft agents doTask rule is not technically topmost, but could easily be modified to be topmost without changing any behavior in our examples. Also, the theory $T$ has non-axiom equations that are not in $T_0$. These equations define functions in a straight forward way, so they do not cause a problem for our symbolic rewriting but may challenge narrowing. Our logical scenarios are ground terms from Maude’s perspective and correspond to terms whose only variables have builtin sorts (in $T_0$). On the other hand, search starts with terms that possibly have non builtin variables in [19]. Generating new symbols to update values plays a similar role to the fresh substitution used in the symbolic rewrite relation of [19]. Important future work is to better understand criteria for allowing equations over non-builtin sorts, to make symbolic rewriting modulo SMT more generally applicable.

A notion of guarded term is introduced in [2] as a method to reduce the search state space in symbolic rewriting modulo SMT. A guarded term is a pair consisting of a term and a constraint, or the disjunction of a set of guarded terms. The paper develops the formal theory of rewriting with guarded terms and presents experiments based on the CASH protocol showing state space reduction for various forms of guard. Although the paper motivates guards by a need to also reduce complexity of constraints sent to the SMT solver, no results on constraint size are reported. The results in the present paper seem to suggest that not only the size of state space matters for automation, but also the size of constraints that are sent to the SMT-Solver. It will be interesting to see if guards can be used to control the tradeoffs between search space size and constraint size explored in the present paper.

7 Conclusions

This paper proposes an extension of Soft Agents frameworks with Rewriting Modulo SMT to enable the automated generation of safety proofs of CPS. We demonstrate its expressiveness with a vehicle platoon scenario which is a common feature of autonomous vehicles. We carry out a collection of experiments demonstrating that delegating verification to rewriting has a positive impact in verification performance.

We are planning to use this framework in several directions that complement related work. We are currently automating the verification conditions for RTA [6]. We also believe that our framework is applicable to problems other than vehicle safety, for example it could be used to enable symbolic security verification by extending our previous work [4].
Acknowledgments. Talcott was partially supported by the U. S. Office of Naval Research under award numbers N00014-15-1-2202 and N00014-20-1-2644, and NRL grant N0017317-1-G002.

References

Executable Semantics and Type Checking for Session-Based Concurrency in Maude

Carlos Alberto Ramírez Restrepo and Jorge A. Pérez

1 Pontificia Universidad Javeriana Cali, Colombia
2 University of Groningen, The Netherlands

Abstract. Session types are a well-established approach to communication correctness in message-passing programs. We present an executable specification of the operational semantics of a session-typed π-calculus, implemented in Maude. We also develop an executable specification of its associated algorithmic type checking, and describe how both specifications can be integrated. We further explore how our executable specification enables us to detect well-typed but deadlocked processes by leveraging reachability and model checking tools in Maude. Our developments define a promising new approach to the (semi)automated analysis of communication correctness in message-passing concurrency.

1 Introduction

This paper presents an executable rewriting semantics for a π-calculus equipped with session types. Widely known as the paradigmatic calculus of interaction, the π-calculus offers a rigorous platform for reasoning about message-passing concurrency. Session types are arguably the most prominent representative of behavioral type systems, which can statically ensure that processes respect their ascribed interaction protocols and never exhibit errors and mismatches.

The integration of (variants of) the π-calculus with different formulations of session types has received much attention from foundational and applied perspectives. As a result, our understanding about (abstract) communicating processes and their typing disciplines steadily reaches maturity. Despite this progress, rigorous connections with more concrete representation models fall short. In particular, the study of session-typed π-calculi within frameworks like Maude [1] seems to remain unexplored. This gap is an opportunity to investigate the formal systems underlying session-typed π-calculi (reduction semantics and type systems) from a fresh yet rigorous perspective, taking advantage of the concrete representation given by executable semantics in Maude.

Looking at session-typed π-calculi from the perspective of Maude is insightful, for several reasons. First, Maude enables the systematic validation of such formal systems and their results, improving over pen-and-paper developments. Second, as there is not a canonical session-typed π-calculus, but actually many different formulations (with varying features and properties), an implementation in Maude could provide a concrete platform for uniformly representing them all.
Third, resorting to Maude as a host representation framework for session-typed \( \pi \)-calculi could help in addressing known limitations of static type checking for deadlock detection, leveraging tools already available in Maude.

This paper reports our work on pursuing these three directions. We adopt the session-typed \( \pi \)-calculus developed by Vasconcelos in [10] as the basis for our implementation in Maude. For this typed language, dubbed \( s\pi \), we first implement its (untyped) reduction semantics as a rewriting semantics, essentially extending prior work on representing the \( \pi \)-calculus in Maude. Then, we implement its associated algorithmic type system, also given in [10]. Well-typedness in [10] ensures \textit{fidelity} (i.e., well-typed processes respect at runtime their ascribed protocols) but does not rule out deadlocks and other kinds of insidious circular dependencies. To address this, we leverage reachability and model checking in Maude. Our Maude developments are publicly available online.\(^3\)

To our knowledge, we are the first to represent session-typed \( \pi \)-calculi using Maude. Prior works have used rewriting logic to investigate the operational semantics for variants of the \( \pi \)-calculus. In [13] and [12], the reduction semantics of a synchronous \( \pi \)-calculus is defined as a rewrite theory, which is implemented in ELAN. The work [9] considers an untyped, asynchronous \( \pi \)-calculus, whose labeled transition semantics is implemented as a rewrite theory, which is used to formalize an associated may-testing preorder. The work [4] concerns a typed process calculus but in a different context, in which types are used to enforce privacy properties. Indeed, such work gives a Maude implementation of the labeled transition semantics of a privacy-oriented variant of the \( \pi \)-calculus and a Maude implementation of its associated type system, which is implemented as a membership equational theory.

The rest of this paper is organized as follows. Next, Section 2 summarizes the syntax and semantics of \( s\pi \). Section 3 describes the definition of our rewriting semantics for \( s\pi \) in Maude, whereas Section 4 presents the rewriting implementation of the algorithmic type checking. Section 5 presents our developments on deadlock detection. Section 6 closes with some concluding remarks. An extended version, available online, contains additional material [5].

\section{The Typed Process Model}

The typed process calculus \( s\pi \), formalized by Vasconcelos [10], is a variant of the synchronous \( \pi \)-calculus (cf. [6]) with constructs for session-based concurrency. Here we summarize its syntax and semantics.

The calculus \( s\pi \) relies on a base set of variables, ranged over by \( x, y, \ldots \). Variables denote \textit{channels} (or \textit{names}). Processes interact to exchange values, which can be variables or booleans. Variables can be seen as consisting of (dual) \textit{endpoints} on which interaction takes place. Rather than non-deterministic choices among prefixed processes, there are two complementary operators: one for offering a finite set of alternatives (called \textit{branching}) and one for choosing one of

\(^3\) See https://gitlab.com/calrare1/session-types
\[
\begin{align*}
P \mid Q & \equiv Q \mid P \\
(P \mid (Q \mid R)) & \equiv ((P \mid Q) \mid R) \\
(\nu xy)(\nu wz)P & \equiv (\nu wz)(\nu xy)P \\
(\nu xy)P & \mid Q \equiv (\nu xy)(P \mid Q)
\end{align*}
\]

if true then \(P_1\) else \(P_2\) \(\equiv P_1\) if false then \(P_1\) else \(P_2\) \(\equiv P_2\)

Fig. 1. Structural congruence Rules for \(\pi\)

such alternatives (selection). More formally, the syntax of values, qualifiers, and processes is presented below:

\[
v ::= x \mid \text{true} \mid \text{false} \quad q ::= \text{un} \mid \text{lin} \\
P ::= 0 \mid \pi v.P \mid q x(y).P \mid P_1 \mid P_2 \mid (\nu xy)P \\
\text{if } v \text{ then } P_1 \text{ else } P_2 \mid x \triangleleft l.P \mid x \triangleright \{l_i : P_i\}_{i \in I}
\]

The inactive process is denoted as \(0\). The output process \(\pi v.P\) sends the value \(v\) along \(x\) and continues as \(P\). Process \(q x(y).P\) denotes an input action on \(x\), which prefixes \(P\). The qualifier \(q\) is used for inputs, which can be linear (to be executed exactly once) or shared. Process \(\text{un } x(y).P\) denotes a persistent input action, which corresponds to (input-guarded) replication in the \(\pi\)-calculus. The parallel composition \(P_1 \mid P_2\) denotes the concurrent execution of \(P_1\) and \(P_2\). Process \((\nu xy)P\) declares the scope of co-variables \(x\) and \(y\) to be \(P\). These co-variables are intended to be the output and input ends of a communication channel. Given a boolean \(v\), process if \(v\) then \(P_1\) else \(P_2\) continues as \(P_1\) if \(v\) is true; otherwise it continues as \(P_2\). Finally, selection process \(x \triangleleft l.P\) chooses an option \(l\) offered by a process prefixed at the co-variable and branching process \(x \triangleright \{l_i : P_i\}_{i \in I}\) offers multiple alternatives, which are labeled \(l_1, l_2, \ldots\); the selection process continues with \(P\) and the branching process with a process \(P_j\).

As usual, \(q x(y).P\) binds \(y\) in \(P\) and \((\nu xy)P\) binds \(x, y\) in \(P\). The set of free and bound names of a process \(P\), denoted \(\text{fn}(P)\) and \(\text{bn}(P)\), is as expected.

The operational semantics for \(\pi\) is given as a reduction semantics, which, as customary, relies on a structural congruence relation, the smallest congruence relation on processes that satisfy the axioms in Fig. 1. Structural congruence includes the usual axioms for inaction and parallel composition as well as adapted axioms for scope restriction, scope extrusion, and conditionals. Armed with structural congruence, the rules of the reduction semantics are presented in Fig. 2. Rules \([R-LinCom]\) and \([R-UnCom]\) induce different patterns for process communication, depending on the qualifier of their corresponding input action. Indeed, processes \(\pi v.P\) and \(q y(z).Q\) can synchronize if \(x\) and \(y\) are co-variables. This is only possible if both processes are underneath a scope restriction \((\nu xy)\). When this occurs, processes \(\pi v.P\) and \(q y(z).Q\) continue respectively as \(P\) and \(Q[v/z]\), i.e., the process obtained from \(Q\) by substituting the free occurrences of \(z\) with \(v\). When \(q = \text{un}\) then process \(q y(z).Q\) remains (Rule \([R-UnCom]\)); otherwise, process \(q y(z).Q\) disappears (Rule \([R-LinCom]\)). Rule \([R-Case]\) stands
for the case synchronization: processes \( x \triangleright l_j.P \) and \( y \triangleright \{ l_i : Q_i \}_{i \in I} \) can synchronize if they are underneath a scope restriction \((\nu x y)\). Process \( x \triangleright l_j.P \) reduces to process \( P \) and process \( y \triangleright \{ l_i : Q_i \}_{i \in I} \) reduces to process \( Q \). Rules for parallel composition, scope restriction and structurally congruent processes are the usual from \( \pi \)-calculus (Rules [R-Par], [R-Res], [R-Struct]).

As an example, consider the processes:

\[
P_1 = \text{un } y_1(t).\text{false.0} \quad P_2 = \text{lin } y_1(w).\text{true.0} \quad P_3 = \pi x_1.x_2.y_1(\bar{z}).\pi z.0
\]

Starting from \( P \), there are two possible sequences of reductions depending on the processes involved in the initial synchronization in the co-variables \( x_1, y_1 \). If the synchronization involves \( P_1 \) and \( P_3 \) then we have:

\[
P \rightarrow \ldots \rightarrow (\nu x_1.y_1)(\nu x_2.y_2)(P_1 \mid P_2 \mid \pi \text{false.0})
\]

On the other hand, if \( P_2 \) and \( P_3 \) synchronize then we have:

\[
P \rightarrow \ldots \rightarrow (\nu x_1.y_1)(\nu x_2.y_2)(P_1 \mid \pi \text{true.0})
\]

The standard form of a process, defined in [10], will be crucial for the executable specification of the reduction semantics. Intuitively, a process is in standard form whenever restrictions are expanded as much as possible. More precisely, we say \( P \) is in standard form if it matches the pattern expression \((\nu x_1.y_1)(\nu x_2.y_2)\ldots(\nu x_n.y_n)(P_1 \mid P_2 \mid \ldots \mid P_h)\), where each \( P_i \) is a process of the form \( \pi v.Q, qx(y).Q, x \triangleright l.Q \) or \( \{ l_i : Q_i \}_{i \in I} \). Every process is structurally congruent to a process in standard form.

