Camilo Rocha (Ed.)

Rewriting Logic and its Applications

16th International Workshop, WRLA 2026

Part of the European Joint Conferences on Theory & Practice of Software (ETAPS 2026)

Turin, Italy, April 11th & 12th, 2026.

Informal Proceedings

Preface

This volume contains the preliminary proceedings of the 16th International Workshop on Rewriting Logic and its Applications (WRLA 2026). Previous editions of the workshop 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, Dublin (Ireland) 2020, Munich (Germany) 2022, and Luxembourg (Luxembourg) 2024.

Rewriting logic is a natural model of computation and an expressive semantic framework for concurrency, parallelism, communication, and interaction. It supports the specification of a wide range of systems and languages across diverse application domains and it also exhibits desirable 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 this workshop is to bring together researchers with a shared interest in rewriting logic and its applications, providing a forum to present recent work, discuss future research directions, and exchange ideas.

WRLA 2026 was held on April 11–12, 2026, in Turin, Italy, as a satellite event of the European Joint Conferences on Theory & Practice of Software (ETAPS). A total of 15 submissions were received, covering a wide range of topics related to rewriting logic and its applications. Each submission was reviewed by at least three program committee members or additional reviewers. After extensive discussion, the program committee selected 11 papers for presentation at the workshop. A selection of the accepted papers will appear in the post-proceedings, to be published in the Springer LNCS series, following the tradition of previous editions of the workshop.

Sincere appreciation is extended to all authors who submitted papers to the workshop, the invited speakers, and those who contributed tutorials. Equal thanks are due to the members of the program committee and the external reviewers for their careful work throughout the review process. As in previous editions, the suggestions provided by the members of the WRLA steering committee were both valuable and greatly appreciated. Finally, deep gratitude is expressed to the local organizers of ETAPS 2026, whose efforts made this workshop possible.

April, 2026 Camilo Rocha

Table of Contents

Program Committee

Kyungmin Bae	Pohang University of Science and Technology (POSTECH), South Korea
Francisco Durán	Universidad de Málaga, Spain
Santiago Escobar	Universitat Politècnica de València, Spain
Nao Hirokawa	JAIST, Japan
Alexander Knapp	Universität Augsburg, Germany
Temur Kutsia	RISC, Johannes Kepler University Linz, Austria
Alberto Lluch-Lafuente	Technical University of Denmark, Denmark
Dorel Lucanu	Alexandru Ioan Cuza University, Romania
Salvador Lucas	Universitat Politècnica de València, Spain
Narciso Martí-Oliet	Universidad Complutense de Madrid, Spain
José Meseguer	University of Illinois at Urbana-Champaign, USA
César Muñoz	NASA Langley, USA
Kazuhiro Ogata	JAIST, Japan
Carlos Olarte	Université Sorbonne Paris Nord, France
Peter Ölveczky	University of Oslo, Norway
Carlos Ramírez	Pontificia Universidad Javeriana, Colombia
Adrián Riesco	Universidad Complutense de Madrid, Spain
Christophe Ringeissen	INRIA, France
Camilo Rocha	Pontificia Universidad Javeriana, Colombia
Rubén Rubio	Universidad Complutense de Madrid, Spain
Carolyn Talcott	SRI International, USA

Additional Reviewers

A
Arusoaie, Andreea-Valentina

B
Bonchi, Filippo

F
Frey, Jonas

S
Sobocinski, Pawel

Generalization of Church-Rosser Strategies for Confluent Abstract Rewriting Systems of the Smallest Uncountable Cardinality

Ievgen Ivanov[1]

Taras Shevchenko National University of Kyiv, Kyiv, Ukraine
ivanov.eugen@gmail.com

Abstract. In the paper we investigate reduction strategies for proving convertibility in confluent abstract rewriting systems (ARS) of the smallest uncountable cardinality that generalize Church-Rosser strategies that are known to exist for countable confluent ARS. Under the Continuum Hypothesis (that is independent of ZFC set theory, if ZFC is consistent), the results obtained in the paper may be useful for guiding the design of (semi-)decision procedures for (restricted instances of) word problems for confluent ARS of the cardinality of the continuum that include rewriting systems on many structures of interest to computer science and applied mathematics (e.g. on the set of graphs with a fixed countable set of vertices, the set of real numbers, the set of finite-dimensional real vectors or matrices, etc.). Proofs of the main results were formalized using Isabelle proof assistant and are available in the supplementary material.

Keywords: Abstract rewriting system · Confluence · Reduction strategy · Word problem · Formal methods · Proof assistant.

1 Introduction

A *word problem* [7,22,2,19] asks whether two mathematical expressions are equivalent in the sense of a specific formal system. Rigorously formulated variants of a word problem have been studied during most of XX century [7,22] (beginning with the works by A. Thue and M. Dehn [22]) for different classes of algebraic structures and rewriting systems. Many of such problem variants turned out to be algorithmically undecidable in the general case, but some turned out to be decidable [7,22]. A highly abstract formulation of a word problem is considered in the *abstract rewriting theory* [2,10,32] inspired by the work [23] by M.H.A. Newman [4]. Abstract rewriting theory can be considered as a part of foundations for more concrete theories of rewriting and applications of rewriting to modeling and analysis of processes of computation and reasoning in different domains, e.g. [32,19,21,30,25].

A word problem for an *abstract rewriting system* (ARS) $(X, \to)$ [2,19] asks for any given elements $a, b \in X$ whether $a \leftrightarrow^* b$ holds (where $\leftrightarrow = \to \cup \to^{-1}$ and $*$ is used to denote the reflexive-transitive closure of a binary relation),

i.e. whether a, b belong to the equivalence relation generated by the reduction relation $\rightarrow$. The latter equivalence is also called a *convertibility* or *conversion* relation [15,5]. At this level of abstraction, the most well known condition that indicates that a word problem may be (in principle) tractable is the condition of *completeness* (confluence and strong normalization) of an ARS.

Completeness ensures that, in principle, the equivalence $a \leftrightarrow^* b$ can be checked [15] by randomly following the reduction relation $\rightarrow$ from the elements a and b till their normal forms (elements that cannot be reduced) and checking the equality of the normal forms of a and b (however, practical feasibility of this procedure depends on representation of elements of an ARS, effectiveness of reduction evaluation and equality checking, and other aspects that are out of scope of the abstract rewriting theory).

The latter fact can be generalized to non-terminating confluent ARS using the notion of a *Church-Rosser strategy* [16,32]. By adapting [16, Definition 6], we will say that a one-step Church-Rosser strategy for an ARS $(X, \rightarrow)$ is a functional subrelation $\rightarrow_s \subseteq \rightarrow$ such that the ARS $(X, \rightarrow)$ and $(X, \rightarrow_s)$ have the same set of normal forms and for every $a, b \in X$, if $a \leftrightarrow^* b$, then $\exists c \in X\ (a \rightarrow_s^* c \wedge b \rightarrow_s^* c)$. This implies that given $\rightarrow_s$ and a, b, if the equivalence $a \leftrightarrow^* b$ holds, then it can be (in principle) proved by (deterministically) following the relation $\rightarrow_s$ from the elements a and b, recording visited elements, and finding a common visited element c (reachable from both a and b) that will be called a *common descendant* [27] of a and b (it also has a role of a witness for $a \leftrightarrow^* b$).

Note that in a (functional) one-step Church-Rosser strategy $\rightarrow_s$ every element has no more than countably many descendants (elements of a reduction sequence $a \rightarrow_s a_1 \rightarrow_s a_2 \rightarrow_s \ldots$), so for a fixed a common descendants of a and b are restricted to a fixed countable set independent of b, which implies that such strategies have limited applicability to uncountable ARS. More precisely, in terms mentioned above, every *countable* confluent ARS *has* a one-step Church-Rosser strategy, but there exist *uncountable* confluent ARS that have *no* one-step Church-Rosser strategy (Proposition 2 in Section 2).

The above mentioned facts motivate investigation of generalizations of one-step Church-Rosser strategies that can provide a method for proving $a \leftrightarrow^* b$ (when it holds) for uncountable confluent ARS (here a proof method is understood in abstract sense at the level of abstract rewriting theory, practical uses of such methods require application-specific representation of elements and reduction rules). In this paper we consider this question for ARS of the smallest uncountable cardinality ($\aleph_1$) in the setting of Zermelo-Fraenkel set theory with the Axiom of Choice (ZFC). More precisely, our results are formalized in the Isabelle proof assistant using HOL logic [24,28], so they can be transferred to ZFC.

The idea of the proposed generalization is to relax the requirement of a Church-Rosser strategy to be deterministic, but restrict nondeterminism of a strategy so that its nondeterministic branchings are used in a proof of $a \leftrightarrow^* b$ as rarely as possible. The topic of this paper is related to the question of what spanning structure can be guaranteed to exist in a connected component of a

confluent ARS of the cardinality $\aleph_1$. We will use technical results on this question from the paper [12] on completeness of the decreasing diagrams method for proving confluence of ARS of the cardinality $\aleph_1$.

Under the Continuum Hypothesis (that is independent of ZFC set theory, if ZFC is consistent), the results obtained in the paper also hold for ARS of the cardinality of the continuum, so they may be useful for designing (semi-)decision procedures for (restricted instances of) word problems for special classes of confluent ARS of the cardinality of the continuum that include rewriting systems on many structures of interest to computer science and applied mathematics (e.g. ARS on the set of graphs with a fixed countable set of vertices, the set of real numbers, the set of finite-dimensional real vectors or matrices, etc.).

2 Preliminaries

A reader can assume that a background theory for the definitions and facts given below is Zermelo-Fraenkel set theory with the Axiom of Choice (ZFC) [8]. Proofs of auxiliary propositions formulated in the paper were formalized and machine-checked in the Isabelle proof assistant [1,24] using HOL logic [28,24]. Results formalized using HOL logic can be translated to the set theoretic setting using a process similar to the one described in [17,18,28].

A short reference table that describes notation used in Isabelle formalizations can be found in [12, Appendix A, table A.1]. A detailed description of Isabelle language can be found in documentation included in Isabelle software [1].

In the paper we will use the following notation:

- $\neg$, $\vee$, $\wedge$, $\Rightarrow$ denote logical connectives (negation, disjunction, conjunction, implication)
- $\mathbb{N}_0 = \{0, 1, 2, ...\}$ is the set of all non-negative integer numbers
- $\mathbb{Z}$ is the set of all integer numbers
- $\mathbb{R}$ is the set of all real numbers
- $|A|$ denotes the cardinality of a set A
- r^+ denotes the transitive closure of a binary relation r
- $r^=$, r^* denote the reflexive closure and the reflexive transitive closure of a binary relation r respectively (on a set that will be clear from the context)
- $r^{-1} = \{(y, x) \mid (x, y) \in r\}$
- $dom(r) = \{x \mid \exists y\ (x, y) \in r\}$ is the domain of a binary relation r.

Sometimes we will use the infix notation to denote that a pair (x, y) belongs to a binary relation R, for example: xRy.

We call an *abstract rewriting system* (ARS) [2,10,32,9] a pair $(X, \rightarrow)$, where X is a set and $\rightarrow \subseteq X \times X$. For convenience and compatibility with HOL formalization [13] (where types cannot be empty), throughout the paper we will implicitly assume that the set of elements of any ARS is *nonempty* ($X \neq \emptyset$), but the main propositions of the paper can be trivially extended to the case of empty ARS.

In an ARS $(X, \to)$ we will call $\to$ a *reduction relation* [2,9,29,10,32] and will call pairs $(a, b) \in \to$ *reduction steps*.

We will say that an ARS $(X, \to)$ is

- *countable* [6], if X is an at most countable set (i.e. there exists a surjective function from the set of natural numbers to X)
- *strongly normalizing* [10], if there is no infinite sequence of reductions

$$x_1 \to x_2 \to x_3 \to ...,$$

  where $x_i \in X$ for $i = 1, 2, 3, ...$
- *confluent* [10], if $\forall a, b, c \in X (a \to^* b \wedge a \to^* c \Rightarrow (\exists d \in X\ (b \to^* d \wedge c \to^* d))$
- *complete* [10], if it is confluent and strongly normalizing.

For an ARS $(X, \to)$ we will usually denote as $\leftrightarrow$ the symmetric closure of the relation $\to$ (i.e. $\leftrightarrow = \to \cup \to^{-1}$), and as $\leftrightarrow^*$ the reflexive transitive closure of $\leftrightarrow$ on X. Note that $\leftrightarrow^*$ is the equivalence relation generated by $\to$ [19], and $\leftrightarrow^*$ is also called the *convertibility* or *conversion* relation [15,5].

An (abstract) *word problem* for an ARS $(X, \to)$ [2,19] asks whether for any given elements $x, y \in X$ the relation $x \leftrightarrow^* y$ holds.

We will use the notation $NF(X, \to)$ for the set of all normal forms [15] (elements that cannot be reduced) in an ARS $(X, \to)$, i.e.

$$NF(X, \to) = X \backslash dom(\to).$$

We will also use the following definitions.

Definition 1 (based on [33, Definition 1]).
A one-step (reduction) strategy for an ARS $(X, \to)$ is a subrelation $\to_s \subseteq \to$ such that $NF(X, \to_s) = NF(X, \to)$, or, equivalently, if $dom(\to) = dom(\to_s)$.

Definition 2. *A one-step strategy $\to_s$ for an ARS $(X, \to)$*

- *is deterministic [16, subsection 2.3], if $\to_s$ is a functional relation, i.e.*

$$\forall a, b\ (a \to_s b \wedge a \to_s c \Rightarrow b = c)$$

- *preserves word problem (or preserves convertibility or preserves conversion), if $\leftrightarrow_s^* = \leftrightarrow^*$ (here the symbol $*$ is used to denote the reflexive transitive closure of a relation on the set X specifically).*

Definition 3 (based on [16, Definition 6]).
A one-step Church-Rosser (or 1-CR) strategy for an ARS $(X, \to)$ is a deterministic one-step strategy $\to_s$ for $(X, \to)$ such that

$$\forall a, b \in X\ (a \leftrightarrow^* b\ \Rightarrow\ \exists c \in X\ a \to_s^* c \wedge b \to_s^* c).$$

Remark 1. We will say that an ARS $(X, \to)$ *has* (or *has no*) strategy with particular properties (e.g. a 1-CR strategy), if there exists (or does not exist) a strategy with the given properties for $(X, \to)$.

Basic facts about 1-CR strategies are summarized in Proposition 1 below (in particular, item 3 of Proposition 1 below highlights a difference between *terminating* confluent (complete) and *non-terminating* confluent ARS).

Proposition 1.

1. *A relation $\to_s$ is a 1-CR strategy for an ARS $(X, \to)$ if and only if $\to_s$ is a deterministic one-step strategy for $(X, \to)$ that preserves word problem.*
2. *In a complete ARS every deterministic one-step strategy is a 1-CR strategy.*
3. *There exists an (acyclic) countable confluent ARS where not every deterministic one-step strategy is a 1-CR strategy.*

Proof (sketch). A machine-checked formal proof in Isabelle is available in [13, lines 2628–2634] (note that in [13] a relation $\to$ is denoted as r, a strategy $\to_s$ is denoted as s, and a nonempty set X implicitly corresponds to the type of components of elements of the relation r).

Proofs of items 1, 2 are straightforward and follow from definitions.

Item 3 can proved using an acyclic, countably infinite ARS illustrated in Figure 1 and formalized in [13, lines 641–646]. Alternatively, item 3 *without* the acyclicity requirement can be proved using a finite ARS with a cycle $(X, \to)$, where $X = \{1, 2\}$ and $a \to b \Leftrightarrow (a = 1 \wedge b \in X)$. □

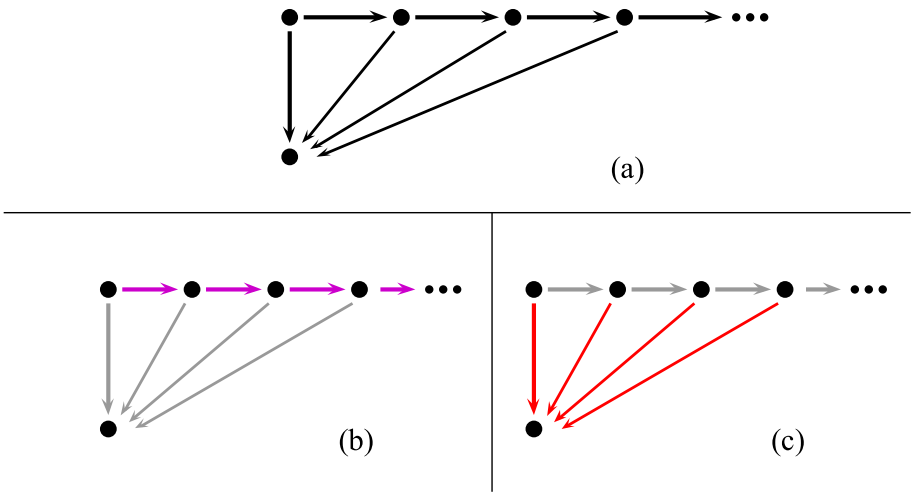


Fig. 1. An illustration for the proof of item 3 of Proposition 1. Subfigure (a): an example acyclic countable confluent ARS (dots denote ARS elements, black arrows denote reduction steps); subfigure (b): violet arrows denote a deterministic one-step strategy that is *not* a 1-CR strategy; subfigure (c): red arrows denote a 1-CR strategy.

Main relations between 1-CR strategies and the confluence property of an ARS are summarized in Proposition 2 below.

Proposition 2.

1. *A countable ARS is confluent if and only if it has a 1-CR strategy.*
2. *Every ARS that has a 1-CR strategy is confluent.*
3. *There exists an (uncountable) confluent ARS* $(X, \to)$ *such that* $|\to| = \aleph_1$ *and* $(X, \to)$ *has no 1-CR strategy.*

Proof (sketch). A machine-checked formal proof in Isabelle is available in [13, lines 2636–2642].

The formal proof of the "only if" part of item 1 is based on construction of a spanning (directed) pseudotree in each connected component of a countable ARS (here a directed pseudotree is a weakly connected directed graph where every node has at most one outgoing arc). The union of such pseudotrees forms a 1-CR strategy as illustrated in Figure 2.

The "if" part of item 1 and item 2 are straightforward and follow from definitions.

Item 3 can be proved by considering a widely known example [6, Footnote 4] of an ARS $(X, \to)$ that lacks the cofinality property [6,14]: X is taken to be the set of all finite subsets of ω_1 (the first uncountable ordinal) and $\to$ is defined by the rule $a \to b \cup \{x\}$ for all $a, b \in X$, $x \in \omega_1$. Then $(X, \to)$ is confluent, but has no 1-CR strategy. □

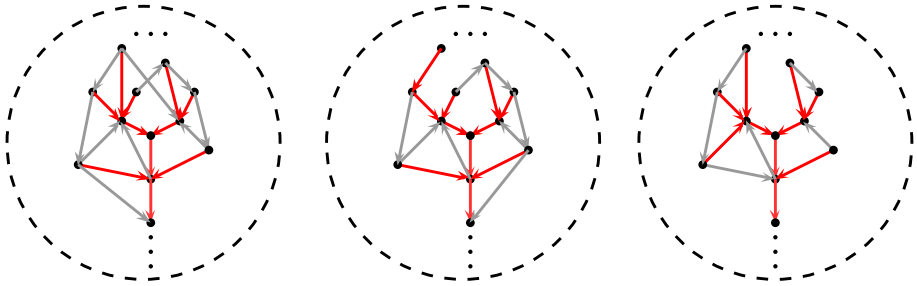


Fig. 2. An illustration for item 1 of Proposition 2. Dashed circles enclose connected components of a countable confluent ARS (considered as a directed graph). Dots denote elements of the ARS. Red and grey arrows denote reduction steps in the ARS. Red arrows denote reduction steps that belong to (possibly infinite) spanning (directed) pseudotrees that together form a 1-CR strategy for the ARS.

3 Generalization of one-step Chuch-Rosser strategies

From item 3 of Proposition 2 it follows that usual (deterministic) one-step Church-Rosser strategies have limited applicability to the problem of proving $a \leftrightarrow^* b$ in uncountable confluent ARS.

We propose an approach that allows one to overcome this issue by relaxing the requirement of a strategy to be deterministic. We will allow elements to have more than one direct successor in a strategy, e.g. $x \to_s y_1$ and $x \to_s y_2$ with $y_1 \neq y_2$, and will assume that traversal of descendants of a and b in such a nondeterministic strategy follows the breadth-first search principle (direct successors of a current element are visited before its further descendants).

However, since nondeterminism of a strategy may complicate search, we will aim to restrict its use. More specifically, in Definition 5 below (of a *k-branching 1-nCR strategy*) we will formalize a restriction that a strategy $\to_s$ is such that reduction steps $x \to_s y$ that have nondeterministic alternatives $x \to_s y' \neq y$ do *not* need to be followed more than k times during traversals of descendants of a and of b in search of a common descendant of a, b.

Let $\to_s$ be a one-step strategy for an ARS $(X, \to)$.

Firstly, let us adapt a few standard concepts from the graph theory to our setting as follows.

Definition 4. *1. A weight function for $\to_s$ is any function $w : (\to_s) \to \mathbb{N}_0$, i.e. w maps arcs $(a, b) \in \to_s$ in the directed graph $(X, \to_s)$ to non-negative integer numbers, so it makes $(X, \to_s)$ a weighted directed graph.*

2. The w-length of a (finite) reduction sequence $a_0 \to_s a_1 \to_s \ldots \to_s a_n$ in $(X, \to_s)$, where w is a weight function for $\to_s$, is the value $\sum_{i=0}^{n-1} w((a_i, a_{i+1}))$.

3. The ball of radius $k \in \mathbb{N}_0$ in $(X, \to_s)$ at an element $a \in X$ w.r.t. a weight function w (for $\to_s$) is the set of all elements $x \in X$ such that there exists a finite reduction sequence $a_0 \to_s a_1 \to_s \ldots \to_s a_n$ (for some $n \in \mathbb{N}_0$) such that its w-length does not exceed k and $a_0 = a$, $a_n = x$.