3 Rewriting Semantics for \( s\pi \)

Syntax Our rewriting semantics for \( s\pi \) adapts the one in [9], which is defined for an untyped \( \pi \)-calculus without sessions. There is a direct correspondence
between the syntactic categories (values, variables, qualifiers, and terms) and
Maude sorts (Value, Chan, Qualifier, and Trm, respectively). We also have
some auxiliary sorts such as Guard, Choice, and Choiceset.

```plaintext
sorts Value Chan Qualifier Trm Guard Choice Choiceset .
subsort Choice < Choiceset .
subsort Chan < Value .

op {_,} : Qid Nat -> Chan [prec 1] .
ops lin un : -> Qualifier [ctor] .
op ___(-): Qualifier Chan Qid -> Guard [ctor prec 5] .

nil : -> Trm [ctor] .

op nev[_,_]: Qid Qid Trm -> Trm [ctor prec 10] .
op _|_: Trm Trm -> Trm [ctor assoc comm prec 12 id: nil] .

if_then_else_fi: Value Trm Trm -> Trm [ctor prec 8] .

op ___<><_->: Chan Qid Trm -> Trm [ctor prec 15] .

op _=>_: Chan Choiceset -> Trm [ctor prec 17] .

op ___: Guard Trm -> Trm [ctor prec 7] .

op ___: Qid Trm -> Choice .

empty : -> Choiceset [ctor] .

op __: Choiceset Choiceset -> Choiceset [ctor assoc comm id: empty] .
```

Following the syntax in Section 2, values can be variables or booleans. We
represent booleans as the constructors True and False whereas we distinguish
variables (sort Chan) as values through the subsort relation. The only construc-
tor for variables {_,} takes a Qid and a natural number. Each production rule
for processes is represented using a constructor, as expected. Notice that the
constructor for input guards ___(-) is preceded by a qualifier. Process 0 is de-
noted as nil and a single guarded term is represented by the constructor _|_.
The constructor for scope restriction new[_,_] uses two instances of Qid, since
it declares a pair of co-variables. The constructor for conditionals is paramet-
ric on an instance of Value. We add constructors for selection and branching
process terms; their definition is as expected. In particular, the constructor for
branching processes relies on instances of Choiceset, which consists of sets of
pairs of Qid and process terms. We use instances of Qid to represent labels.

**Substitutions** As we have seen, the semantics of $\pi$ relies on substitutions of
variables with values. To deal with substitutions in Maude, we follow Thati
et al.’s approach [9] and use Stehr’s CINNI calculus [8], an explicit substitution
calculus, which provides a mechanism to implement $\alpha$-conversion at the language
level. The idea behind CINNI is to syntactically associate each use of a variable $x$
to an index, which acts as a counter of the number of binders for $x$ that are found
before it is used. In CINNI, there are three types of substitution operations:
A simple substitution of a variable \( a \) for a variable \( x \) takes place if the index of \( x \) is 0; the index is decreased by 1 otherwise. A shift substitution over \( a \) increases by 1 the index and a substitution \( S \) can be lifted to skip one index. Any substitution over a variable \( a \) has no effect on other variables.

We now present the definition of explicit substitutions for \( s\pi \) using an approach similar to [8]. We first present the definition of the variable substitutions. We use the sort \( \text{Subst} \) and the substitution application is performed by the operator \( \llbracket \_ \rrbracket \), which takes a substitution and a variable. We define the three substitutions above as presented there, by means of some equations.

\[
\begin{align*}
\text{sort} & \quad \text{Subst} . \\
\text{op} & \quad \llbracket \_ := \_ \rrbracket : \text{Qid} \times \text{Value} \to \text{Subst} . \\
\text{op} & \quad \llbracket \text{shiftup} \rrbracket : \text{Qid} \to \text{Subst} . \\
\text{op} & \quad \llbracket \text{lift} \_ \rrbracket : \text{Qid} \times \text{Subst} \to \text{Subst} . \\
\text{op} & \quad \_ : \text{Subst} \to \text{Chan} .
\end{align*}
\]

\[
\begin{align*}
eq & \quad \llbracket \_ := \_ \rrbracket a\{0\} = v . \\
eq & \quad \llbracket \_ := \_ \rrbracket a\{s(n)\} = a\{n\} . \\
\text{ceq} & \quad \llbracket \_ := \_ \rrbracket b\{n\} = b\{n\} \text{ if } a \neq b . \\
eq & \quad \llbracket \text{shiftup} \_ \rrbracket a\{n\} = a\{s(n)\} . \\
\text{ceq} & \quad \llbracket \text{shiftup} \_ \rrbracket b\{n\} = b\{n\} \text{ if } a \neq b . \\
eq & \quad \llbracket \text{lift} \_ S \rrbracket a\{0\} = a\{0\} . \\
\text{eq} & \quad \llbracket \text{lift} \_ S \rrbracket a\{n\} = \llbracket \text{shiftup} \_ \rrbracket S a\{n\} . \\
\text{ceq} & \quad \llbracket \text{lift} \_ S \rrbracket b\{n\} = \llbracket \text{shiftup} \_ \rrbracket S b\{n\} \text{ if } a \neq b .
\end{align*}
\]

Equipped with these elements, we adapt to the \( s\pi \) syntax the equations associated to the explicit substitutions for the process terms as follows:

\[
\begin{align*}
\text{op} & \quad \_ : \text{Subst} \to \text{Trm} \text{[prec 3]} . \\
\text{op} & \quad \text{subst-aux} : \text{Subst} \times \text{Choiceset} \to \text{Choiceset} . \\
\text{eq} & \quad \_ \text{ nil} = \text{nil} . \\
\text{eq} & \quad S \text{ (new \{x \ y\} \ P)} = \text{new \{x \ y\} (\llbracket \text{lift x} \ S \rrbracket \text{lift y} \ S \rrbracket \ P)} . \\
\text{eq} & \quad S \text{ (q a(y) . P )} = q S a(y) . \text{[(lift y} \ S \rrbracket \ P) . \\
\text{eq} & \quad S \text{ (a < b > . P)} = (S a) < (S b) > . (S P) . \\
\text{ceq} & \quad S \text{ (a < v > . P)} = (S a) < v > . (S P) \text{ if } v == \text{True or } v == \text{False} . \\
\text{ceq} & \quad S \text{ (if } v \text{ then } P \text{ else } Q \text{ fi)} = \text{if } v \text{ then } (S P) \text{ else } (S Q) \text{ fi} \text{ if } v == \text{True or } v == \text{False} . \\
\text{eq} & \quad S \text{ (a >> \{CH\})} = (S a) \gg \{ \text{subst-aux}(S, \text{CH}) \} . \\
\text{eq} & \quad S \text{ (a << x \ . P)} = (S a) \ll x \ . (S P) . \\
\text{eq} & \quad S \text{ (P \ | \ Q)} = (S P) \text{ | } (S Q) . \\
\text{eq} & \quad \text{subst-aux}(S, \text{empty}) = \text{empty} . \\
\text{eq} & \quad \text{subst-aux}(S, \text{ (x : P) \ CH)} = (x : (S P)) \text{ subst-aux}(S, \text{CH}) .
\end{align*}
\]
In each equation, we deal with a specific production rule for process terms. In each process, the substitution $S$ is applied in each variable and each subprocess as expected. Particularly, a lift substitution is performed over $x$, $y$ and $S$ to skip the index 0 and perform the substitution in the remaining indices for the scope restriction operator. In this way, the substitution $S$ has the expected effect.

**Structural Congruence** To represent the rules in Fig. 1, we exploit the Maude equational attributes `assoc`, `comm`, and `id` to declare the associative, commutative, and identity axioms for parallel composition, with process `nil` acting as its identity. This suffices to cover the rules on the two first lines of Fig. 1. The remaining rules are explicitly declared as equations below:

```plaintext
<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eq new[x y] nil = nil .</code></td>
<td></td>
</tr>
<tr>
<td>`ceq P</td>
<td>new[x y] Q = new [x y] (Q</td>
</tr>
<tr>
<td><code>if P /= nil /\ Q /= nil /\ CS := freenames(P) /\ x(0) in CS and y(0) in CS .</code></td>
<td></td>
</tr>
<tr>
<td><code>eq if True then P else Q fi = P .</code></td>
<td></td>
</tr>
<tr>
<td><code>eq if False then P else Q fi = Q .</code></td>
<td></td>
</tr>
<tr>
<td>`ceq P</td>
<td>new[x y] Q = new [x y] (Q</td>
</tr>
<tr>
<td><code>if P /= nil /\ Q /= nil /\ CS := freenames(P) /\ x(0) in CS and not y(0) in CS .</code></td>
<td></td>
</tr>
<tr>
<td>`ceq P</td>
<td>new[x y] Q = new [x y] (Q</td>
</tr>
<tr>
<td><code>if P /= nil /\ Q /= nil /\ CS := freenames(P) /\ not x(0) in CS and y(0) in CS .</code></td>
<td></td>
</tr>
<tr>
<td>`ceq P</td>
<td>new[x y] Q = new [x y] (Q</td>
</tr>
<tr>
<td><code>if P /= nil /\ Q /= nil /\ CS := freenames(P) /\ not x(0) in CS /\ not y(0) in CS .</code></td>
<td></td>
</tr>
</tbody>
</table>
```

In particular, scope extrusion is represented through four equations corresponding to the four cases in the presence of $x$, $y$ in the free names of process $P$. Function `freenames` stands for the Maude implementation for function $fn$ over processes.

**Operational Semantics** Combined, the Maude rewriting rules, the equational attributes, and the explicit equations associated to variables of sort `Trm` can appropriately express the reduction semantics of $s\pi$ and manipulate terms in a compositional fashion. A process is reduced to a simpler equivalent form by virtue of the equational theory; a process is rewritten as long as it satisfies the structure required for a rule wherever the process is located. As a consequence, subprocesses are also rewritten and we do not need to explicitly represent the contextual rules ([R-Par] and [R-Rts]).

A process is converted into standard form using the explicit congruence rules. This way, the scope of every unguarded occurrence of the `new` operator is extended to the top level.

Process interaction in $s\pi$ can only occur through co-variables and therefore processes that are involved must be underneath a scope restriction over such co-variables. Nonetheless, since in the standard form the order of the unguarded occurrences of the `new` operator is irrelevant, it would be necessary to explicitly
look for the processes that are enabled to interact, which would affect the efficiency of the rewriting specification. To counter this, we include an auxiliary operator, dubbed new*, which declares a list of pairs of new co-variables, rather than just a single pair. This is equivalent to using nested new operators, i.e., the term new* [x1 y1 x2 y2 ... xn yn] P is equivalent to the term

\[
\text{new [x1 y1] new [x2 y2] ... new [xn yn] P.}
\]

We declare the constructor for the sort QidSet with the equational attribute comm to impose that the order among the pairs of new co-variables is not distinguished. In this way, whatever they are the process to interact, these will be underneath a scope restriction new* and the interaction will be enabled.

sorts QidPair QidSet . subsort QidPair < QidSet .
op __ : Qid Qid -> QidPair [ctor] .
op mt : -> QidSet [ctor] .
op __ : QidSet QidSet -> QidSet [ctor comm assoc id: mt] .
op new* [ ] _ : QidSet Trm -> Trm [ctor] .

Given a process \( P \), let us write \( \llbracket P \rrbracket \) to denote its representation in Maude. A reduction rule \( P \rightarrow Q \) can be associated to a rewriting rule \( l : \llbracket P \rrbracket \Rightarrow \llbracket Q \rrbracket \).