We will use a particular family of weight functions, parameterized by one-step strategies $\to_s$,

$$bw_{\to_s} : (\to_s) \to \mathbb{N}_0$$

that we will call *branching weight functions*.

For every one-step strategy $\to_s$ for the ARS $(X, \to)$ the weight function $bw_{\to_s}$ is defined as follows:

$$bw_{\to_s}((a, b)) = \begin{cases} 0, & \text{if } \{b' \mid a \to_s b'\} = \{b\}, \\ 1, & \text{if } \{b' \mid a \to_s b'\} \neq \{b\}, \end{cases}$$

i.e. $bw_{\to_s}$ maps a pair $(a, b) \in \to_s$ to the number 0, if b is a unique direct successor of a in $\to_s$, and maps (a, b) to 1 otherwise.

We propose the following generalization of 1-CR strategies.

Definition 5. *For $k \in \mathbb{N}_0$, a k-branching one-step nondeterministic Church-Rosser (or k-branching 1-nCR) strategy for an ARS $(X, \to)$ is a one-step strategy $\to_s$ for $(X, \to)$ such that*

$$\forall a, b \in X \ (a \leftrightarrow^* b \Leftrightarrow B_k(a) \cap B_k(b) \neq \emptyset \ ),$$

where $B_k(x)$ denotes the ball of radius k in $(X, \to_s)$ at an element x w.r.t. the weight function $bw_{\to_s}$ (see Figure 3 below).

Remark 2. The sets $B_k(x)$ in Definition 5 can be explicitly expressed as:

$$B_k(x) = \{\ y \in X \mid \exists n \in \mathbb{N}_0\ \exists a_0, a_1, ..., a_n \in X \\ x = a_0 \to_s a_1 \to_s ... \to_s a_n = y\ \wedge \\ \wedge\ |\{i \in \{0, 1, ..., n-1\} \mid \exists b, b'\ (b \neq b' \wedge a_i \to_s b \wedge a_i \to_s b')\}| \leq k\ \}.$$

Remark 3. Definition 5 means that in order to prove $a \leftrightarrow^* b$ using a k-branching 1-nCR strategy $\to_s$, it is sufficient to explore all reduction sequences in $(X, \to_s)$ that begin at a and b and have $bw_{\to_s}$-length no more than k (i.e. contain no more than k reduction steps that start at an element with non-unique successor in $(X, \to_s)$), and find a common descendant of a, b in them.

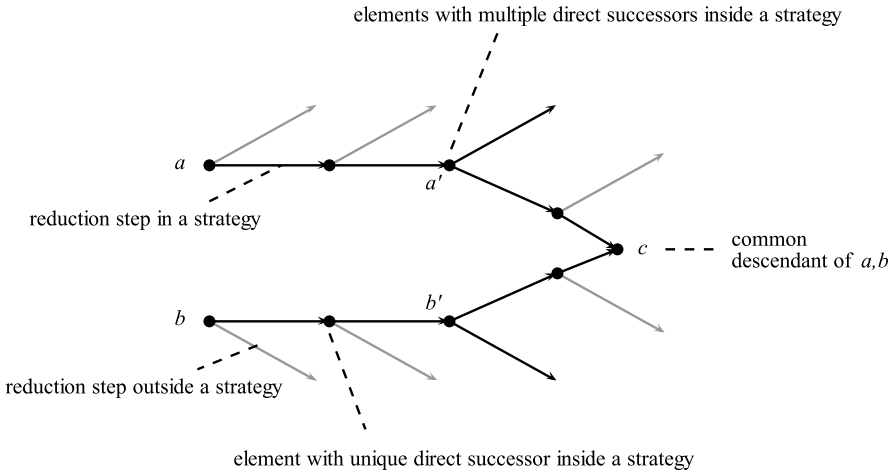


Fig. 3. An illustration for Definition 5. Black dots denote elements of an ARS. Grey and black arrows denote reduction steps in an ARS. Black arrows denote reduction steps that belong to a k-branching 1-nCR strategy in an ARS, grey arrows denote reduction steps that do *not* belong to the given strategy. The k-branching 1-nCR strategy condition: every pair of convertible elements (e.g. a, b in the illustration), can be joined using two reduction sequences that belong to the strategy and contain no more than k reduction steps that originate at elements with non-unique direct successor in the strategy (e.g. a', b' in the illustration).

Proposition 3. *For any ARS $(X, \to)$ and any $k \in \mathbb{N}_0$, every k-branching 1-nCR strategy in $(X, \to)$ preserves word problem.*

A machine-checked formal proof of Proposition 3 in Isabelle is available in [13, lines 2644–2662]. The proof is straightforward and follows from definitions. □

A link between the usual 1-CR strategies and 1-nCR strategies (that are intended to generalize 1-CR strategies) is described in the following proposition.

Proposition 4. *Let $(X, \to)$ be an ARS. Then the following two conditions are equivalent for any relation $\to_s \subseteq X \times X$:*

1. *$\to_s$ is a 1-CR strategy for $(X, \to)$*
2. *$\to_s$ is a deterministic 0-branching 1-nCR strategy for $(X, \to)$.*

A machine-checked formal proof of Proposition 4 in Isabelle is available in [13, lines 2664–2679]. The proof is straightforward and follows from definitions. □

Remark 4. A 0-branching 1-nCR strategy is *not* required to be deterministic (for example, in the ARS $(\{0,1,2\}, \to)$, where $\to = \{(0,1),(1,2),(2,1),(1,0)\}$, the relation $\to_s = \to$ is a nondeterministic 0-branching 1-nCR strategy). However, the Axiom of Choice implies that every ARS has a deterministic one-step strategy, and, in particular, if $\to_s$ is a 0-branching 1-nCR strategy in an ARS $(X, \to)$, then $(X, \to_s)$ has some deterministic one-step strategy $\to'_s \subseteq \to_s$. It is straightforward to check that any such $\to'_s$ is a 1-CR strategy in $(X, \to)$. Thus whenever an ARS $(X, \to)$ has a 0-branching 1-nCR strategy, it also has a 1-CR strategy.

The following analog of item 1 of Proposition 2 holds for 1-nCR strategies.

Proposition 5. *Let $(X, \to)$ be a countable ARS.*
Then $(X, \to)$ is confluent if and only if $(X, \to)$ has a 0-branching 1-nCR strategy.

Proof (sketch). A machine-checked formal proof in Isabelle is available in [13, lines 2681–2685]. The "if" part is straightforward and follows from definitions. The "only if" part follows from Proposition 4 and item 1 of Proposition 2. □

4 Main result

Below we give the main theorem of this paper that is a direct analog of Proposition 5 (applicable to countable ARS) for ARS that have a reduction relation of the cardinality $\aleph_1$ or less (where the "0-branching" condition is replaced with the "1-branching" condition).

Theorem 1. *Let $(X, \to)$ be an ARS such that $|\to| \leq \aleph_1$.*
Then $(X, \to)$ is confluent if and only if $(X, \to)$ has a 1-branching 1-nCR strategy.

Proof (sketch). A machine-checked formal proof in Isabelle is available in [13, line 2693 and below] (note that in [13] a relation $\to$ is denoted as r, a strategy is denoted as s, and a nonempty set X implicitly corresponds to the type of components of elements of r).

The "if" part is straightforward and follows from definitions.

The proof of the "only if" part is based on the idea described below.

Consider a fixed set $U \neq \emptyset$ and a positive integer k. Denote as 2^{U^k} the set of all k-ary relations on U and as $\mathcal{P}_f(A)$ the set of all finite subsets of a set A.

Let us say that a set $\mathcal{R} \subseteq 2^{U^k}$ is *characterized by finite extensions*, if for every $A \in 2^{U^k}$,

$$A \in \mathcal{R} \;\Leftrightarrow\; \mathcal{P}_f(A) \cap \mathcal{R} \text{ is a cofinal subset of } \mathcal{P}_f(A) \text{ w.r.t. } \subseteq$$

i.e.

$$A \in \mathcal{R} \;\Leftrightarrow\; (\forall F \in \mathcal{P}_f(A) \;\; \exists F' \in \mathcal{P}_f(A) \cap \mathcal{R} \quad F \subseteq F').$$

Confluence is an example of a property of interest to the abstract rewriting theory that induces a set characterized by finite extensions: using induction on the cardinality of a finite subrelation of a confluent relation it can be shown that the set $CR(U) = \{r \in 2^{U \times U} \mid (U, r) \text{ is a confluent ARS}\}$ is characterized by finite extensions.

Let us say that a set $\mathcal{R} \subseteq 2^{U^k}$ is *closed under chain unions*, if for every $\mathcal{C} \subseteq \mathcal{R}$:

$$(\forall A, B \in \mathcal{C} \;\; A \subseteq B \vee B \subseteq A) \;\; \Rightarrow \;\; \left(\bigcup \mathcal{C}\right) \in \mathcal{R}.$$

Also, let us say that a set $\mathcal{R}' \subseteq 2^{U^k}$ is the *chain-union closure* of $\mathcal{R} \subseteq 2^{U^k}$, if $\mathcal{R}'$ is the least (in the sense of $\subseteq$) subset of 2^{U^k} such that

$$\mathcal{R} \subseteq \mathcal{R}' \text{ and } \mathcal{R}' \text{ is closed under chain unions.}$$

Using an argument reminiscent to the proof of Iwamura's lemma about directed sets [20] it can be shown that if a set $\mathcal{R} \subseteq 2^{U^k}$ is characterized by finite extensions and contains the empty relation (i.e. $\emptyset \in \mathcal{R}$), then $\mathcal{R}$ is the chain-union closure of $\{A \in \mathcal{R} \mid A \text{ is finite}\}$.

In particular, since $CR(U)$ is characterized by finite extensions and $\emptyset \in CR(U)$, the set $CR(U)$ is the chain-union closure of $\{r \in CR(U) \mid r \text{ is finite}\}$. This statement can be viewed as an induction principle for proving completeness of confluence criteria. However, we do not apply it directly to the "only if" part of the current theorem. Instead, we constructed a refinement of such an induction principle (in essence, by imposing a number of technical conditions on elements of a chain that approximate a confluent relation) that implies that in a confluent relation $\rightarrow$ of cardinality $\aleph_1$ a connected component that lacks the cofinality property can be represented as a union of a chain of countable connected components of confluent relations that has a spanning subgraph of the form illustrated in Figure 4 (formed by red and green arrows). The cardinality assumption $|\rightarrow| \leq \aleph_1$ is used in the proof to guarantee countability of members of this chain. Then the union of such spanning subgraphs for connected components of an ARS that lack the cofinality property and of spanning (directed) pseudotrees for components that have the cofinality property is shown to form a 1-branching 1-nCR strategy.

Details of the proof are available in formal lemmas *lem_ex1b1CR_confl_w1* and *lem_exspn_1ndccr_refl_ccr_scfw1* at lines 2485–2550 and 2051–2389 in [13].