The reduction rules can be stated as follows:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>crl [FLAT]</td>
<td>( P \Rightarrow P' ) if ( P' := \text{flatten}(P) ) and ( P \neq P' ).</td>
</tr>
<tr>
<td>rl [LINCOM]</td>
<td>new* [(x y) nl] ( x{N} \mathbin{&lt;} v \mathbin{&gt;} . P \mid \text{lin } y(N)(z) ). Q \mid R \Rightarrow \text{new* [(x y) nl] } P \mid [z := v] Q \mid R .</td>
</tr>
<tr>
<td>rl [UNCOM]</td>
<td>new* [(x y) nl] ( x{N} \mathbin{&lt;} v \mathbin{&gt;} . P \mid \text{un } y(N)(z) ). Q \mid R \Rightarrow \text{new* [(x y) nl] } P \mid [z := v] Q \mid \text{un } y(N)(z) ). Q \mid R .</td>
</tr>
<tr>
<td>rl [CASE]</td>
<td>new* [(x y) nl] ( x(N) \mathbin{&lt;&lt;} w . P \mid (y(N) \mathbin{&gt;&gt;} { (w : Q) \text{ CH } }) \mid R \Rightarrow \text{new* [(x y) nl] } P \mid Q \mid R .</td>
</tr>
</tbody>
</table>

Rule FLAT normalizes the whole process. In this sense, additional to the implicit rewriting performed by the equations associated to the congruence rules, the nested new declarations are stated as a flat declaration new*. We use an auxiliary operation flatten, which is defined as follows:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>op flatten : Trm -&gt; Trm .</td>
<td></td>
</tr>
<tr>
<td>eq flatten(new [x y] P) = flatten(new* [x y] P) .</td>
<td></td>
</tr>
<tr>
<td>eq flatten(new* [nl] new [x y] P) = flatten(new* [nl x y] P) .</td>
<td></td>
</tr>
<tr>
<td>eq flatten(new* [nl] new* [nl'] P) = flatten(new* [nl nl'] P) .</td>
<td></td>
</tr>
<tr>
<td>eq flatten(P) = P [otherwise] .</td>
<td></td>
</tr>
</tbody>
</table>

Rules LINCOM, UNCOM and CASE correspond to the specification of the reduction rules related to synchronization in the calculus semantics (see Fig. 2). In these rules, nl stands for the additional co-variables being declared. As expected, Rules LINCOM, and UNCOM perform a substitution of the variable z for the value v.

We include also some equations which capture natural equivalences for processes involving the auxiliary operator new*. 189
\begin{align*}
\text{eq new* [nl] nil = nil .} \\
\text{eq new* [x y nl] y{N} < v > . P | q x{N}(z) . Q | R =} \\
\text{new* [y x nl] y{N} < v > . P | q x{N}(z) . Q | R .} \\
\text{eq new* [x y nl] (y{N} << w . P) | (x{N} >> \{ CH \}) | R =} \\
\text{new* [y x nl] (y{N} << w . P) | (x{N} >> \{ CH \}) | R .}
\end{align*}

Given a pair \( x, y \) of co-variables, we assume that the first action of \( x \) is an output or a selection and the first action \( y \) is an input or a branching. The last two equations swap \( x \) and \( y \) when this is not the case, to enable the execution of the rewriting rules.

Our rewriting specification enables us to directly execute a possible sequence of reductions over a process using the Maude command `rew`. In this way, we can obtain a stable (final) reachable process, which cannot reduce further. Moreover, we can use the reachability command `search` to: (i) perform all possible sequence of reductions of a process and obtain every possible stable process and (ii) check whether a process that fits some pattern is reachable or if a specific process is reachable. In Section 4, we leverage commands `search` and `modelCheck` to detect deadlocked \( s\pi \) processes.

**Specification Correctness** The transition system associated to our rewrite theory in Maude can be shown to coincide with the reduction semantics in Section 2. This operational correspondence result is detailed in [5].

## 4 Algorithmic Type Checking for \( s\pi \)

### 4.1 Type Syntax

We present a Maude implementation of the algorithmic type checking given in [10]. The type system considers *typing contexts*, denoted \( \Gamma \), which associate each variable to a specific type, denoted \( T \). Typing contexts and types are defined inductively as follows:

\[
\begin{align*}
\Gamma & ::= \emptyset \mid \Gamma, x : T \\
q & ::= \text{lin} \mid \text{un} \\
p & ::= \text{?}T.T \mid \text{!}T.T \mid \&\{l_i : T_i\}_{i \in I} \mid \oplus\{l_i : T_i\}_{i \in I} \\
T & ::= \text{bool} \mid \text{end} \mid q p \mid a \mid \mu a.T
\end{align*}
\]

where \( q \) stands for qualifiers and \( p \) stands for pretypes. Moreover, \( x \) denotes a variable, each \( l_i \) denotes a label and \( a \) denotes a general variable. For simplicity, we assume a single basic type for values (\text{bool}). Each variable is associated to a (session) type, which represents its intended protocol. In the above grammar, these types correspond to qualified pretypes. The pretype \( ?T_1.T_2 \) (resp. \( !T_1.T_2 \)) is assigned to a variable that first receives (resp. sends) a value of type \( T_1 \) and then proceeds to type \( T_2 \). The pretype \( \&\{l_i : T_i\}_{i \in I} \) (resp. \( \oplus\{l_i : T_i\}_{i \in I} \)) is assigned to a variable that can offer (resp. select) \( l_i \) options and continues with type \( T_i \) depending on the label selected. The type \text{end} (empty sequence) denotes the type of a variable where no interaction can occur. Recursive types can express
infinite sequences of actions; in the type \( \mu a.T \), \( a \) corresponds to a type variable that must occur guarded in \( T \).

We encode session types in Maude by associating the non-terminals context, qualifiers, pretypes, and types to sorts \texttt{Context}, \texttt{Qualifier}, \texttt{Pretype}, and \texttt{Type}.

<table>
<thead>
<tr>
<th>sorts</th>
<th>Type Context ChoiceT ChoiceTset</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>op ?___</code></td>
<td>: Type Type \rightarrow Pretype</td>
</tr>
<tr>
<td><code>op !____</code></td>
<td>: Type Type \rightarrow Pretype</td>
</tr>
<tr>
<td><code>op +{___}</code></td>
<td>: ChoiceTset \rightarrow Pretype</td>
</tr>
<tr>
<td><code>op &amp;{___}</code></td>
<td>: ChoiceTset \rightarrow Pretype</td>
</tr>
<tr>
<td><code>op u[a]___</code></td>
<td>: Qid Type \rightarrow Type</td>
</tr>
<tr>
<td><code>op var</code></td>
<td>: Qid \rightarrow Type</td>
</tr>
<tr>
<td><code>ops bool end</code></td>
<td>: \rightarrow Type</td>
</tr>
<tr>
<td><code>op __</code></td>
<td>: Qualifier Pretype \rightarrow Type</td>
</tr>
<tr>
<td><code>ops nil invalid-context</code></td>
<td>: \rightarrow Context</td>
</tr>
<tr>
<td><code>ops _:_</code></td>
<td>: Value Type \rightarrow Context</td>
</tr>
<tr>
<td><code>ops _:_</code></td>
<td>: Context Context \rightarrow Context</td>
</tr>
<tr>
<td><code>ops _:_</code></td>
<td>: Qid Type \rightarrow ChoiceT</td>
</tr>
<tr>
<td><code>op empty</code></td>
<td>: \rightarrow ChoiceTset</td>
</tr>
<tr>
<td><code>ops _:_</code></td>
<td>: ChoiceTset ChoiceTset \rightarrow ChoiceTset</td>
</tr>
</tbody>
</table>

Each production rule is given as a specific constructor. In particular, constructors \(+{___}\) and \&{___}\) represent the pretypes \( \oplus \{l_i : T_i\}_{i \in I}\) and \( \&\{l_i : T_i\}_{i \in I}\), respectively. The pairs of labels \( l_i \) and subtypes \( T_i \) are defined as instances of the sort \texttt{ChoiceTset}. The recursive type \( \mu a.T \) is given as the constructor \( u[a]___\) and the type variables are given as the constructor \( \text{var} \). Typing contexts are defined as expected. An empty context is denoted as \( \text{nil} \) whereas a single context is associated to the constructor \( _:_ \). General contexts are provided by the constructor \( _:_ \), which is annotated with the equational attributes \texttt{assoc}, \texttt{comm} and \texttt{id} since the order is irrelevant in typing contexts and the construction is associative. Finally, we added a constant \texttt{invalid-context} to be used in the type checking to denote a typing error.

### 4.2 Algorithmic Type Checking

We follow the algorithmic type checking proposed in [10]. This type system enables to type check the \( \pi \) processes from Section 2, with a minor caveat: algorithmic type checking uses processes in which the restriction operator has a corresponding type annotation, i.e., it uses \((\nu xy : T)P\) instead of \((\nu xy)P\). Consequently, we add a constructor for the sort \texttt{Trm} in the Maude specification:

| op `new[___]` | : Qid Qid Type Trm \rightarrow Trm [ctor prec 28] |

Following [10], we implement the type checking algorithm by relying on some auxiliary functions for type duality (i.e., compatibility), type equality, and context update and difference, among others. They are implemented by means of functions and equations in Maude. See [5] for the details of the Maude implementation for type duality (function \texttt{dual}), context update (function \texttt{+}), and the context difference (function \texttt{\textbackslash}).

Algorithmic type checking is expressed by using sequents of the form \( \Gamma_1 \vdash v : T ; \Gamma_2 \) for values and \( \Gamma_1 \vdash P : \Gamma_2 ; L \) for processes. These two sequents have an input-output reading: sequent \( \Gamma_1 \vdash v : T ; \Gamma_2 \) denotes an algorithm that takes \( \Gamma_1 \)
and \(v\) as input and returns \(T\) and \(\Gamma_2\) as output; similarly, sequent \(\Gamma_1 \vdash P : \Gamma_2\); \(L\) denotes an algorithm that takes \(\Gamma_1\) and \(P\) as input and produces \(\Gamma_2\) and \(L\) as output. While \(\Gamma_2\) is a residual context, the set \(L\) collects linear variables occurring in subject position. Intuitively, \(L\) tracks the linear variables that are used in \(P\) to prevent that they are used again in another process. Both algorithms are given by means of typing rules, which we specify in Maude as an equational theory.

Fig. 3 shows the typing rules for values, \(\Gamma \vdash v : T; \Gamma\).

| \(\Gamma \vdash true : bool; \Gamma\) | \(\Gamma_1, x : \text{lin } p, \Gamma_2 \vdash x : \text{lin } p; (\Gamma_1, \Gamma_2)\) |
| \(\Gamma \vdash false : bool; \Gamma\) | \(\text{un}(T)\)

| \(\Gamma_1, x : T, \Gamma_2 \vdash x : T; (\Gamma_1, x : T, \Gamma_2)\) |

Fig. 4 shows some of the typing rules for \(\pi\) processes; they largely correspond to the rules in [10].

Function \(\text{type-value}\) produces an instance of the sort \(\text{TupleTypeContext}\). This sort groups a context and a type or a set of variables and it has only one constructor \([_\_\_]\). The equations related to the typing of boolean values arise as expected, according to the corresponding typing rule. In those cases, a tuple that contains the unmodified context and the type \(\text{bool}\) is produced. For unrestricted variables, given that some types are infinite then, before the update, the unrestricted types are unfolded (cf. the \text{unfold} operation). Unfolding is the mechanism defined in [10] to deal with infinite types: If a type \(T\) is a recursive type \(\mu\alpha.U\) then the substitution \(U[\mu\alpha.U/\alpha]\) is performed. Otherwise, the type \(T\) remains unaltered. For linear variables, we also unfold the type when necessary and the linear type is returned and removed from the context.