□

Corollary 1. *In ZFC + CH (Continuum Hypothesis), every confluent ARS* $(X, \to)$ *with* $|\to| \leq |\mathbb{R}|$ *has a 1-branching 1-nCR strategy.*

Remark 5. In practical sense the CH assumption is not very restrictive, since, assuming ZFC is consistent, for any concrete $(X, \to)$ it is *not* provable in ZFC that $(X, \to)$ is a confluent ARS with $|\to| \leq |\mathbb{R}|$ that lacks a 1-branching 1-nCR strategy.

Remark 6. Elements with non-unique successor in a 1-branching 1-nCR strategy (e.g. a', b' in Figure 3) can be thought of as kinds of "*virtual normal forms*": in Figure 3, a', b' are determined uniquely by a and b respectively (within a strategy). Of course, they are not true normal forms (they are reducible), but Theorem 1 suggests that (when they exist) they can serve as important milestones during search of a common descendant of a, b.

In particular, Theorem 1 suggests that when a relation $\to$ of cardinality $\aleph_1$ (or cardinality $|\mathbb{R}|$ under CH) in an abstract word problem $a \leftrightarrow^* b$ is known to be confluent (or can be enforced to be confluent like in completion procedures, e.g. using the decreasing diagrams method [26] for analysis of local peaks), when designing a (semi-)decision procedure for (possibly restricted instances of) this problem, one can try to:

1. select a set of elements that will serve as virtual normal forms (elements with non-unique successor in a 1-branching 1-nCR strategy)
2. design a deterministic procedure to reach them from a given element (perform "*virtual normalization*")
3. design a procedure to check joinability of such elements.

Note that for 1-nCR strategies it is *not* required that any given element (e.g. a virtual normal form mentioned above) has finitely many direct successors in a strategy since this would imply a restriction on the cardinality of the set of descendants of any element in a strategy like in the case of deterministic strategies. In fact, in the case of uncountable ARS an element may have uncountably many direct successors in a 1-nCR strategy, so, in practical sense, search in such a strategy should be understood *not* as enumeration of successors of an element, but as logical analysis of properties of successors of an element (e.g. using automated theorem proving techniques).

5 Examples

The examples given below illustrate the notion of a k-branching 1-nCR strategy.

Example 1. Let $(X, \to)$ be an ARS where $X = \mathbb{R}^2$ (i.e. elements are 2-dimensional real vectors) and $\to$ is defined by the rule

$$(x, y) \to (ky, kx) \text{ for every } k \in \mathbb{R} \text{ such that } k > 0,$$

or, more formally,

$$\to = \{((x, y), (x', y')) \in \mathbb{R}^2 \times \mathbb{R}^2 \mid \exists k \in \mathbb{R}\ k > 0 \wedge x' = ky \wedge y' = kx\},$$

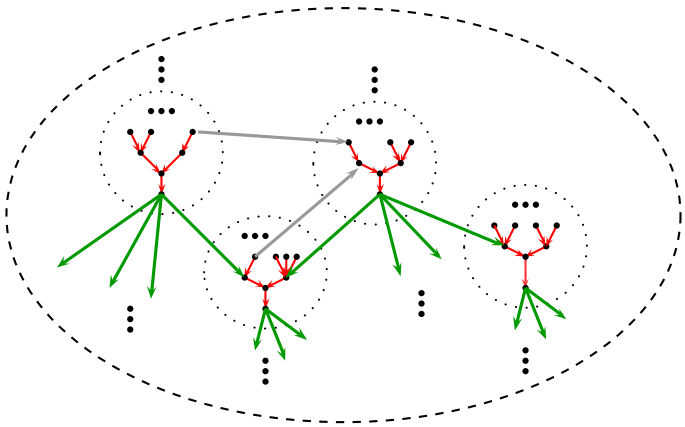


Fig. 4. An illustration for the "only if" part of Theorem 1. Dashed ellipse encloses a connected ARS of the cardinality $\aleph_1$ that lacks cofinality property, where grey, red, green arrows denote reduction steps colored in such a way that, *ignoring* grey arrows, red arrows start at elements with unique direct successor and green arrows start at elements with non-unique direct successor. This connected component lacks a spanning (directed) pseudotree, but has a spanning structure that consists of all red and green arrows that forms a 1-branching 1-nCR strategy for this component (node pairs can be joined without using green arcs more than once on paths to their common descendant).

i.e. components of a vector (x, y) are scaled using a positive real coefficient k and exchanged. It is straightforward to check that $(X, \to)$ is confluent, but *not* terminating (e.g. $(1, 1) \to (2, 2) \to (3, 3) \to ...$).

Consider a relation $\to_s \subseteq X \times X$ such that $(x, y) \to_s (x', y')$ if and only if the following two conditions hold:

1. $(x, y) \to (x', y')$
2. if $x > y$, then $(x', y') = (y, x)$.

Then $\to_s$ is 1-branching 1-nCR strategy. Here virtual normal forms mentioned in Remark 6 are vectors (x, y) with $x \leq y$. If a pair of such vectors has a common descendant in $\to$, it can found immediately in 1 reduction step by finding an appropriate positive scaling coefficient k in the definition of $\to$ above.

Despite being uncountable, this ARS also has a 0-branching 1-nCR strategy, and, in fact, it has a 1-CR strategy, e.g. $\to'_s \subseteq X \times X$ such that $(x, y) \to'_s (x', y')$ if and only if the following two conditions hold, where $||.||$ is the Euclidean norm:

1. if $x > y$ or $||(x, y)|| = 0$, then $(x', y') = (y, x)$
2. if $x \leq y$ and $||(x, y)|| \neq 0$, then $(x', y') = (y, x)/||(y, x)||$.

Still, it follows from item 3 of Proposition 2 and Remark 4 that there *do* exist uncountable ARS that have *no* 0-branching 1-nCR strategy.

Example 2. Assume the Continuum Hypothesis (CH) and consider ARS $(X, \to)$ based on [6, Footnote 4], where X is the set of all finite subsets of $\mathbb{R}$ (i.e. elements of the ARS are finite sets of real numbers) and

$$\to = \{(a, b) \in X \times X \mid \exists x \in \mathbb{R}\ b = a \cup \{x\}\}.$$

Then $(X, \to)$ satisfies the conditions of Theorem 1, is confluent, and has a 1-branching 1-nCR strategy.

The word problem for this ARS is trivial ($a \leftrightarrow^* b$ holds for all $a, b \in X$, i.e. $(X, \to)$ is weakly connected as a graph), but the existence of a 1-branching 1-nCR strategy may not be immediately obvious.

In this example an explicit 1-branching 1-nCR strategy can be given as follows (without assuming CH). Let $F : [0, 1) \to X$ be a surjective function (such a function exists, because $|X| = |\mathbb{R}|$, but for a more explicit construction of F see Remark 7 below).

For every $x \in \mathbb{R}$ denote $F_p(x) = F(x - \lfloor x \rfloor)$, where $\lfloor x \rfloor = \max\{n \in \mathbb{Z} \mid n \leq x\}$.

Let $\to_s \subseteq X \times X$ be a relation such that for all $a, b \in X$, $a \to_s b$ if and only if there exists $x \in \mathbb{R}$ such that the following two conditions hold:

1. $b = a \cup \{x\}$
2. if $a \neq \emptyset$, $F_p(\max a) \backslash a \neq \emptyset$, and $\min(F_p(\max a) \backslash a) < \max a$, then $x = \min(F_p(\max a) \backslash a)$.

Then $\to_s$ is a one-step strategy for $(X, \to)$.

For $k = 0, 1$ denote as $B_k(x)$ the ball of radius k in $(X, \to_s)$ at an element x w.r.t. $bw_{\to_s}$. Let $a, b \in X$. It is straightforward to check that there exist $a' \in B_0(a)$ and $b' \in B_0(b)$ such that $\{c \mid a' \to_s c\} = \{c \mid a' \to c\}$ and $\{c \mid b' \to_s c\} = \{c \mid b' \to c\}$. Since F is surjective, there exists $t \in [0, 1)$ such that $a' \cup b' = F(t)$. Let $u = t + \min\{n \in \mathbb{N}_0 \mid \forall x \in a' \cup b'\ x < n\}$. Then $a' \to_s a' \cup \{u\}$, $b' \to_s b' \cup \{u\}$. Moreover, $u \notin a' \cup b'$ and $u = \max(a' \cup \{u\}) = \max(b' \cup \{u\})$. Then

$$F_p(u) \backslash (a' \cup \{u\}) = F(t) \backslash (a' \cup \{u\}) = (a' \cup b') \backslash (a' \cup \{u\}) = b' \backslash a',$$

$$F_p(u) \backslash (b' \cup \{u\}) = F(t) \backslash (b' \cup \{u\}) = (a' \cup b') \backslash (b' \cup \{u\}) = a' \backslash b'.$$

Then it is straightforward to check that $a' \cup b' \cup \{u\} \in B_0(a' \cup \{u\}) \cap B_0(b' \cup \{u\})$. Then $B_1(a) \cap B_1(b) \neq \emptyset$. From this it follows that $\to_s$ is a 1-branching 1-nCR strategy.

Remark 7. An example of a surjective function F from $[0, 1)$ to the set of all finite subsets of $\mathbb{R}$ mentioned in Example 2 can be described as follows.

Let us fix a surjective function $g : \mathbb{R} \to \mathbb{R}^2$ (g can be required to be *continuous* and can be obtained using one of *space-filling curve* constructions [31]).

For every $x \in \mathbb{R}$ denote as $g_0(x)$ and $g_1(x)$ components of $g(x)$, i.e. real numbers such that $g(x) = (g_0(x), g_1(x))$. For every $x \in \mathbb{R}$ denote

$$F_0(x) = \{g_1(g_0^{(i)}(x)) \mid i \in \mathbb{N} \wedge i \leq g_1(x)\},$$

where $\mathbb{N} = \{1, 2, 3, ...\}$ and $g_0^{(i)}(x) = \underbrace{g_0(g_0(...(g_0(x)...))}_{i}$.

As in Example 2, let X be the set of all finite subsets of $\mathbb{R}$. Then it is straightforward to check that F_0 is a surjective function from $\mathbb{R}$ to X.

Let $f : [0, 1) \to \mathbb{R}$ be a surjective function, e.g.

$$f(t) = \frac{1}{1-t} \sin\left(\frac{1}{1-t}\right)$$

for $t \in [0, 1)$ and let $F : [0, 1) \to X$ be the composition of F_0 and f, i.e.

$$F(t) = F_0(f(t))$$

for all $t \in [0, 1)$. Then F is a surjective function from $[0, 1)$ to X.

6 Formalization in proof assistant

We formalized propositions from Sections 2, 3 and Theorem 1 from Section 4 (main theorem) using Isabelle 2025 proof assistant [1,24] and HOL logic in the file `NCRS_N1.thy` [13] that consists of about 2700 lines. The theory `NCRS_N1.thy` depends on a previous theory `DCRN1.thy` [11] (supplementary material for the paper [12]) and reuses parts of proofs of technical results from `DCRN1.thy`.