Fig. 4 shows some of the typing rules for \(\pi\) processes; they largely correspond to the rules in [10].
\[
\begin{array}{c}
\Gamma \vdash 0 : \Gamma; \emptyset \\
\Gamma_1 \vdash P : \Gamma_2; L_1 \\
\Gamma_2 \vdash L_1 \vdash Q : \Gamma_3; L_2 \\
\Gamma_1 \vdash P \mid Q : \Gamma_3; L_2 \\
\hline
\text{[A-Inact]} & \text{[A-Par]} \\
\hline
\Gamma_1, x : T, y : T \vdash T : T; L_1 \vdash (x, y) \vdash L_1 \vdash \{x, y\} \vdash \{T, L_1\} \vdash \{x, y\} \vdash \{T, L_1\} \\
\Gamma_1 \vdash y : q \text{ bool}; \Gamma_2 \vdash P : \Gamma_3; L_2 \vdash q : \{x\} \vdash Q : \{x\} \\
\Gamma_1 \vdash y : q \text{ bool} \vdash P \mid Q : \{x\} \vdash Q : \{x\} \\
\hline
\text{[A-Res]} & \text{[A-Par]} \\
\hline
\Gamma_1 \vdash x : q ! T, U; \Gamma_2 \vdash P : \Gamma_3; L_1 \vdash x : U \vdash P : \Gamma_3; L_2 \\
\Gamma_1 \vdash x : q ! T, U; \Gamma_2 \vdash P : \Gamma_3; L_1 \vdash x : U \vdash P : \Gamma_3; L_2 \\
\hline
\text{[A-Out]} & \text{[A-Out]} \\
\hline
\Gamma_1 \vdash x : q \text{ bool}; \Gamma_2 \vdash P : \Gamma_3; L_1 \vdash x : q \text{ bool} \vdash P : \Gamma_3; L_2 \\
\Gamma_1 \vdash x : q \text{ bool}; \Gamma_2 \vdash P : \Gamma_3; L_1 \vdash x : q \text{ bool} \vdash P : \Gamma_3; L_2 \\
\hline
\text{[A-In]} & \text{[A-In]} \\
\hline
\Gamma_1 \vdash x : q \text{ bool}; \Gamma_2 \vdash P : \Gamma_3; L_1 \vdash x : q \text{ bool} \vdash P : \Gamma_3; L_2 \\
\Gamma_1 \vdash x : q \text{ bool}; \Gamma_2 \vdash P : \Gamma_3; L_1 \vdash x : q \text{ bool} \vdash P : \Gamma_3; L_2 \\
\hline
\text{[A-Branch]} & \text{[A-Branch]} \\
\hline
\Gamma_1 \vdash x : q \{l_{1 : L_1} \} \vdash \Gamma_2 \vdash P : \Gamma_3; L_1 \vdash x : q \{l_{1 : L_1} \} \vdash \Gamma_2 \vdash P : \Gamma_3; L_1 \\
\Gamma_1 \vdash x : q \{l_{1 : L_1} \} \vdash \Gamma_2 \vdash P : \Gamma_3; L_1 \vdash x : q \{l_{1 : L_1} \} \vdash \Gamma_2 \vdash P : \Gamma_3; L_1 \\
\hline
\text{[A-Branch]} & \text{[A-Branch]} \\
\hline
\Gamma_1 \vdash x : q \{l_{1 : L_1} \} \vdash \Gamma_2 \vdash P : \Gamma_3; L_1 \vdash x : q \{l_{1 : L_1} \} \vdash \Gamma_2 \vdash P : \Gamma_3; L_1 \\
\end{array}
\]

Fig. 4. Typing Rules for Processes, \( \Gamma \vdash P : \Gamma; L \)

Rule [A-Inact] proceeds as expected. Process 0 is well-typed and the typing context \( \Gamma \) remains unaltered and the set of linear variables is empty. Rule [A-Par] handles parallel composition; to check a process \( P \mid Q \) over a context \( \Gamma_1 \), the type of \( P \) is checked and the resulting context \( \Gamma_2 \) is used to type-check process \( Q \), making sure that the linear variables used for \( P \) are first removed by using the context difference function (\( \Gamma_2 \vdash L_1 \)). This ensures that free linear variables are used only once. The output of the algorithm for \( Q \) (context \( \Gamma_2 \) and set \( L_2 \)) then corresponds to the output of the entire process \( P \mid Q \). Rule [A-Res] type-checks a process \( \nu x : T \vdash P \) in a context \( \Gamma_1 \); it first checks the type of sub-process \( P \) in the context \( \Gamma_1 \) extended with the association of variables \( x, y \) to the type \( T \) and its dual type, denoted \( \overline{T} \). It is expected that if type \( T \) (\( \overline{T} \)) is linear then it should not be in the resulting context \( \Gamma_2 \); otherwise, if type \( T \) (\( \overline{T} \)) is unrestricted then it will appear in \( \Gamma_2 \). We require that variables \( x, y \) are deleted from the residual context (\( \Gamma_2 \vdash \{x, y\} \)) and from the set \( L \) of linear variables.

Rule [A-Branch] verifies that type of value \( v \) is \( \text{bool} \) in the context \( \Gamma_1 \), and requires that the typechecking of \( P \) and \( Q \) in the context \( \Gamma_2 \) generate the same residual context \( \Gamma_3 \) and the same set \( L \), since both processes should use the same linear variables. Rule [A-Out] handles output processes; it uses the incoming context \( \Gamma_1 \) to check the type of \( x \), which should be of the form \( q ! T.U \). Then, it checks that the type of \( v \) in the residual context \( \Gamma_2 \) is \( T \). The type of the continuation \( P \) is checked in a new context \( \Gamma_3 \) extended with the association of \( x \) and the continuation type \( U \). The rule enforces that types \( q ! T.U \) and \( U \) must be equivalent.
when $x$ is unrestricted (i.e., $q = \text{un}$). The rule returns a context $\Gamma_4$ and a set of variables $L$ joined with $x$, if linear. Rule $[A-\text{IN}]$ presents some minor modifications with respect to the one in [10]. We require that in the case of replication there are no (free) subjects on linear variables in process $P$ except possibly the input variable $y$. Other than this, this rule is similar to Rule $[A-\text{OUT}]$.

Rule $[A-\text{Sel}]$ looks the type of $x$ in the incoming context $\Gamma_1$. This type must be of the form $q \oplus \{(l_i : T_i)_{i \in I}\}$. Subsequently, the continuation $P$ is type-checked under the resulting context $\Gamma_2$ updated with a new assumption for $x$, which is associated to a type $T_i$. This rule produces as result the context $\Gamma_3$ and the set of linear variables $L$ is augmented with $x$ if linear. Context $\Gamma_3$ and set $L$ also corresponds to the output of the type checking of process $P$. Finally, we have Rule $[A-\text{Branch}]$, which has some minor modifications with respect to the rule in [10]. More precisely, this rule has been changed to require that the sets of (free) subjects on linear variables $L_i$ only differ in the input variable $y$. The additional details of this rule is quite similar to Rule $[A-\text{Sel}]$.

As an example of type checking, if $T = \text{lin} ! \text{bool}. \text{lin} ? \text{bool}. \text{end}$ then we can establish the following sequent:

$$a : \text{bool} \vdash (\nu x_1 y_1 : T)(\nu y_1(v). \overline{\pi y_1.0} | \overline{\pi a.\text{lin} x_1(z).0}) : (a : \text{bool})\{x_1, y_1\}$$

The algorithm for type-checking processes is implemented as a function $\text{type-term}$ that receives an instance of the sort $\text{Context}$ and an instance of the sort $\text{Trm}$. Moreover, it produces an instance of the sort $\text{TupleTypeContext}$ that groups the resulting typing context and the set $L$ of linear variables that were collected during type-checking. Each rule is implemented by an equation:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[A-\text{INACT}]$</td>
<td>$\text{ceq type-term}(C, \text{nil}) = [C \text{ mt}]$</td>
</tr>
<tr>
<td>$[A-\text{PAR}]$</td>
<td>$\text{ceq type-term}(C, P \mid Q) = [C_2 L_2] \text{ if } [C_1 L_1] := \text{type-term}(C, P) \land [C_2 L_2] := \text{type-term}(C_1 / L_1, Q)$</td>
</tr>
<tr>
<td>$[A-\text{RES}]$</td>
<td>$\text{ceq type-term}(C, \text{new } [x y : T] P) =$ $[C_1 / (x{0} y{0})] \text{ remove(remove(L1, x{0}), y{0})}] \text{ if } [C_1 L_1] := \text{type-term}(C, (x{0} : T), (y{0} : \text{dual}(T))), P) \lor \text{ceq type-term}(C, i f \ v \ \text{then } P \ \text{else } Q \ \text{fi}) =$ $[C_2 L_1] \text{ if } [C_1 \text{ bool}] := \text{type-value}(C, v) \land [C_2 L_1] := \text{type-term}(C_1, P) \land [C_2 L_1] := \text{type-term}(C_1, Q)$</td>
</tr>
<tr>
<td>$[A-\text{OUT}]$</td>
<td>$\text{ceq type-term}(C, a &lt; v &gt; . P) =$ $[C_3 (\text{if } q = \text{lin} \text{then } (L_1 a) \text{ else } L_1 \text{ fi})] \text{ if } [C_1 (q ! T . U)] := \text{type-value}(C, a) \lor [C_2 T'] := \text{type-value}(C_1, v) \land \text{equal}(T, T') \lor [C_3 L_1] := \text{type-term}([C_2 + a : U], P) \lor [C_4 a(y) . P) =$ $[(C_2 / y{0}) \text{ mt}] \text{ if } [C_1 (\text{un } T . U)] := \text{type-value}(C, a) \lor [C_2 L] := \text{type-term}([C_1, (y{0} : T)] + a : U, P) \land \text{remove(L, y{0})} = \text{mt}$</td>
</tr>
</tbody>
</table>
When type checking is successful, function type-term produces an outgoing type context and a set of variables. Those elements are grouped using the constructor [_,_], which is associated to the sort TupleTypeContext. We use a Maude comment to annotate each equation with the correspondent typing rule. The correspondence is quite intuitive: we highlight some important details. An empty set of variables is represented with the constant mt. We remark that the operator / stands for the context difference operation that removes some variables of a type context, whereas operator 'remove' drops a variable of a variable set. In the equation for Rule [A-Out], we do not use the same variable T in the type associated to variable a and the type associated to value v as it would be expected, since the types are possibly infinite and there are many possible representations for the same infinite type. Instead, we use another variable T' and we check that T and T' are equivalent, using function equal.

We divide Rule [A-In] in two different equations for linear and unrestricted inputs. In the linear case, it is possible that the type of the subject a is linear or unrestricted; when the variable is linear it must be included in the returned set of linear variables. In the unrestricted case, the type of subject a is required to be unrestricted inasmuch as the attempt to use a linear variable in an unrestricted fashion must be rejected. Moreover, we require that the only free linear variable used in process P is y{0} (condition remove(L, y{0}) == mt).

4.3 Type Soundness

Vasconcelos [10] established that the type system for sπ is sound: a closed, well-typed process is guaranteed to have a well-defined behavior according to the ascribed protocols and the reduction semantics of the calculus. Also, the algorithmic type checking, as implemented in this section, is proven correct. With these elements in mind, we can integrate both the rewriting specification of the operational semantics and the implementation of the algorithmic type checking. This way, we only execute well-typed processes. For this purpose, we use two auxiliary functions well-typed and erase. The former checks whether a process does not have typing errors:
Function **well-typed** applies the algorithm for type checking **type-term** over a process \( P \) and returns **true** when type-checking is successful, i.e. when the result is not **ill-typed**. Function **erase** proceeds inductively on the structure of a process; when it reaches an annotated subprocess ‘\( \text{new} [x \ y : T] \ P \)’, it removes the annotation to produce ‘\( \text{new} [x \ y] \ P \)’—see [5] for details.

Correspondingly, we extend our specification of the reduction semantics to enable the execution of annotated processes, i.e., processes that use the operator \((\nu xy : T)P\) instead of the operator \((\nu xy)P\):

\[
\text{rl [TYPED]} : \text{new} [x \ y : T] \ P \Rightarrow \begin{cases} 
\text{if well-typed(new} [x \ y : T] \ P) \\
\text{then erase(new} [x \ y] \ P \) \text{ else ill-typed-process fi .}
\end{cases}
\]

We check whether process \( \text{new} [x \ y : T] \ P \) is well-typed; if so, we rewrite it as an equivalent process in which each occurrence of \( \text{new} [x \ y : T] \) is replaced by \( \text{new} [x \ y] \) through the function **erase**. Otherwise, process \( \text{new} [x \ y : T] \ P \) is rewritten as **ill-typed-process**, a constant that denotes that the process has a typing error and cannot be executed.