Instructions on re-checking formal proofs are given in Appendix at the end of this paper.

Unlike most usual formalizations of results from the rewriting theory in Isabelle/HOL, our formalization significantly uses basic facts about ordinal numbers and cardinalities of sets. For this reason, we use a formalization of ordinal and cardinal numbers *HOL-Cardinals* described in [3] as a dependency.

Formalizations of main definitions from Sections 2, 3 in [13, lines 25–65] have the following forms:

```
— s is a one-step strategy for ARS (U,r)
abbreviation is_1step_strategy :: "'U rel ⇒ 'U rel ⇒ bool"
where
  "is_1step_strategy s r ≡ s ⊆ r ∧ Domain s = Domain r"

— s is a deterministic one-step strategy for ARS (U,r)
definition is_det_1step_strategy :: "'U rel ⇒ 'U rel ⇒ bool"
where
  "is_det_1step_strategy s r ≡ is_1step_strategy s r ∧ single_valued s"

— s is a one-step Church-Rosser strategy for ARS (U,r)
definition is_CR_1step_strategy :: "'U rel ⇒ 'U rel ⇒ bool"
where
  "is_CR_1step_strategy s r ≡ is_det_1step_strategy s r
                        ∧ ( ∀ a b. (a,b) ∈ (r ∪ r^-1)^* ⟶
                             (∃ c. (a,c) ∈ s^* ∧ (b,c) ∈ s^*) )"
```

— `ai` is a (finite) reduction sequence of `n` elements in ARS `(U,s)`

```
definition is_finrseq :: "'a rel ⇒ nat ⇒ (nat ⇒ 'a) ⇒ bool"
where
  "is_finrseq s n ai ≡ (∀ i < n. (ai i, (ai (Suc i))) ∈ s)"
```

— `w`-length of a reduction sequence `ai` of `n` elements

```
definition wlen :: "('a ⇒ 'a ⇒ nat) ⇒ (nat ⇒ 'a) ⇒ nat ⇒ nat"
where
  "wlen w ai n ≡ ( ∑i<n. w (ai i) (ai (Suc i)) )"
```

— ball of radius `k` at `a` in ARS `(U,s)` w.r.t. weight function `w`

```
definition wball :: "'a rel ⇒ ('a ⇒ 'a ⇒ nat) ⇒ nat ⇒ 'a ⇒ 'a set"
where
  "wball s w k a = {x. ∃ n ai. is_finrseq s n ai
                          ∧ wlen w ai n ≤ k ∧ ai 0 = a ∧ ai n = x }"
```

— `(bw s)` is a branching weight function for ARS `(U,s)`

```
definition bw :: "'a rel ⇒ 'a ⇒ 'a ⇒ nat"
where
  "bw s a b ≡ (if {b'. (a,b') ∈ s} = {b} then 0 else 1)"
```

— `s` is a `k`-branching one-step nondeterministic Church-Rosser strategy in ARS `(U,r)`

```
definition is_kb_1nCR_strategy :: "nat ⇒ 'U rel ⇒ 'U rel ⇒ bool"
where
  "is_kb_1nCR_strategy k s r ≡ is_1step_strategy s r
      ∧ ( ∀ a b. (a,b) ∈ (r ∪ r^-1)^* ⟷
                 wball s (bw s) k a ∩ wball s (bw s) k b ≠ {} )"
```

Also, in [13] the following formal definition of the confluence property from [11, lines 99–100] is used (here r denotes a binary relation):

```
definition confl_rel
  where "confl_rel r ≡ (∀a b c. (a,b) ∈ r^* ∧ (a,c) ∈ r^* ⟶
                           (∃ d. (b,d) ∈ r^* ∧ (c,d) ∈ r^*) )"
```

Formalization of Proposition 1 in [13, lines 2628–2633] has the following form:

```
lemma lem_prop1:
    "∀ s r. is_CR_1step_strategy s r = (is_det_1step_strategy s r ∧
                                        (s ∪ s^-1)^* = (r ∪ r^-1)^*)"
and "∀ r. ( confl_rel r ∧ wf (r^-1) ⟶
        (∀ s. is_det_1step_strategy s r ⟶ is_CR_1step_strategy s r) )"
and "∃ r::nat rel. ( confl_rel r ∧
    ¬ ( ∀ s. is_det_1step_strategy s r ⟶ is_CR_1step_strategy s r ) )"
```

Formalization of Proposition 2 in [13, lines 2636–2641] has the following form:

```
lemma lem_prop2:
    "∀ r::'a rel. ( |{x::'a. True}| ≤o |{x::nat. True}| ⟶
                    ( confl_rel r = (∃ s. is_CR_1step_strategy s r) ) )"
and "∀ r. ( (∃ s. is_CR_1step_strategy s r) ⟶ confl_rel r )"
```

```
and "∃ r::nat set set rel. |r| =o cardSuc |{x::nat. True}| ∧
                    confl_rel r ∧ ( ¬ (∃ s. is_CR_1step_strategy s r) )"
```

Formalizations of Propositions 3, 4, 5 in [13, lines 2644–2685] have the following forms:

```
lemma lem_prop3:
fixes r::"'U rel" and k::nat
shows "∀ s. is_kb_1nCR_strategy k s r ⟶ (s ∪ s^-1)^* = (r ∪ r^-1)^*"

lemma lem_prop4:
fixes r s::"'U rel"
shows "is_CR_1step_strategy s r =
                 (is_det_1step_strategy s r ∧ is_kb_1nCR_strategy 0 s r)"

lemma lem_prop5:
fixes r::"'U rel"
assumes "|{x::'U. True}| ≤o |{x::nat. True}|"
shows "confl_rel r = (∃ s. is_kb_1nCR_strategy 0 s r)"
```

Formalization of Theorem 1 (main theorem) in [13, lines 2693–2696] has the following form:

```
theorem thm_1:
fixes r::"'U rel"
assumes "|r| ≤o cardSuc |{x::nat. True}|"
shows "confl_rel r = (∃ s. is_kb_1nCR_strategy 1 s r)"
```

Here the symbols $|\cdot|$ and $\leq_0$, and *cardSuc* denote the cardinality of a set/relation, cardinality comparison, and the cardinal successor operation respectively. They are defined in the theory *HOL-Cardinals* [3].

7 Conclusions

In the paper we have investigated strategies for proving convertibility in confluent ARS of the smallest uncountable cardinality that generalize one-step Church-Rosser strategies for countable confluent ARS. Under the Continuum Hypothesis, the results obtained in the paper (in particular, Theorem 1) may be useful for guiding the design of (semi-)decision procedures for (restricted instances of) word problems for confluent ARS of the cardinality of the continuum that include rewriting systems on many structures of interest to computer science and applied mathematics (e.g. on the set of graphs with a fixed countable set of vertices, the set of real numbers, the set of finite-dimensional real vectors or matrices, etc.).

Investigation of applications of the obtained results to concrete rewriting systems and investigation of strategies for ARS of high uncountable cardinality are subjects of future work.

Appendix

Instructions on re-checking formal proofs in the supplementary material [13] are given below:

1. Install Isabelle 2025 software using instructions at:

   https://isabelle.in.tum.de/website-Isabelle2025/installation.html

2. Run the installed Isabelle 2025 software.
3. Obtain the file `NCRS_N1.thy` (supplementary material for this paper) from:

   https://doi.org/10.5281/zenodo.18079151

   and obtain the file `DCRN1.thy` (dependency of `NCRS_N1.thy`) from:

   https://doi.org/10.5281/zenodo.14254256

4. Place both files `NCRS_N1.thy` and `DCRN1.thy` in the same directory.
5. Choose File → Open from the menu in Isabelle window and select the file `NCRS_N1.thy` (its dependency will be loaded automatically).
6. Scroll to the end of the loaded text in Isabelle window and wait until all red marks on the right scroll bar disappear (this may take several minutes).
7. Now formal proofs have been checked. The main result (theorem `thm_1`) is located at the line 2693 in `NCRS_N1.thy`.

References

1. Isabelle proof assistant. https://isabelle.in.tum.de
2. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge university press (1999)
3. Blanchette, J.C., Popescu, A., Traytel, D.: Cardinals in Isabelle/HOL. Lecture Notes in Computer Science, vol. 8558, pp. 111–127 (2014)
4. Cardone, F., Hindley, J.: Lambda-calculus and combinators in the 20th century. In: Handbook of the History of Logic, Volume 5, pp. 723–817. North-Holland (2009)
5. Endrullis, J., Klop, J.W.: De Bruijn's weak diamond property revisited. Indagationes Mathematicae **24**, 1050–1072 (2013)
6. Endrullis, J., Klop, J.W., Overbeek, R.: Decreasing diagrams for confluence and commutation. Logical Methods in Computer Science **16**, 23:1–23:25 (2020)
7. Evans, T.: Word problems. Bulletin of the American Mathematical Society **84**(5), 789–802 (1978)
8. Fraenkel, A., Bar-Hillel, Y., Levy, A.: Foundations of Set Theory. Elsevier (1973)
9. Hindley, J.: The Church–Rosser Property and a Result in Combinatory Logic. Ph.D. thesis, University of Newcastle-upon-Tyne (1964)
10. Huet, G.: Confluent reductions: Abstract properties and applications to term rewriting systems. Journal of the ACM **27**(4), 797–821 (1980)
11. Ivanov, I.: Formal proof of completeness of the decreasing diagrams method for proving confluence of relations of the least uncountable cardinality. https://doi.org/10.5281/zenodo.14254256 (2024)

12. Ivanov, I.: Completeness of the decreasing diagrams method for proving confluence of rewriting systems of the least uncountable cardinality. Leibniz international proceedings in informatics **337**, 25:1–25:20 (2025), https://doi.org/10.4230/LIPIcs.FSCD.2025.25
13. Ivanov, I.: Supplementary material for this paper. https://doi.org/10.5281/zenodo.18079151 (2025)
14. Klop, J.W.: Combinatory Reduction Systems. Ph.D. thesis, Rijks-universiteit Utrecht (1980)
15. Klop, J.W.: Term rewriting systems. Centrum voor Wiskunde en Informatica (1990)
16. Klop, J., van Oostrom, V., van Raamsdonk, F.: Reduction strategies and acyclicity. Lecture Notes in Computer Science, vol. 4600, pp. 89–112 (2007)
17. Krauss, A., Schropp, A.: A mechanized translation from Higher-Order Logic to set theory. Lecture Notes in Computer Science, vol. 6172, pp. 323–338. Springer (2010)
18. Kunčar, O., Popescu, A.: A consistent foundation for Isabelle/HOL. Journal of Automated Reasoning **62**, 531–555 (2019)
19. Malbos, P.: Lectures on algebraic rewriting. http://hal.science/hal-02461874v1 (2019)
20. Markowsky, G.: Chain-complete posets and directed sets with applications. Algebra universalis **6**(1), 53–68 (1976)
21. Meseguer, J.: Twenty years of rewriting logic. The Journal of Logic and Algebraic Programming **81**, 721–781 (2012)
22. Muller-Stach, S.: Max Dehn, axel Thue and the Undecidable. In: Max Dehn: Polyphonic Portrait, volume 46 of History of Mathematics series (2024)
23. Newman, M.H.A.: On theories with a combinatorial definition of "equivalence". Annals of mathematics pp. 223–243 (1942)
24. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: a proof assistant for higher-order logic. Springer (2002)
25. Olarte, C., Olveczky, P.: Timed strategies for real-time rewrite theories. In: Rewriting Logic and its Applications. 15th International Workshop, WRLA 2024, Luxembourg City, Luxembourg, April 6–7, 2024. pp. 181–201 (2024)
26. van Oostrom, V.: Confluence by decreasing diagrams. Theoretical computer science **126**(2), 259–280 (1994)
27. van Oostrom, V.: Confluence for Abstract and Higher-Order Rewriting. Ph.D. thesis, Vrije Universiteit Amsterdam (1994)
28. Pitts, A.: The HOL logic. In: Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, pp. 191–232. Cambridge University Press (1993)
29. Rosen, B.: Tree manipulating systems and Church-Rosser theorems. Journal of the ACM **20**, 160–187 (1973)
30. Rubio, R., Marti, N., Pita, I., Verdejo, A.: Strategies, model checking and branching-time properties in maude. In: Rewriting Logic and its Applications. 13th International Workshop, WRLA 2020, Dublin, Ireland, April 25th & 26th, 2020. pp. 172–185 (2020)
31. Sagan, H.: Space-Filling Curves. Springer (1994)
32. Terese: Term Rewriting Systems. Cambridge University Press (2003)
33. Toyama, Y.: Reduction strategies for left-linear term rewriting systems. Lecture Notes in Computer Science, vol. 3838, pp. 198–223 (2005)