## 5 Lock and Deadlock Detection in Maude

Although the type system for \( s\pi \) given in [10] enables us to statically detect processes whose variables are used according to their ascribed protocols (expressed as session types), there are processes that are well-typed but that exhibit unwanted behaviors, in particular deadlocks. For example, consider the process:

\[
P = x_3!\text{true}.x_1!\text{true}.y_2!!\text{false}.0 \mid \text{lin } y_3(z).\text{lin } x_2(w).\text{lin } y_1(t).0
\]

Process \( P \) is well-typed in a context \( x_1 : \text{lin } !\text{bool.end}, y_1 : \text{lin } ?\text{bool.end}, x_2 : \text{lin } ?\text{bool.end}, y_2 : \text{lin } !\text{bool.end}, x_3 : \text{lin } !\text{bool.end}, y_3 : \text{lin } ?\text{bool.end} \). Then, process \( (\nu x_1 y_1 x_2 y_2 x_3 y_3)P \) can reduce but becomes deadlocked after such a synchronization, due to a circular dependency on variables \( x_1, y_1, x_2, y_2 \).

### 5.1 Definitions

Here we characterize deadlocks in \( s\pi \) and we show how we can use the rewrite specification of the operational semantics and the Maude tools for detecting processes with deadlocks. We follow the formulation of deadlock and lock freedom given by Padovani [3], which uses the notion of pending communication. We start by defining the reduction contexts \( \mathcal{C} \):

\[
\mathcal{C} ::= [\ ] \mid (\mathcal{C} \mid P) \mid (\nu xy)\mathcal{C}
\]
The notion of pending communication in a process $P$ with respect to variables $x, y$ is defined with the following auxiliary predicates:

\[
\begin{align*}
in(x, P) & \overset{\text{def}}{=} P \equiv C[\text{lin } x(y).Q] \land x \notin \text{bn}(C) \\
in^*(x, P) & \overset{\text{def}}{=} P \equiv C[\text{un } x(y).Q] \land x \notin \text{bn}(C) \\
out(x, P) & \overset{\text{def}}{=} P \equiv C[\text{v}.Q] \land x \notin \text{bn}(C) \\
sync(x, y, P) & \overset{\text{def}}{=} \left( (\text{in}(x, P) \lor \text{in}^*(x, P)) \land \text{out}(y, P) \right) \\
wait(x, y, P) & \overset{\text{def}}{=} \left( (\text{in}(x, P) \lor \text{out}(y, P)) \land \neg sync(x, y, P) \right)
\end{align*}
\]

There, we assume the extension of function $\text{bn}(.)$ to reduction contexts. Intuitively, the first three predicates express the existence of a pending communication on a variable $x$. More in details:

- Predicate $\text{in}(x, P)$ holds if $x$ is free in $P$ and there is a subprocess of $P$ that is able to make a linear input on $x$. Predicate $\text{in}^*(x, P)$ is its analog for unrestricted inputs.
- Predicate $\text{out}(x, P)$ holds if $x$ is free in $P$ and a subprocess of $P$ is waiting to send a value $v$.
- Predicate $\text{sync}(x, y, P)$ denotes a pending input on $x$ for which a synchronization on $y$ is immediately possible.
- Predicate $\text{wait}(x, y, P)$ denotes a pending input/output for which a synchronization on $x, y$ is not immediately possible.

Let us write $\rightarrow^*$ to denote the reflexive, transitive closure of $\rightarrow$. Also, write $P \nrightarrow$ if there is no $Q$ such that $P \rightarrow Q$. With these elements, we now proceed to characterize the deadlock and lock freedom properties. We say process $P$ is

- **deadlock free** if for every $Q$ such that $P \rightarrow^* (\nu x_1 y_1)(\nu x_2 y_2)\ldots(\nu x_n y_n)Q \nrightarrow$ it holds that $\neg \text{wait}(x_i, y_i, Q)$ for every $x_i$.
- **lock free** if for every $Q$ such that $P \rightarrow^* (\nu x_1 y_1)(\nu x_2 y_2)\ldots(\nu x_n y_n)Q$ and $\text{wait}(x_i, y_i, Q)$ there exists $R$ such that $Q \rightarrow^* R$ and $\text{sync}(x_i, y_i, R)$ hold.

This way, a process is deadlock free if there are not stable states with pending inputs or outputs; a process is lock free if it is able to eventually perform a synchronization in any pending input or output.

We can use Maude to verify deadlock freedom and lock freedom for typed processes. Indeed, we can use the reachability tool `search` and the LTL model checker `modelCheck`. We first represent the previous predicates over process terms as functions in Maude over instances of the sorts $\text{Trm}$ and $\text{Chan}$:

\[
\begin{align*}
\text{ops in out in* : Chan Trm \rightarrow Bool .} \\
\text{ops sync wait : Chan Chan Trm \rightarrow Bool .} \\
\text{op wait-aux : QidSet Trm \rightarrow Bool .} \\
\text{eq in(a, \text{lin } a(x) . Q | R) = true .} \\
\text{eq in(a, P) = false [owise] .} \\
\text{eq in*(a, un a(x) . Q | R) = true .} \\
\text{eq in*(a, P) = false [owise] .}
\end{align*}
\]
Above, we use function wait-aux to determine if a group of pairs of co-
variables contains a pair for which there is a pending communication.

The deadlock freedom property imposes that there should be no stable states
in which there are pending communications. Consequently, we can use the Maude
command search as follows to determine whether a process is deadlock free:

```
search init =>!
  new* [nl:QidSet] P:Trm such that wait-aux(nl:QidSet, P:Trm) .
```

where init denotes for the process to be checked. We recall that the search
command with the arrow =>! looks for final (stable) states. In this way, init is
deadlock free if the search returns no solution.

For the lock freedom property, we can not use the reachability tool since
this property requires the checking some intermediate states. Consequently, we
represent the lock freedom property as an LTL formula and use the built-in
LTL model checker in Maude. Below, we define the Maude predicates psync and
pwait that we will use in the LTL model checker:

```
ops pwait psync : Chan Chan -> Prop [ctor] .
eq new* [(x y) nl] P |= pwait(x{0}, y{0}) =
  wait(x{0}, y{0}, P) or wait(y{0}, x{0}, P) .
eq new* [(x y) nl] P |= psync(x{0}, y{0}) =
  sync(x{0}, y{0}, P) or sync(y{0}, x{0}, P) .
```

In the predicates psync and pwait, we use normalized processes, i.e., pro-
cesses where the nested scope restrictions are flattened in an equivalent process
that uses the operator new*. This assumption simplifies the definitions. Both
psync and pwait predicates use the functions in, in*, out, sync and wait as
expected according to the definition.

The Kripke structure that is generated for Maude will use such normalized
process term as states. The Maude predicates pwait and psync hold with respect
to a pair of dual variables if there is a pending communication and there is a
synchronization in the process associated to a state. The lock freedom property
imposes for each variable that if in any state there is a pending communication
then eventually there will be a synchronization. Formalizing the lock freedom
property requires to check each possible subject. For that reason, the LTL for-
mla formula associated to this property depends on the variables being used in the
process. We define a function build-lock-formula that takes the used vari-
ables and builds the corresponding LTL formula as follows:
ops P1 P2 P3 P4 P5 : -> Trm .
eq P1 = new* [('y1' 'x1')('y2' 'x2')('y3' 'x3')]
('x3'{0} < True > . 'x1'{0} < True > . 'y1'{0} < False > . nil | 
lin 'y3'('z') . lin 'y2'('x') . lin 'x2'('w') . nil) .

ops P2 = new* [('x1' 'y1')('x2' 'y2')('a' 'b')]
('x1'{0} < 'b'{0} > . nil | 'a'{0} < True > . nil |
un 'y1'('z') . 'x2'{0} < 'z'{0} > . nil |
un 'y2'('w') . 'x1'{0} < 'w'{0} > . nil ).

Fig. 5. Processes in Maude

This way, the resulting LTL formula corresponds to the conjunction of subformulas associated to each dual variable. The model checker can be used as follows:

red modelCheck(init, build-lock-formula(vars)) .

where init stands for the process term and vars stands for a set of pairs of co-variables. If the init is lock-free then the invocation of modelCheck will produce true. Otherwise, the invocation will show a counterexample with a sequence of rules that produces a state where the formula is not fulfilled.

5.2 Examples

We give a couple of examples of well-typed processes in sπ, with different lock- and deadlock-freedom properties. (See [5] for additional examples.)

\[ P_1 = (\nu x_1 y_1)(\nu x_2 y_2)(\nu x_3 y_3)(\nu y_1 x_1 y_1 y_2 x_2 y_2 y_3 x_3) \]
\[ P_2 = (\nu x_1 y_1)(\nu x_2 y_2)(\nu x_1 y_1 y_1 y_2 x_2 y_2 y_3 x_3, y_3) \]

Process \( P_1 \) is a simple process that reduces to a deadlock immediately after a synchronization on the co-variables \( x_3, y_3 \). Process \( P_2 \) represents an infinite process where the variable \( b \) is repeatedly shared through communications on \( x_1, y_1, x_2, y_2 \). The process is a not lock-free: \( b \) is never used to synchronize with its co-variable \( a \). Fig. 5 gives the Maude terms associated to these processes.

We analyze \( P_1 \) using Maude by executing:

search P1 =! new* [nl:QidSet] P:Trm 
such that wait-aux(nl:QidSet, P:Trm) .
red modelCheck(P1, 
build-lock-formula(('y1' 'x1')('y2' 'x2')('y3' 'x3'))) .

We obtain the following results, which confirm that \( P_1 \) is not deadlock free and not lock free:
Consider now a similar execution for process P2:

```
search P2 =>! new*[nl:QidSet] P:Trm
such that wait-aux(nl:QidSet, P:Trm).
red modelCheck(P2, build-lock-formula(('x1' 'y1')('x2' 'y2')('a' 'b'))).
```

We obtain the following results, which confirm that P2 is an infinite process that is deadlock free but not lock free:

```
search in TEST : P2 =>! new*[nl:QidSet] P:Trm
such that wait-aux(nl:QidSet, P:Trm) = true.
No solution.
result ModelCheckResult: counterexample(nil,
{new*[('a' 'b') ('x1' 'y1') 'x2' 'y2']
 'a'{0} < True > . nil | 'x1'{0} < 'b'{0} > . nil |
 un 'y1'{'0}('z') . 'x2'{0} < 'z'{0} > . nil |
 un 'y2'{'0}('w') . 'x1'{0} < 'w'{0} > . nil, 'UNCOM}
{new*[('a' 'b') ('x1' 'y1') 'x2' 'y2']
 'a'{0} < True > . nil | 'x2'{0} < 'b'{0} > . nil |
 un 'y1'{'0}('z') . 'x2'{0} < 'z'{0} > . nil |
 un 'y2'{'0}('w') . 'x1'{0} < 'w'{0} > . nil, 'UNCOM})
```

6 Closing Remarks

In this paper, we have reported on an executable specification in Maude of the operational semantics and the associated algorithmic type-checking of $\pi_s$, a session-typed $\pi$-calculus proposed by Vasconcelos in [10]. We integrated both specifications closely following his formulation. To our knowledge, ours is the...
first Maude implementation of a session-typed process language. Because typing
in [10] does not exclude deadlocks, we leverage built-in tools in Maude and ex-
cutable specifications to detect well-typed dead-locked processes. In our view,
these developments establish a promising starting point to the automated anal-
ysis of message-passing concurrency specifications.

As future work, we intend to adapt our equational theories to leverage the
confluence checker tool available in Maude. Additionally, we expect to extend
our executable specifications to perform behavioral analysis of the processes
that implement \textit{multiparty session types}, in the spirit of [7]. Likewise, we aim to
explore the automated analysis of communication correctness of an extension of
\$\pi\$ with \textit{higher-order} process communication, in which values can be abstractions
(functions from names to processes).

\textbf{Acknowledgements} This work has been partially supported by the Dutch Re-
search Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness
for Communicating Software).

We are grateful to Camilo Rocha for his useful suggestions on this work, and
to the anonymous reviewers for their careful reading and constructive remarks.

\textbf{References}

1. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott,
C.: All about Maude - a High-Performance Logical Framework: How to Specify,
Program and Verify Systems in Rewriting Logic. Springer-Verlag, Berlin, Heidel-
berg (2007), \url{https://doi.org/10.1007/978-3-540-71999-1}
2. Franco, J., Vasconcelos, V.T.: A concurrent programming language with refined
session types. In: Counsell, S., Núñez, M. (eds.) Software Engineering and Formal
Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-
Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised
(2013), \url{https://doi.org/10.1007/978-3-319-05032-4_2}
of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Com-
puter Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium
on Logic in Computer Science (LICS). CSL-LICS '14, Association for Comput-
ing Machinery, New York, NY, USA (2014), \url{https://doi.org/10.1145/2603088.
2603116}
4. Pitsiladis, G., Stefaneas, P.: Implementation of Privacy Calculus and Its Type
Checking in Maude: 8th International Symposium, ISOA 2018, Limassol, Cyprus,
org/10.1007/978-3-030-03421-4_30}
5. Ramírez Restrepo, C.A., Pérez, J.A.: Executable rewriting semantics for session-
based concurrency (Extended Version). Tech. rep., Universidad Javeriana / Univer-
sidad of Groningen (2022), \url{https://japerezp.github.io/files/wrla22-long.pdf}
Cambridge University Press (2003), \url{http://books.google.com/books?id=QkBL_7Vti1PgC}
to λ-, ν- and π- calculi. In: Futatsugi, K. (ed.) The 3rd International Workshop on
Rewriting Logic and its Applications, WRLA 2000, Kanzawa, Japan, September
9. Thati, P., Sen, K., Martí-Oliet, N.: An executable specification of asynchronous pi-
https://doi.org/10.1016/j.ic.2012.05.002
11. Verdejo, A., Martí-Oliet, N.: Implementing CCS in Maude 2. In: Gadducci, F.,
Montanari, U. (eds.) Proceedings Fourth International Workshop on Rewriting
Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002. Elec-
https://doi.org/10.1016/S1571-0661(05)82540-X
12. Viry, P.: Input/output for ELAN. Electronic Notes in Theoretical Computer Sci-
International Workshop on Rewriting Logic and its Applications, WRLA 1996,
Asilomar, USA, 1996
Maude as a library: an efficient all-purpose programming interface

Rubén Rubio\[0000−0003−2983−3404\]
Universidad Complutense de Madrid, Madrid, Spain
rubenrub@ucm.es

Abstract. We present a general and efficient programming interface to Maude from Python and other programming languages. All relevant Maude entities and operations are exposed in a documented object-oriented library to facilitate the integration and interoperability with other tools. This paper describes the design and implementation of the library, explains how to use it, and discusses some mature applications.

1 Introduction

Formal tools are more useful when they can cooperate and interact with the outside world through simple and well-defined interfaces. In addition to the traditional command-line interfaces, popular tools like the Z3 [7] and CVC4 [2] SMT solvers or the Lean [8] theorem prover are offering programming interfaces to their functionality from languages like C++ and Python. Some are even conceived as libraries in the first place, like the Spot [13] platform for LTL and \(\omega\)-automata. This laudable trend also reaches mainstream programming languages like C/C++, where the Clang compiler can be used as a library to inspect the abstract syntax tree of programs and control the different compilation phases.

Maude [5] is a high-performance logical and semantic framework based on rewriting logic [23]. Maude programs are collections of modules corresponding to rewriting-logic specifications, where states are terms in an equational logic that are transformed by the nondeterministic application of rewrite rules. Rewriting logic is reflective and Maude provides a universal theory where terms, modules, and other related concepts are represented as data that can be manipulated within the language. Several tools for analyzing Maude specifications and application-specific interactive interfaces have been written using these metaprogramming features. However, interacting with external tools and programming graphical interfaces is not so easy within Maude, and the interpreter has occasionally been extended with custom ad hoc extensions. Examples are the Maude Formal Environment [12], which interacts with external termination provers, and several analysis and visualization tools of the ELP group at Universitat Politècnica de València [1].

Maude is currently being used behind the scenes by some tools like the Tamarin prover [22] for security protocol verification, the \(\kappa\) semantic framework [24], and the heterogeneous tool set Hets [6], among others. All this software includes ad hoc code to run an instance of the Maude interpreter as a
separate process, issue commands to its standard input stream, and parse their answers. The IMaude agent of the InterOperability Platform (IOP) [21] follows the same approach to communicate with Maude, but then provides an abstraction for other user-defined agents of this framework to interact with the language. IMaude is used by the Pathway Logic Workbench [31], Mobile Maude [4], and the graphical interface to Maude-NPA [29], among others.

We present here an intuitive programming interface for Python and other programming languages that exposes almost all functionality of the Maude interpreter and some useful extensions. Unlike previous tools, these language bindings are directly linked with the Maude implementation, so several new possibilities and better performance are expected from this approach. The library comes with detailed documentation and API reference, and it has already been used in some relevant projects (see Section 6).

Its implementation relies on the Simple Wrapper and Interface Generator (SWIG) [10], so bindings can be generated for any language supported by this tool. However, only Python has been extensively tested and enhanced with language-specific adaptations to provide a more natural interface. The Python module is available at the Python Package Index (PyPI) and can be installed with the command `pip install maude`. Currently, the binding for Java has also been tested to a lesser degree and the those for other languages must be compiled from source. Instructions are available for some of them. In the following, we will focus on the Python flavor of the bindings for simplicity, although most information can be generalized to other languages.

This paper starts with a quick overview of the library in Section 2, which is further illustrated by a simple example in Section 3. Some advanced features are introduced in Section 4, and the implementation is described in Section 5. Finally, Sections 6 and 7 mention some applications and complete the discussion on related work in this introduction. More information can be found at github.com/fadoss/maude-bindings including documentation, examples, the API reference, and the source code of the library.

2 Overview of the library

In this section, we describe the design and overall organization of the language bindings, which coincide for all supported languages. However, we will stick to the Python instance for simplicity, as explained before.

The maude library exhibits all relevant Maude entities and operations as objects and methods of the target language. There are classes Term for terms, Module for modules, Sort for sorts, Symbol for symbols (or operators), Equation for equations, Substitution for substitutions, and so on. Most commands in the Maude interpreter are gathered as methods of the Term class, like reduce, rewrite, search, get_variants, and vu_narrow. Some commands that are not applied to a singular term like unify are available through the Module class. Operations that are reserved to the metalevel in the Maude interpreter are also implemented as regular methods, like iterating over the arguments of a term with

204
arguments, obtaining its least sort with `getSort`, its root symbol with `symbol`, or applying a substitution with `instantiate`, among others. Figure 1 shows a selection of the basic classes along with some methods that relate them.

A simple program that reduces the term $2 \times 3$ with the maude Python package and prints its result 6 to the terminal would look as follows

```python
import maude
maude.init()
m = maude.getModule('NAT')
t = m.parseTerm('2 * 3')
t.reduce()
print(t)
```

The first two instructions load the maude package and initialize it with the `init` function. This must be called before anything else in the library since it sets up some required resources and loads the Maude prelude. Everything in Maude takes place within modules, so a `Module` object is needed to begin with, and it can be obtained with the `getModule` or `getCurrentModule` functions. Typically, we will then parse a term with the `parseTerm` method and apply some operations to it. The contents of the module can also be inspected.

While the library offers enough resources to manipulate terms without resorting to the metalevel, moving through different levels of reflection is natively supported with the `upTerm` and `downTerm` methods of `Module`. For expressions in the Maude strategy language, these methods are called `upStrategy` and `downStrategy`. Moreover, a `Module` object can be obtained from its metarepresentation using the `downModule` function, while the converse operation can be achieved by simply reducing an `upModule` term in the `META-LEVEL` module.

In the next section, we illustrate the possibilities of the library through an example, giving further details on how to use it. Other advanced features are described in Section 4, and more information is available on the home page of the language bindings.
3 How to use the library, illustrated by an example

In this section, a toy interactive rewriter is implemented using the \texttt{maude} library, as an excuse to illustrate its usage and possibilities. Most of this example can be programmed directly in Maude using reflection, probably in a more verbose and complex manner, but the same procedures can be used when actual interaction with the outside world is pursued.

Our interactive prototype will repeatedly read commands from the terminal and reply to them. Implementing this kind of interface in Python is easy thanks to the standard \texttt{cmd} module. We only need to subclass the \texttt{cmd.Cmd} class and provide a method \texttt{do\_cmdname} to handle the command \texttt{cmdname}. Its full source code is available in the \texttt{inter.py} file of the bindings repository. As already explained, we should start by importing the library with \texttt{import maude} and initializing it with \texttt{maude.init()}. The \texttt{InteractiveRewriter} class holding the implementation of all commands in the interpreter can then be defined.\textsuperscript{1}

```python
import cmd
class InteractiveRewriter(cmd.Cmd):
    def __init__(self):
        super().__init__()  # base class constructor
        self.module = None  # current module
        self.term = None    # current term
```

For the moment, only two attributes are maintained, the current module and the term being rewritten. In order to bring modules to our scope, we need a \texttt{load} command to read them from Maude source files. Thus, we implement a method \texttt{do\_load} that essentially delegates on the \texttt{load} function of the library.

```python
def do_load(self, path):
    maude.load(path)
    self.module = maude.getCurrentModule()
    print('The current module is', self.module)
```

In addition, we set the current module using the \texttt{getCurrentModule} function, which gives the \texttt{Module} object for the last module that has been entered or explicitly selected in the file. Its name is printed in the screen by printing the object itself. However, we may want to select another module, for what we also provide commands to list the available modules and to select one of them.

```python
def do_list(self, _):
    for md in maude.getModules():
        print(md)

def do_select(self, name):
    self.module = maude.getModule(name)
```

\textsuperscript{1} The official documentation of the \texttt{cmd} module and other Python features that may appear is available at \texttt{docs.python.org}.

206
For example, assume we have a file `foo.maude` with the following module.

```latex
mod FOO-MODULE is
  sorts Foo Bar .
  subsort Bar < Foo .

  ops a b c : -> Bar [ctor] .
  op f : Foo Foo -> Foo [ctor] .
  op g : Foo -> Foo [ctor] .

  vars X Y : Foo .

  rl [swap] : f(X, Y) => f(Y, X) .
endm
```

After running the `inter.py` script with Python, the following command prompt will appear, where we can load `foo.maude` using the `load` command.

```plaintext
*** Interactive rewriter for Maude ***
```

```plaintext
IReW> load foo
The current module is FOO-MODULE
```

Term manipulation. At this point, we need to choose a term to start rewriting.

```python
def do_start(self, text):
    self.term = self.module.parseTerm(text)
```

Issuing the command `start t` makes `t` the current term in this session. We are not taking care about errors, but `self.term` would be `None` and error messages would have been printed if `text` could not be parsed as a term. For printing the syntax tree of this term, we can prepare a command `tree` by writing a method `do_tree` as before, which may simply call the following recursive function:

```python
def print_tree(term, indent=' '):
    print(f'{indent}{term.symbol()} : {term.getSort()}')
    for argument in term.arguments():
        print_tree(argument, indent + '    ')
```

The `print_tree` function starts by printing the top symbol of `term` and its sort with the appropriate indentation, and then proceeds recursively on the argument via the `arguments` method. Notice that strings prefixed by `f` in Python are formatted by replacing the expressions between brackets with their values. For example, we can show the syntax tree of `f(g(a), b)` in `FOO-MODULE` by selecting this term with `start` and calling the `tree` command.
Standard commands. One of the most useful commands in Maude is `reduce`.

```python
def do_reduce(self, _):
    nrew = self.term.reduce()
    print(f'Reduced to {self.term} in {nrew} rewrites.')
```

Methods like `reduce` and `rewrite` modify the term to which they are applied and return the number of rewrites instead. Since the original term is overwritten, if desired, it can be copied before with its `copy` method. Another command with a straightforward implementation is the strategy-rewriting command `srewrite`:

```python
def do_srewrite(self, text):
    strategy = self.module.parseStrategy(text)
    for result, nrew in self.term.srewrite(strategy):
        print(f'{result} in {nrew} rewrites')
```

Methods like `srewrite`, `search`, and `vu_narrow` that may produce multiple solutions return an iterator and do not alter the original term. As an example, we apply the strategy `swap ; next` to the current term with this command:

```
IRew> srewrite swap ; next
f(b, g(b)) in 2 rewrites
```

Applying rules. Since we are committed to implementing an interactive rewriter, we should provide a command `step` to execute a single rewrite on the current term.

```python
def do_step(self, label):
    results = []  # results of the rewriting step
    for k, (result, subs, ctx, rl) in enumerate(self.term.apply(label if label else None)):
        where = self.print_context(ctx, rl.getLhs())
        print(f'({k}) {result} by applying {rl} on {where} with {subs} with {subs}')
        results.append(result)

    self.select_one(results)
```

The `apply` method of `Term` calculates all possible rewrites with any rule labeled with the given string (or any rule at all if `None` is given instead). It provides an
iterator over the rewritten terms (result), the matching substitutions (subs) and contexts (ctx), and the applied rules themselves (rl). Contexts designate a single position in a term, and we see them here as functions that fill that position with the given term. In other words, ctx(subs.instantiate(rl.getLhs())) is the original term, and ctx(subs.instantiate(rl.getRhs())) is result. In this case, we hide in the print_context method how the context is processed since we will come back to this soon. Every result is accumulated in a list that is later passed to another unspecified method select_one that lets the user choose the next term.