An Algebraic Specification for Quantum Computation in Maude*

Canh Minh Do and Kazuhiro Ogata

Japan Advanced Institute of Science and Technology (JAIST), Nomi, Japan
{canhdo,ogata}@jaist.ac.jp

Abstract. With the rapid advancement of quantum computing, formal verification of quantum systems has become an increasingly important research topic. At the same time, the algebraic specification community, particularly the Maude community, has started to show interest in quantum computing. However, a dedicated algebraic specification framework for quantum computation is still lacking. This paper introduces |AS4QC>, an algebraic specification for quantum computation in Maude, which provides a formal framework for modeling, symbolic and exact reasoning about, and verifying quantum systems. We employ the ring $\mathbb{D}[\omega]$ to exactly represent complex numbers in quantum computation without approximation, use Dirac's bra-ket notation to model quantum states and quantum operations, and leverage a set of laws from quantum mechanics and basis matrix operations expressed in Dirac notation to automate reasoning about quantum computation. As case studies, we use |AS4QC> to verify the correctness of several nontrivial quantum programs, demonstrating the effectiveness and practicality of our approach.

Keywords: algebraic specification · quantum computation · verification

1 Introduction

Quantum computing has attracted significant attention worldwide, and with substantial investment from both governments and major companies, the development of large-scale quantum computers is increasingly seen as a matter of time and sustained effort. Quantum computing differs fundamentally from classical computing by exploiting principles of quantum mechanics, such as superposition, entanglement, and interference. These properties enable quantum algorithms, such as Shor's algorithm for integer factorization and discrete logarithms [24], and Grover's algorithm for unstructured database search [12], to achieve substantial speedups over their best-known classical counterparts. However, due to the counterintuitive and probabilistic nature of quantum computation, the likelihood of programming errors in quantum programs is much higher than in classical ones. Moreover, traditional testing techniques are much less

* This work was supported by JSPS KAKENHI Grant Numbers JP23K28060, JP23K19959, JP24K20757, JP24KK0185.

effective in the quantum setting because of nondeterminism and probabilistic outcomes inherent in quantum measurement. Consequently, formal verification is expected to play a crucial role in ensuring the reliability of quantum systems before they can be trusted in safety-critical and security-sensitive applications.

The algebraic specification community, particularly the Maude community, has started to show interest in quantum computing. However, a dedicated algebraic specification framework for quantum computation is still lacking, which makes the development of formal verification techniques for quantum systems in Maude challenging. In this paper, we present `|AS4QC>`, an algebraic specification for quantum computation in Maude, which provides a formal framework for modeling, symbolic and exact reasoning about, and verifying quantum systems. To develop `|AS4QC>`, we need to support algebraic representations of complex numbers, quantum states, and quantum operations, as well as automated reasoning about quantum computation. Kliuchnikov et al. conjectured in [16], and Giles and Selinger later proved in [11], that a unitary matrix can be exactly realized by a quantum circuit over the universal Clifford + T gate set if and only if all its matrix entries belong to the ring $\mathbb{D}[\omega]$. Therefore, it is promising to study the ring $\mathbb{D}[\omega]$ and its use for the exact algebraic representation of complex numbers arising in quantum computation. Accordingly, we employ $\mathbb{D}[\omega]$ to algebraically represent complex numbers in quantum computation without approximation. For the algebraic representation of quantum states and quantum operations, we adopt Dirac's bra-ket notation [5] instead of explicitly using complex vectors and matrices as in [20,15,27,25]. Dirac notation is widely used in quantum mechanics because of its succinctness and convenience, and it allows us to obtain compact representations that are well-suited to algebraic manipulation. For automated reasoning, we leverage a set of laws from quantum mechanics and basis matrix operations expressed in Dirac notation to perform quantum computation.

Maude [17] is a high-level specification and programming language based on rewriting logic [18], focusing on simplicity, expressiveness, and performance, and providing a wide range of formal analysis facilities. Therefore, we use it to develop the algebraic specification `|AS4QC>` for quantum computation. The algebraic specification allows us to use symbolic values for complex numbers, thereby enabling symbolic quantum computation, which is a major advantage of our approach. As case studies, we use `|AS4QC>` to verify the correctness of several nontrivial quantum programs, including Entanglement, Quantum Teleportation [1], Entanglement Swapping [28], Quantum Secret Sharing [14], Quantum Relay Scheme [4], Bidirectional Quantum Teleportation [13], Quantum Network Coding [21], and Grover's Search [12], to demonstrate the effectiveness and practicality of our approach. The algebraic specification `|AS4QC>` and case studies are publicly available at https://github.com/canhminhdo/as4qc.

The rest of the paper is organized as follows. Section 2 introduces basic notations from quantum computation. Section 3 presents the algebraic specification for quantum computation. Section 4 describes a case study in `|AS4QC>`. Section 5 shows the experimental results for several case studies. Section 6 mentions related work and finally concludes the paper with some future directions.

2 Preliminaries

This section describes basic notations from quantum computation (refer to [19] for more details) and assumes that the reader has some basic knowledge of linear algebra.

The state space of a quantum system is a Hilbert space $\mathcal{H}$, which is a complete complex vector space equipped with an inner product. A pure state of a quantum system is described by a unit vector written in Dirac's ket notation as $|\psi\rangle \in \mathcal{H}$. The conjugate transpose of $|\psi\rangle$ is a row vector written in Dirac's bra notation as $\langle\psi| \triangleq |\psi\rangle^\dagger$. The inner product of $|\psi\rangle$ and $|\phi\rangle$ is written as $\langle\psi|\phi\rangle$ and they are orthogonal if $\langle\psi|\phi\rangle = 0$. The outer product of them, denoted $|\psi\rangle\langle\phi|$, is a linear operator that maps any $|\psi'\rangle \in \mathcal{H}$ to $\langle\phi|\psi'\rangle\,|\psi\rangle$. The length of $|\psi\rangle$ is defined as $\||\psi\rangle\| \triangleq \sqrt{\langle\psi|\psi\rangle}$. A set of vectors $B \triangleq \{|i\rangle : i \in I\} \in \mathcal{H}$ is orthonormal if each $|i\rangle$ is normalized and every two vectors in the set are orthogonal. Furthermore, if they span the whole space $\mathcal{H}$; that is, any vector in $\mathcal{H}$ can be written as a linear combination of vectors in B, then B is called an orthonormal basis of $\mathcal{H}$.

A qubit is a quantum system whose state space is the two-dimensional Hilbert space $\mathcal{H}_2 = \mathbb{C}^2$. The state of a qubit can be written as $\alpha\,|0\rangle + \beta\,|1\rangle$ with $|\alpha|^2 + |\beta|^2 = 1$, where $\alpha, \beta \in \mathbb{C}$, $|0\rangle \triangleq [1\ 0]^T$, and $|1\rangle \triangleq [0\ 1]^T$ with ^{T} denoting the transpose operator. The coefficients α and β are called the amplitudes of this quantum state. The set $\{|0\rangle, |1\rangle\}$ forms an orthonormal basis of $\mathcal{H}_2$ and is called the computational basis. For multiple qubits, we use tensor products of Hilbert spaces. Let $\mathcal{H}_A$ and $\mathcal{H}_B$ be the Hilbert spaces of systems A and B, respectively. The tensor product of $\mathcal{H}_A$ and $\mathcal{H}_B$ is the Hilbert space $\mathcal{H}_A \otimes \mathcal{H}_B$ (or $\mathcal{H}_{AB}$) of their composite system AB. It consists of linear combinations of product states $|\psi\rangle \otimes |\phi\rangle$ (often written $|\psi\phi\rangle$ or $|\psi\rangle\,|\phi\rangle$) with $|\psi\rangle \in \mathcal{H}_A$ and $|\phi\rangle \in \mathcal{H}_B$. Any state in $\mathcal{H}_A \otimes \mathcal{H}_B$ that cannot be written as a product state is called an entangled state. For example, the Bell (or EPR) state $|\Phi^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$ is maximally entangled.

In quantum computing, we mainly perform two types of operations: unitary transformation (also known as quantum gates) and quantum measurement. A unitary transformation on a quantum system in the Hilbert space $\mathcal{H}$ is a linear operator $U : \mathcal{H} \to \mathcal{H}$ satisfying $UU^\dagger = U^\dagger U = I_{\mathcal{H}}$, where $U^\dagger$ is the conjugate transpose of U and $I_{\mathcal{H}}$ is the identity operator on $\mathcal{H}$. After unitary transformation, a state $|\psi\rangle$ is changed to $U\,|\psi\rangle$ deterministically. We present the definition of some standard unitary transformations as follows.

$$\mathrm{X} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \qquad \mathrm{Y} = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \qquad \mathrm{Z} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix},$$

$$\mathrm{H} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \qquad \mathrm{T} = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}, \qquad \mathrm{CX} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

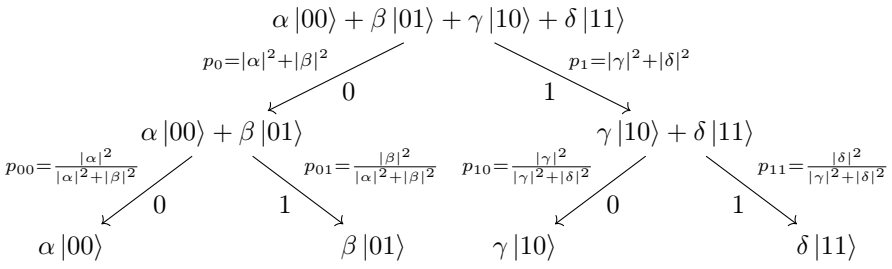
A quantum measurement on a quantum system in the Hilbert space $\mathcal{H}$ is a collection of measurement operators $\{M_m\}$ satisfying the completeness relation

$\sum_m M_m^\dagger M_m = I$. Measuring a state $|\psi\rangle$ yields outcome m with probability $p(m) = \| M_m |\psi\rangle \|^2$, and the state is changed to $|\psi_m\rangle = \frac{M_m|\psi\rangle}{\sqrt{p(m)}}$. Note that $M_m |\psi\rangle$ is normalized by dividing by its norm $\sqrt{p(m)} = ||M_m |\psi\rangle||$, so that it becomes a unit vector. As an example, we consider the measurement $\{M_0 = |0\rangle\langle 0|, M_1 = |1\rangle\langle 1|\}$ applied to the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ with $|\alpha|^2 + |\beta|^2 = 1$. The probabilities of obtaining outcomes 0 and 1 are $|\alpha|^2$ and $|\beta|^2$, respectively and the state collapses to $|0\rangle$ or $|1\rangle$ accordingly. Consequently, quantum measurement changes a quantum system in a nondeterministic and probabilistic way.