IRew> start f(f(b, c), a)
IRew> step swap
(0) f(a, f(b, c))
   by applying rl f(X, Y) => f(Y, X) [label swap] .
   on top with X=f(b, c), Y=a
(1) f(f(c, b), a)
   by applying rl f(X, Y) => f(Y, X) [label swap] .
   on f(@, a) with X=b, Y=c

Select one of the options (0-1):

Matching and substitutions. In addition to the rules in the module, the interactive rewriter may be interested in experimenting with new rules, for what we add a command inline to apply inline rules.

IRew> start f(g(a), b)
IRew> inline g(X) => c
(0) f(c, b) in f(@, b) with X=a

There is a single option, done.

This command can be implemented by manually matching the left-hand side and replacing it with the right-hand side instantiated with the matching substitution. The match method of Term is the appropriate resource for this. It takes a pattern as an argument.

```python
def do_inline(self, text):
    arrow_index = text.index(’=>’)
    lhs = self.module.parseTerm(text[:arrow_index])
    rhs = self.module.parseTerm(text[arrow_index + 2:])

    results = []  # results of the inline rewriting

    for k, (subs, ctx) in enumerate(self.term.match(lhs, maxDepth=maude.UNBOUNDED)):
        result = ctx(subs.instantiate(rhs))
        where = self.print_context(ctx, lhs)
```

209
The first block in the method separates the left- and right-hand sides of the inline rule and parses them in the current module. Then, \texttt{lhs} is matched against the current term \texttt{self.term}, obtaining the matching substitution \texttt{subs} and context \texttt{ctx}. By default, matching is limited to the top symbol without extension, but \texttt{minDepth} and \texttt{maxDepth} can be set to fix maximum and minimum depths. The auxiliary method \texttt{print\_context} can be defined as follows.

```python
def print_context(self, ctx, lhs):
    var_name = f'<<PH>>:{lhs.getSort()}'
    var_term = self.module.parseTerm(var_name)

    ctx = ctx(var_term)

    return 'top' if ctx.isVariable() else str(ctx).replace(var_name, '@')
```

The context is instantiated with a placeholder variable \texttt{<<PH>>}. If the result is a variable, matching has happened on top. Otherwise, we replace the placeholder by the \texttt{@} sign for aesthetic reasons.

**Building terms and modules.** Once convinced with the new rule, we may want to add it to the current module with a new command \texttt{add}. Since modules are immutable in Maude, the library does not provide any direct resource to modify them, but we can always draw on the metalevel. This requires a more complex processing that we will carefully explain. Given the command \texttt{add l => r}, suppose both sides of the rule have been parsed into the variables \texttt{lhs} and \texttt{rhs}, like in the \texttt{inline} command. To modify the current module at the metalevel, we should obtain its metarepresentation by evaluating the \texttt{upModule} operator of the \texttt{META-LEVEL} module at the beginning of our \texttt{do\_add} method.

```python
ml = maude.getModule('META-LEVEL')

if self.metamodule is None:
    self.metamodule = 
        ml.parseTerm(f"upModule('{self.module}, false")")
    self.metamodule.reduce()
```

The module term is stored in the \texttt{metamodule} attribute of the interpreter for the next time. Remember that the metarepresentation of a module in Maude is an operator with a set-like argument for each type of declaration or statement in it. Hence, we will construct the metarepresentation of the new rule and insert it in the slot of rule statements. The first ingredient is the operator \texttt{rl=>}_{-[]}.
for unconditional rules in the universal theory of META-LEVEL. The \texttt{findSymbol}
method of \texttt{Module} allows finding operators in the module by their names and
signatures, given as a sequence of domain kinds and a range kind. These kinds
should be obtained first with the \texttt{findSort} and \texttt{kind} methods.

\begin{verbatim}
    term_kind = ml.findSort('Term').kind()
    rule_kind = ml.findSort('Rule').kind()
    attr_kind = ml.findSort('Attr').kind()
    rl_symb = ml.findSymbol('rl$_\Rightarrow_\&$[-].', (term_kind, rule_kind))
\end{verbatim}

Now, we only have to fill the gaps with the metarepresentations of \texttt{lhs} and \texttt{rhs},
and with the constant \texttt{none} for the attribute part of the statement. We parse this
latter constant with the \texttt{parseTerm} as usual, but providing the additional argument
\texttt{attr_kind} to restrict parsing to this kind and avoid ambiguities. Finally,
\texttt{Symbol}'s \texttt{makeTerm} constructs a term with a given sequence of arguments.

\begin{verbatim}
    rl_term = rl_symb.makeTerm((ml.upTerm(lhs), ml.upTerm(rhs), ml.parseTerm('none', attr_kind)))
\end{verbatim}

Syntactic sugar is provided for invoking the \texttt{makeTerm} method when a \texttt{Symbol}
object is applied as a function, as done with \texttt{rls_symb} below. Now, \texttt{rl_term}
must be inserted into the seventh argument of the metamodule, which holds the
set of rules in system and strategy modules. For simplicity, we assume that the
module is not a functional one. In order to add the rule to this set, we must
build a new term with the union operator \texttt{__} of \texttt{RuleSet}. The list of arguments
of the metamodule is obtained in the \texttt{mm_args} variable.

\begin{verbatim}
    rls_symb = ml.findSymbol('__', (rule_kind, rule_kind), rule_kind)
    mm_args = list(self.metamodule.arguments())
    mm_args[7] = rls_symb(mm_args[7], rl_term)
\end{verbatim}

Finally, the module is reassembled with the \texttt{makeTerm} method.

\begin{verbatim}
    self.metamodule = self.metamodule.symbol().makeTerm(mm_args)
\end{verbatim}

This new metamodule is converted to a \texttt{Module} object with the \texttt{downModule}
function, then assigned to the \texttt{module} attribute of the rewriter.

\begin{verbatim}
    self.module = maude.downModule(self.metamodule)
\end{verbatim}

Term objects in the library belong to a fixed module and they cannot operate
with entities from other modules, even if related by inclusion. Hence, if a term
was already set, we must reparse it in the new module.

\begin{verbatim}
    if self.term:
        self.term = self.module.parseTerm(str(self.term))
\end{verbatim}
We can check that the new command works by executing the interpreter.

IRew> start a
IRew> add a => c
The rule has been inserted.
IRew> step
(0) b by applying rl a => b [label next] . on top...
(1) c by applying rl a => c . on top with empty

Select one of the options (0-1): 1

Interoperability. To conclude and connect with the interoperability goals of the library, we will implement a command trs that exports the rules in the module into the standard TRS format, used by multiple verification tools.

IRew> load foo
IRew> trs
(VAR X:Foo Y:Foo)
(RULES
  f(X:Foo, Y:Foo) -> f(Y:Foo, X:Foo)
  a -> b
)

Since the format includes a VAR entry specifying the set of variables in the rules, we must calculate this set with the following straightforward recursive function.

def find_vars(term, varset):
    if term.isVariable():
        varset.add(term)
    else:
        for argument in term.arguments():
            find_vars(argument, varset)

Terms and most objects in the library can be safely used in dictionaries, sets, and other data structures since they support equality comparison and have hashing functions defined. The implementation of the trs command simply iterates over the rules printing them. Instead of the default conversion of terms into strings, we use the prettyPrint method that permits finer control on how they are printed. In particular, a zero argument causes terms to be printed in prefix form as required by the TRS format. Variables are also printed with an explicit type annotation.

def do_trs(self, _):
    varset = set()  # variables in the rules

    for rl in self.module.getRules():
        find_vars(rl.getLhs(), varset)
        find_vars(rl.getRhs(), varset)
In the general case, we should also take care of generating identifiers that respect the grammar of the TRS format and consider equations and structural axioms. The complete version of this example includes two more commands termination and confluence that automatically check these properties on the rules using the AProVE [16] and CSI [32] tools, with the generated TRS specification as input.

4 Advanced features

This section introduces two features of the library with useful applications and no direct correspondence in the Maude interpreter.

4.1 Rewrite graphs and model checking

Exploring the graph of all reachable states and transitions from a given initial term is useful for debugging, visualizing, and model checking Maude specifications. We can recursively build this graph in the library using the apply method or in Maude itself using the descent functions metaSearch or metaXapply, but this does not work for strategy-controlled models and such a common operation deserves to be a builtin feature. The language bindings offer two classes RewriteGraph and StrategyRewriteGraph to explore the rewrite graph of standard and strategy-controlled models, respectively. States are indexed by natural numbers starting from zero, the state’s term can be obtained with getStateTerm, its successors can be enumerated with getNextState, and other methods can be used to obtain the rule applied in each transition. This makes it easy to program a search or any other algorithm in Python that directly operates with the graph produced by Maude.

Moreover, a high-level interface to the Maude LTL model checker [15] and its extension for strategy-controlled systems [25] is provided through these graphs. This is more convenient than reducing, as usual, the modelCheck operator of the MODEL-CHECKER module.\(^2\) The modelCheck method of both graphs receives

\(^2\) Even though the strategy language is part of the official releases of Maude [11], the strategy-aware model checker [25] is not yet, but we have included it in the Maude build used for this library.
a term of sort \texttt{Formula} and returns a record indicating whether the formula holds and a counterexample that refutes if it does not. Counterexamples are described by a cycle and a path to it from the initial state, both given as lists of indices in the rewrite graph. One of the advantages of this approach is that the same graph can be used to model check multiple properties, hence saving the work required for the generation of the model in successive executions. Moreover, we can further process the graph or the counterexample when model checking has finished.

4.2 Custom special operators

Having overly shown that the \texttt{maude} module lets Python programmers evaluate Maude code in their programs, the interaction in the opposite direction, calling Python code from Maude, has not been explored yet.

User-defined and many predefined functions in Maude are specified with equations, but the prelude also includes some special operators whose behavior is internally defined in the C++ code of the interpreter. Most operators on the built-in types \texttt{Nat}, \texttt{Float}, \texttt{Qid}, and \texttt{String}, some polymorphic operators like equality \texttt{==}, and most descent functions in the \texttt{META-LEVEL} module are examples of special operators. Moreover, the Maude implementation has occasionally been extended ad hoc with new special operators, like in the Maude Formal Environment [12].