The measurement involving multiple qubits is more complex. Let us consider a two-qubit system in a state $|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$ with $|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$. We use the measurement $\{M_0 = |0\rangle\langle 0|, M_1 = |1\rangle\langle 1|\}$, but note that we do not normalize quantum states after each measurement. When we measure the first qubit, one of the following will occur:

- the outcome 0 with probability of $p_0 = |\alpha|^2 + |\beta|^2$, collapsing the quantum state to $\alpha|00\rangle + \beta|01\rangle$, and
- the outcome 1 with probability of $p_1 = |\gamma|^2 + |\delta|^2$, collapsing the quantum state to $\gamma|10\rangle + \delta|11\rangle$.

Next, we measure the second qubit. The possible outcomes are illustrated in the following diagram.



In this diagram, nodes represent quantum states, and transitions are labeled with the probabilities and measurement outcomes. The total probabilities of obtaining 00, 01, 10, and 11 as the result of the two measurements are $p_0 p_{00} = |\alpha|^2$, $p_0 p_{01} = |\beta|^2$, $p_1 p_{10} = |\gamma|^2$, and $p_1 p_{11} = |\delta|^2$. It is common to normalize quantum states to unit vectors after each measurement, ensuring that the sum of the absolute squares of the amplitudes is one. However, the diagram shows that a different normalization is more convenient. This paper adopts the normalization convention from [22,8] such that each state is normalized so that the sum of the absolute squares of the amplitudes equals the total probability of reaching that state. It means that when measuring a state $|\psi\rangle$ with outcome m, the state changes to $|\psi_m\rangle = M_m|\psi\rangle$, which may not be a unit vector since normalization (division by its norm) is omitted. This convention significantly simplifies our representation and computation and does not affect physical observations.

For each closed subspace V of $\mathcal{H}$, there exists a unique projection operator $\mathcal{P}_V$. Every state $|\psi\rangle \in \mathcal{H}$ can be written as $|\psi\rangle = |\psi_V\rangle + |\psi_{V^\perp}\rangle$ with $|\psi_V\rangle \in V$

and $|\psi_{V^\perp}\rangle \in V^\perp$, the orthogonal complement of V. The projection $\mathcal{P}_V : \mathcal{H} \to V$ is defined by $\mathcal{P}_V |\psi\rangle = |\psi_V\rangle$. Since closed subspaces of $\mathcal{H}$ are in one-to-one correspondence with projection operators, we often identify $\mathcal{P}_V$ and V, and write $\mathcal{P}_V$ as $\mathcal{P}$ when the closed subspace V is clear from context. Moreover, if a state $|\psi\rangle$ is in the closed subspace V of the projection operator $\mathcal{P}_V$, then $\mathcal{P}_V |\psi\rangle = |\psi\rangle$.

3 Algebraic Specification for Quantum Computation

This section presents the algebraic specification `|AS4QC>` for quantum computation in Maude, including complex number representation, quantum states and quantum operations, automated reasoning about quantum computation based on a set of laws from quantum mechanics and basic matrix operations, and the usability of `|AS4QC>`. We assume that the reader is familiar with Maude.

3.1 Complex Number Representation

This work focuses on symbolic and exact reasoning for quantum computation. Therefore, it is necessary to exactly represent complex numbers arising in quantum computation without approximation. Kliuchnikov et al. conjectured in [16], and Giles and Selinger later proved in [11], that a unitary matrix can be exactly realized by a quantum circuit over the universal Clifford + T gate set, possibly using one additional ancilla, if and only if its matrix entries belong to the ring $\mathbb{D}[\omega]$. This ring consists of all complex numbers of the form $\frac{1}{\sqrt{2}^k}(a + b\omega + c\omega^2 + d\omega^3)$, where $k \in \mathbb{N}, a, b, c, d \in \mathbb{Z}$, and $\omega = e^{i\pi/4} = \cos(\pi/4) + i\,\sin(\pi/4) = (1+i)/\sqrt{2}$. Using the ring $\mathbb{D}[\omega]$, a complex number is algebraically represented by a quadruple (a, b, c, d) of integers and a normalization factor k of a natural number. Although $\mathbb{D}[\omega]$ does not include all complex numbers, it forms a dense subset of complex numbers such that any complex number can be approximated to arbitrary precision by an element of the ring $\mathbb{D}[\omega]$ [29]. Moreover, the subset of complex numbers is sufficient to describe various standard quantum gates. Our algebraic specification currently supports X, Y, Z, H, S, T, Rx[π/2], Ry[π/2], Rz[π/2], CX, CY, CZ, SWAP, CCX, CCY, CCZ, MCX, MCY, MCZ, MCSWAP, which already includes the universal Clifford + T gate set. This algebraic representation of complex numbers is also sufficient to exactly represent all reachable states of OpenQASM circuits [1] when the initial state is a basis state because complex numbers in the ring $\mathbb{D}[\omega]$ are closed under addition and multiplication.

We employ the ring $\mathbb{D}[\omega]$ to algebraically represent complex numbers and also assume that all amplitudes of a quantum state share a common normalization factor k as in [3,2]. Rather than storing this factor for each amplitude, we store a single global value k for the whole quantum state. The value of k can be inferred from the fact that the sum of probabilities over all basis states is one. Moreover, the initial quantum state is usually a basis state, k is initialized to zero in most

[1] OpenQASM (open quantum assembly language) is an imperative programming language for describing quantum circuits used by IBM.

cases. For each application of an H, Rx[$\pi/2$], Ry[$\pi/2$], or Rz[$\pi/2$] gate, the value of k is incremented since these gates introduce a factor of $1/\sqrt{2}$. It is important to note that this factor is omitted in our representations for these quantum gates, which further simplifies their algebraic representations. After all operations have been applied, the final quantum state is normalized by multiplying it by $(1/\sqrt{2})^k$. Under this representation, each amplitude of a quantum state is represented as a quadruple (a, b, c, d) of integers and a value k is stored globally. Arithmetic operations on complex numbers in this quadruple form, such as addition and multiplication, are therefore significantly simplified.

We have the following lemma to distinguish between zero and nonzero complex numbers in our algebraic specification.

Lemma 1 *Let* $\frac{1}{\sqrt{2}^k}(a + b\omega + c\omega^2 + d\omega^3)$ *be an element of the ring* $\mathbb{D}[\omega]$*, where* $k \in \mathbb{N}$*,* $a, b, c, d \in \mathbb{Z}$*, and* $\omega = e^{i\pi/4}$*. Then, we have*

$$\frac{1}{\sqrt{2}^k}(a + b\omega + c\omega^2 + d\omega^3) = 0 \text{ if and only if } a = b = c = d = 0.$$

Proof. It is straightforward to prove this lemma by substituting $\omega = (1 + i)/\sqrt{2}$*,* $\omega^2 = i$*, and* $\omega^3 = (i - 1)/\sqrt{2}$*.*

Based on the algebraic representation of complex numbers in the ring $\mathbb{D}[\omega]$ and Lemma 1, we formalize complex numbers as quadruples of integers and declare addition, multiplication, and conjugation operations for complex numbers in Maude as follows.

```
sorts ZeroCpx NzCpx Cpx .
subsorts ZeroCpx NzCpx < Cpx .
op (_,_,_,_) : Zero Zero Zero Zero -> ZeroCpx [ctor] .
op (_,_,_,_) : NzInt Int Int Int   -> NzCpx   [ctor] .
op (_,_,_,_) : Int NzInt Int Int   -> NzCpx   [ctor] .
op (_,_,_,_) : Int Int NzInt Int   -> NzCpx   [ctor] .
op (_,_,_,_) : Int Int Int NzInt   -> NzCpx   [ctor] .
op (_,_,_,_) : Int Int Int Int     -> Cpx     [ctor] .
op _+_ : Cpx Cpx     -> Cpx   [comm assoc prec 33] .
op _*_ : Cpx Cpx     -> Cpx   [comm assoc prec 31] .
op _*_ : NzCpx NzCpx -> NzCpx [ditto] .
op _^* : Cpx   -> Cpx   [prec 29] .
op _^* : NzCpx -> NzCpx [prec 29] .
op _^* : NzCpx -> NzCpx [prec 29] .
```

where `ZeroCpx`, `NzCpx`, and `Cpx` denote the sorts of zero complex numbers, nonzero complex numbers, and complex numbers, respectively. The addition, multiplication, and conjugation operations of complex numbers of the form $a + b\omega + c\omega^2 + d\omega^3$ become simple by substituting ω, ω^2, ω^3, $\omega^4 = -1$, $\omega^* = -\omega^3$, $(\omega^2)^* = -\omega^2$, and $(\omega^3)^* = -\omega$ as appropriate and performing the corresponding calculations. This process finally yields concise algebraic results. For example, the conjugate $(a + b\omega + c\omega^2 + d\omega^3)^*$ is $(a - d\omega - c\omega^2 - a\omega^3)$. Accordingly, we define the conjugation operation on complex numbers in Maude as follows.

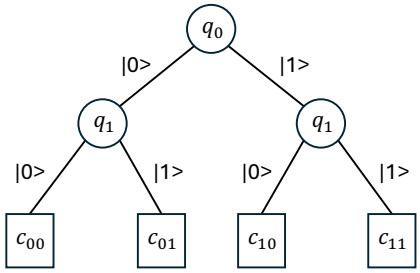


Fig. 1: The binary decision tree of $c_{00}\,|00\rangle + c_{01}\,|01\rangle + c_{10}\,|10\rangle + c_{11}\,|11\rangle$

```
eq (I1, I2, I3, I4)^* = (I1, - I4, - I3, - I2) .
```

where `I1`, `I2`, `I3`, `I4` are variables of sort `Int`. One important operation on complex numbers in quantum computation is computing the squared absolute value of a complex number, which is used to determine the probability of a basis state in a quantum state. We present the following result to facilitate the calculation of the probability of a basis state.

$$\begin{aligned} |a + b\omega + c\omega^2 + d\omega^3|^2 &= (a + b\omega + c\omega^2 + d\omega^3)(a + b\omega + c\omega^2 + d\omega^3)^* \\ &= (a + b\omega + c\omega^2 + d\omega^3)(a - d\omega - c\omega^2 - a\omega^3) \\ &= a^2 + b^2 + c^2 + d^2 + \sqrt{2}(ab + bc + cd - ad). \end{aligned}$$

Note that the value of the probability is a real number when $\sqrt{2}$ appears in the result of $|a + b\omega + c\omega^2 + d\omega^3|^2$.