The language bindings allow declaring custom special operators whose behavior against equational reduction and/or rule rewriting is defined in the target language. In the Maude side, the operator should be declared first with the \texttt{special} attribute and its \texttt{id-hook SpecialHubSymbol} option. For instance, the gamma function that extends the factorial to real (and complex) numbers can be declared as the following \texttt{gamma} operator within a module.

\begin{verbatim}
op gamma : Float -> Float [ special ( id-hook SpecialHubSymbol ) ] .
\end{verbatim}

On the Python side, we have to define and register the callback that is invoked when a term with \texttt{gamma} on top is reduced or rewritten. This is done by subclassing the \texttt{maude.Hook} class and implementing its \texttt{run} method, and then calling the functions \texttt{connectEqHook} and/or \texttt{connectRlHook} to register an object of the class as the handler for the special operator.

\begin{verbatim}
class GammaHook(maude.Hook):
    def run(self, term, data):
        module = term.symbol().getModule()
        argument, = term.arguments()
        value = math.gamma(float(argument))
        return module.parseTerm(str(value))
\end{verbatim}
The `run` method receives the `term` that it should return reduced or rewritten. The implementation of `gamma` is directly provided in this case by the `math` module of the Python standard library. Finally, we install the hook for equational reduction with the `connectEqHook` function.

```python
hook = GammaHook()
maude.connectEqHook('gamma', hook)
```

After that, when we explicitly or implicitly reduce terms containing `gamma` in the library, `hook`’s `run` would be executed and we would obtain the desired number.

```
Gamma> 1.2 + gamma(6.5)
2.8908527781504438e+2
```

There is another argument, `data`, in the signature of `run` giving access to the `op-hook` and `term-hook` attributes of the special operator. Suppose we want to implement a custom predicate that tells whether a number is prime.

```python
op isPrime : Nat -> Bool [special (
    term-hook trueTerm (true)
    term-hook falseTerm (false)
)]
```

Using the above term hooks for the Boolean constants, we can define its `run` method by the expression

```python
data.getTerm('trueTerm' if test_prime(argument) else 'falseTerm')
```

for some `test_prime` Python function. While the same can be achieved by parsing the constants with `parseTerm`, the advantage of hooks is that keep working even if truth values are renamed, for example to `tt` and `ff`, in a module importation within Maude. Further details are explained in the documentation.

## 5 Implementation

The language bindings for Maude are implemented on top of the official implementation of Maude using some additional C++ code and the Simple Wrapper and Interface Generator (SWIG) [10], as illustrated in Figure 2. The desired programming interface is specified by selecting the classes, functions, and methods of the Maude implementation and the additional helper code that want to be exposed in the target language. Several languages like Python, Java, Lua, C#, Scheme, PHP, and JavaScript are supported, but only Python has been extensively tested and used in our case. From this specification, SWIG generates glue code in the selected language and in C, and this latter is then compiled into a binary module for the target language interpreter. This module is linked to the Maude implementation, which we have compiled as a shared library by adapting the build process. Indeed, we already did it to integrate Maude as a plugin for the language-independent model checker LTSmin [26]. Notice that
Maude does not provide an official stable interface and the bindings are using its internal classes, so the implementation should be adapted on every new release of Maude. Moreover, instead of using the official Maude implementation as is, the language bindings are linked with our extension including a model checker for systems controlled by strategies [25], which does not alter any other aspect of the Maude implementation.

A large part of the classes and methods of the interface are direct wrappers to the homonym classes and methods of the Maude implementation, but some are implemented on purpose to facilitate the interaction. For example, terms are represented in Maude sometimes as trees and sometimes as nodes in a directed acyclic graph, but this particularity is hidden to the library user in the uniform Term class. This type is backed by an auxiliary C++ class EasyTerm that chooses the appropriate representation and manage the conversion between them. Custom special operators in Section 4.2 are supported by a SpecialHubSymbol subclass of the Symbol type of the Maude implementation written on purpose to allow registering C functions as callbacks for the equational reduction and rule rewrite handling methods of the symbol. The connection with the target language is based on the directors feature of SWIG and the maude.Hook class, whose run method implemented in the target language can be called from the registered callbacks of the special operator.

When the Python interpreter executes the import maude statement, it loads the Python script generated by SWIG with the definition of all the classes and functions of the interface. The Python code loads the binary module that has been built from the SWIG-produced C code and the helper classes in the middle part of Figure 2. This module is linked with the dynamic library libmaude.so (.dylib in macOS or .dll in Windows) that contains the compiled code of the Maude implementation. Every object of the library in the target language holds a pointer to an object living in the Maude implementation, whose methods are invoked when the equivalent methods of the library are called. However, arguments may need to be translated in the process, for example, from a Python
list to a C++ vector. This is done by the glue code generated by the interface generator.

5.1 Performance considerations

Despite the cost of the translations mentioned in the last paragraph, the approach is expectedly much more efficient than the interaction through the text interface of the interpreter, especially when the results of the operations are frequently reused. We have executed some small experiments to compare the performance of reduction using (1) the maude Python library, (2) an I/O interaction that inputs reduce commands on a running Maude interpreter process and parse their results, and (3) a socket-based approach that communicates with a Maude-implemented TCP server that replies with the reduced forms of the terms it receives line by line. Reducing the constant 0 in the predefined module CONVERSION takes respectively (1) 3.21 $\mu$s, (2) 11.27 $\mu$s, and (3) 48.31 $\mu$s, so the best results are obtained with maude Python library. Moreover, the last two options have been implemented in the simplest way possible and assuming unrealistic constraints, so production-ready implementations would likely be more costly.

The performance improvement is more noticeable when reusing the output of previous operations. For example, consider an unrealistic Maude function fibonacci that expands a given list of integers by appending the sum of two leftmost numbers to the left. Calling this function iteratively on the result of the previous call takes the amount of time depicted in Figure 3 (in logarithmic scale) for an increasing number of iterations. In this case, the socket alternative has been specifically improved to store and reuse the result of the previous call, obtaining a performance that is comparable to the maude library. However, this is provided by the bindings out of the box. All these benchmarks are available at the bindings repository.
6 Some applications

Since the first version of the library was released, almost two years ago, it has been applied from small quick scripts to more relevant projects. Examples of the latter are the integration of Maude into a robotic environment and a unified interface to several external verification tools.

6.1 Integration of Maude into the Robot Operating System

The Robot Operating System (ROS) [9] is a collaborative robotic framework organized as a collection of nodes that deal with the different robotic tasks and communicate with each other by message passing. One of its most prominent components is the navigation module. The officially supported languages for programming ROS nodes are C++ and Python, but in a recent work [20] Maude has been used for programming an alternative path-planning node and experimenting with the inclusion of declarative languages in this context. The maude Python library provides the required connection between the communication infrastructure of ROS and the actual path-planning algorithm. Even though random access to the map is enabled by a custom special operator (see Section 4.2), the efficiency of the Maude-based planner is not comparable to the existing optimized C++ implementation, but the integration has been used for the formal verification of the latter. The more abstract Maude implementation of the navigation algorithm has been formally verified via model checking and SMT solving, and the correspondence with the official C++ planner has been established by differential testing with a huge collection of maps and paths.

In the process, the Maude library has been used for automating the evaluation of test cases, temporal properties, and verification conditions. For this latter case, we have extended the builtin SMT support in Maude with unsupported theories like arrays and uninterpreted functions. This extension and the other scripts using this library are available at [19].

6.2 The unified Maude model checker

The unified Maude model checking tool umaudemc [26] provides a uniform interface to the Maude LTL model checker [15] and several external model checkers for LTL, CTL, CTL*, and \( \mu \)-calculus on standard and strategy-controlled Maude specifications. This interface reads the input data of the model-checking problem, builds the corresponding Kripke structure, calls the appropriate backend, and shows the results to the user. Among the supported backends, there are LTSmin, NuSMV [3], pyModelChecking, Spot [13], and a builtin \( \mu \)-calculus implementation written in Python. The maude library and the rewrite graphs discussed in Section 4.1 are used to generate the models, evaluate the atomic propositions, parse the temporal formula, and so on. More recently, we have extended umaudemc for specifying probabilities on top of Maude specifications, and checking properties and calculating quantitative values by probabilistic model checking using PRISM [18] and Storm [17] or by statistical model checking through
simulation or the MultiVeSta tool [30]. By using external tools, we can efficiently support more logics and techniques while reducing the maintenance effort.

Moreover, umaudemc provides graphical and web-based interfaces for model checking, allows postprocessing the counterexamples, and generates graphical representations of the rewrite graphs in different formats. This tool can also be used as a library for application-specific model-checking interfaces [27,28].

7 Related work

As discussed in the introduction, several tools in the verification community maintain programming interfaces in addition to the traditional command-line ones, so that they can be used from other tools. Most applications interacting with Maude use ad hoc text-based communication with the interpreter, and the implementation of Maude has occasionally been extended to interact with external tools. The IMaude component of the IOP framework [21] is the closest precedent to this work in this context since it provides a reusable and application-agnostic interface between Maude and external programs. However, our language bindings replace the textual communication with the interpreter with a more efficient binary connection with its implementation, extend the available functionality, simplify the installation process, and can be used from potentially more programming languages.

On the other hand, Maude itself is being extended for a richer connection to the outside world. The notion of external objects used for accessing Internet sockets since Maude 2.0 has been applied to read and write files and standard streams in 3.0, to external processes in 3.1, and to time and filesystem operations in the latest development versions. External tools have also been integrated into Maude 2.7.1 with limited support for SMT solving via the CVC4 [2] and Yices2 [14] tools.

8 Conclusions

We have introduced a general-purpose efficient programming interface to Maude from Python and other programming languages. Almost all functionality of the Maude interpreter is available through these language bindings along with some useful additions. This work facilitates the interoperability between Maude and other tools, and tackles the claim for using Maude from external programs.

As future work, the library can be improved and extended in several directions, like adding native support for multiple interpreter sessions with separate databases through the infrastructure of metainterpreters, allowing the construction and manipulation of modules at the object level, or distributing compiled versions of the bindings for other languages. Moreover, there is currently no clear and explicit C/C++ interface, which can be very useful for applications where performance is a critical matter. Regarding applications, there are many possibilities for the library as we have suggested along the paper, from the elaboration of interfaces for specific frameworks to the development of more general tools.
Acknowledgments. I would like to thank Enrique Martin-Martín, Manuel Montenegro, Adrián Riesco, Juan Rodríguez-Hortalá, and Óscar Martín for the suggestions that brought this library into existence and their feedback to improve it. The first version of the bindings and extensions like custom operators were originally written for [20]. I also thank Narciso Martí-Oliet and Alberto Verdejo for their comments on this manuscript. This work was partially supported by the Spanish Ministry of Science and Innovation through projects TRACES (TIN2015-67522-C3-3-R) and ProCode (PID2019-108528RB-C22), and by the Spanish Ministry of Universities through the grant FPU17/02319.

References

8. Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, Automated


Author Index

A
Akleylek, Sedat 26

C
Chichester, Dalton 6

D
Dinh Tran, Duong 26
Du, Wei 6
Durán, Francisco 44

E
Escobar, Santiago 1, 26, 82, 143

H
Hansen, Simon Thrane 61

K
Kauffman, Raymond 6

L
Lin, Hai 6
Lynch, Christopher 6
López-Rueda, Raúl 82

M
Marshall, Andrew M. 6
Martínez, Daniela 44
Meadows, Catherine 6
Meseguer, José 82, 102, 122
Minh Do, Canh 143
Mödersheim, Sebastian 2

N
Narendran, Paliath 6
Nigam, Vivek 164

O
Ogata, Kazuhiro 26, 143
Otmani, Ayoub 26

P
Pérez, Jorge A. 182

R
Ramírez Restrepo, Carlos Alberto 182
Ravishankar, Veena 6
Riesco, Adrián 143
Rocha, Camilo 44
Rovira, Luís 6
Rozek, Brandon 6
Rubio, Rubén 3, 203
<table>
<thead>
<tr>
<th>Last Name</th>
<th>First Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Salaün</td>
<td>Gwen</td>
<td>4</td>
</tr>
<tr>
<td>Skeirik</td>
<td>Stephen</td>
<td>122</td>
</tr>
<tr>
<td>Talcott</td>
<td>Carolyn</td>
<td>164</td>
</tr>
<tr>
<td>Ölveczky</td>
<td>Peter Csaba</td>
<td>5, 61</td>
</tr>
</tbody>
</table>