3.2 Quantum States and Quantum Operations

Dirac's bra-ket notation [5] is widely used in quantum mechanics because of its succinctness and convenience. Therefore, we adopt it to represent quantum states and quantum operations, rather than explicitly using complex vectors and matrices. This idea is inherited from our previous work [6,9]. This notation makes our representation compact and well-suited to algebraic manipulation.

We first formalize basic vectors, covectors, and matrices written in Dirac's bra-ket notation as $|0\rangle$ and $|1\rangle$, $\langle 0|$ and $\langle 1|$, and $|0\rangle\langle 0|$, $|0\rangle\langle 1|$, $|1\rangle\langle 0|$, and $|1\rangle\langle 1|$, respectively. For each of vectors, covectors, and matrices, we formalize constructors of scalar multiplication $\cdot$, addition $+$, multiplication $\times$, tensor product $\otimes$, and conjugate transpose †. Additionally, we formalize the inner product $\langle \_ | \_ \rangle$ and norm square $\| \_ \|^2$ operations for vectors. Terms representing quantum states and quantum operations are then built from scalars (or complex numbers) formalized in Section 3.1, basic elements, and their constructors.

Quantum states over n qubits can be viewed as a binary decision tree over n quantum variables. For example, a two-qubit state $c_{00}\,|00\rangle + c_{01}\,|01\rangle + c_{10}\,|10\rangle + c_{11}\,|11\rangle$ can be represented by a binary decision tree over the quantum variables

q_0 and q_1 as shown in Figure 1. This representation is inspired by [30,3] and the probability amplitude c_{00} of the basis state $|01\rangle$ corresponds to the branch $q_0 \xrightarrow{|0\rangle} q_1 \xrightarrow{|1\rangle} c_{01}$ in the binary decision tree. Although a two-qubit state can be written as a linear combination of basis states, this representation is not efficient for computation. Therefore, we adopt a binary decision tree-based representation, expressing the state as $|0\rangle \otimes (c_{00} \cdot |0\rangle + c_{01} \cdot |1\rangle) + |1\rangle \otimes (c_{10} \cdot |0\rangle + c_{11} \cdot |1\rangle)$. This hierarchical structure simulates a binary decision tree, uniquely represents quantum states, and enables more efficient manipulation. For example, if we measure the first qubit q_0, we only need to consider the term $|0\rangle\otimes(c_{00}\cdot|0\rangle+c_{01}\cdot|1\rangle)$ and simply discard the remainder in the case of a zero outcome; and similarly for the case of a one outcome. Therefore, we use this binary decision tree-based representation for quantum states in our algebraic specification. Note that this representation is also used for the bra representations of quantum states.

Quantum operations are built from scalars and basic matrices using their constructors. We do not impose a normal form representation for quantum operations as we have done for quantum states; instead, we prepare them in a designated representation to speed up computation when applying quantum operations to quantum states. This is possible because users do not directly manipulate quantum gates but interact with them through a high-level interface provided by our algebraic specification. We can definitely make a normal form for quantum operations for certain purposes, such as equivalence checking of quantum circuits; however, doing so may degrade performance, and therefore, we do not adopt a normal form for quantum operations in this work. For intuition, some standard unitary matrices can be represented as follows.

$$
\begin{aligned}
\mathrm{X} &= |0\rangle\langle 1| + |1\rangle\langle 0|\,, & \mathrm{Y} &= (0,0,-1,0)\cdot|0\rangle\langle 1| + (0,0,1,0)\cdot|1\rangle\langle 0|\,,\\
\mathrm{Z} &= |0\rangle\langle 0| + (-1,0,0,0)\cdot|1\rangle\langle 1|\,, & \mathrm{H} &= |0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| + (-1,0,0,0)\cdot|1\rangle\langle 1|\,,\\
\mathrm{CX} &= |0\rangle\langle 0|\otimes \mathrm{I} + |1\rangle\langle 1|\otimes \mathrm{X}, & \mathrm{T} &= |0\rangle\langle 0| + (0,1,0,0)\cdot|1\rangle\langle 1|\,,\\
\mathrm{P0} &= |0\rangle\langle 0|\,, & \mathrm{P1} &= |1\rangle\langle 1|\,,
\end{aligned}
$$

where the factor $1/\sqrt{2}$ is omitted in the Hadamard gate H, as explained in Section 3.1 and the symbol I denotes the identity operator on the Hilbert space $\mathcal{H}_2$. We also present the projection operators P0 and P1 corresponding to the measure operators M_0 and M_1, respectively, as shown in Section 2. The algebraic representation of complex numbers in the ring $\mathbb{D}[\omega]$ again simplifies the representation of these quantum operations, especially in the cases of H and T gates. Note that X, Y, Z, H, and T gates and P0 and P1 projectors act on one qubit, while CX gate acts on two qubits. Let $\bar{q} = q_0, \ldots, q_{n-1}$ be a quantum register, that is, a sequence of distinct quantum variables, used to refer to quantum states $|\psi\rangle$ over n qubits. Let $\bar{r}_i$ be a subsequence of $\bar{q}$. To apply a quantum gate U_i to a quantum state $|\psi\rangle$ with respect to the quantum register $\bar{r}_i$, we first construct the cylindrical extension of U_i in the form $\overline{U}_i = U_i \otimes I_i$ so that the dimension of $\overline{U}_i$ matches that of $|\psi\rangle$. Here, I_i is the identity operator on the Hilbert space $\mathcal{H}_{\bar{q}\setminus\bar{r}_i}$. For example, when applying the Hadamard gate H to the qubit q_0 of a two-qubit state shown in Figure 1, the cylindrical extension of H is $\mathrm{H}\otimes\mathrm{I}$. We use the rep-

resentation of $\mathrm{H} \otimes \mathrm{I}$ in the form $(|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| + (-1,0,0,0) \cdot |1\rangle\langle 1|) \otimes \mathrm{I}$ instead of $|0\rangle\langle 0| \otimes \mathrm{I} + |0\rangle\langle 1| \otimes \mathrm{I} + |1\rangle\langle 0| \otimes \mathrm{I} + (-1,0,0,0) \cdot |1\rangle\langle 1| \otimes \mathrm{I}$, where the tensor product $\otimes$ is distributed over the addition $+$ in the latter representation. We utilize the former representation for quantum operations to make them succinct and improve computational efficiency, which is demonstrated in the next section.

3.3 Automated Reasoning for Quantum Computation

We leverage a set of laws from quantum mechanics and basic matrix operations, as shown in Table 1, to enable automated reasoning for quantum computation in Maude. This set of laws is also inherited from our previous work [6,9]. Because $|0\rangle$ and $|1\rangle$ can be viewed as 2×1 matrices and $\langle 0|$ and $\langle 1|$ can be viewed as 1×2 matrices, these laws effectively describe matrix calculations using Dirac notation, along with zero matrices, identity matrices, and scalars. These laws are formalized as equations in Maude and are used to simplify terms representing quantum states, quantum operations, and the application of quantum operations to quantum states. It is important to note that these laws need to be applied flexibly to achieve the representations of quantum states and quantum operations described in Section 3.2.

For example, we would like to reduce the term $\mathrm{CX} \times ((\mathrm{H} \otimes \mathrm{I}) \times |0\rangle \otimes |0\rangle)$ to check whether its result is $|0\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle$. Note that the factor $1/\sqrt{2}$ in the Hadamard gate is handled by the global value k as explained in Section 3.1. The term says that the H gate acts on the first qubit, followed by the CX gate, where the control and target qubits are the first and second qubits, respectively. The simplification of the term goes as follows:

$\mathrm{H} \times |0\rangle$
$\to (|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| + (-1,0,0,0) \cdot |1\rangle\langle 1|) \times |0\rangle$
$\to |0\rangle\langle 0| \times |0\rangle + |0\rangle\langle 1| \times |0\rangle + |1\rangle\langle 0| \times |0\rangle + ((-1,0,0,0) \cdot |1\rangle\langle 1|) \times |0\rangle$
$\to |0\rangle + |1\rangle$

$(\mathrm{H} \otimes \mathrm{I}) \times (|0\rangle \otimes |0\rangle)$
$\to (\mathrm{H} \times |0\rangle) \otimes (\mathrm{I} \times |0\rangle)$
$\to (|0\rangle + |1\rangle) \otimes |0\rangle$
$\to |0\rangle \otimes |0\rangle + |1\rangle \otimes |0\rangle$

$\mathrm{CX} \times ((\mathrm{H} \otimes \mathrm{I}) \times (|0\rangle \otimes |0\rangle))$
$\to (|0\rangle\langle 0| \otimes \mathrm{I} + |1\rangle\langle 1| \otimes \mathrm{X}) \times (|0\rangle \otimes |0\rangle + |1\rangle \otimes |0\rangle)$
$\to (|0\rangle\langle 0| \otimes \mathrm{I} \times |0\rangle \otimes |0\rangle) + (|0\rangle\langle 0| \otimes \mathrm{I} \times |1\rangle \otimes |0\rangle) + (|1\rangle\langle 1| \otimes \mathrm{X} \times |0\rangle \otimes |0\rangle) + (|1\rangle\langle 1| \otimes \mathrm{X} \times |1\rangle \otimes |0\rangle)$
$\to |0\rangle \otimes (\mathrm{I} \times |0\rangle) + |1\rangle \otimes (\mathrm{X} \times |0\rangle)$
$\to |0\rangle \otimes |0\rangle + |1\rangle \otimes ((|1\rangle\langle 0| + |0\rangle\langle 1|) \times |0\rangle)$
$\to |0\rangle \otimes |0\rangle + |1\rangle \otimes (|1\rangle\langle 0| \times |0\rangle + |0\rangle\langle 1| \times |0\rangle)$
$\to |0\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle$

where multiple rewrite steps are subsumed to avoid verbosity. Using these laws, the application of quantum operations to quantum states can be simplified into new quantum states. The entire process is carried out automatically in Maude with `|AS4QC>`, and the resulting state is the same as expected. The key idea is to reduce the matrix multiplication in the form of $\langle i|j\rangle$ with $i, j \in \{0, 1\}$ into a

Table 1: A set of laws used for automated reasoning about quantum computation

No.	Law
L1	$\langle 0\|0\rangle = \langle 1\|1\rangle = 1, \langle 0\|1\rangle = \langle 1\|0\rangle = 0$
L2	Associativity of $\times, +, \otimes$ and Commutativity of $+$
L3	$0 \cdot \boldsymbol{A}_{m\times n} = \mathbf{O}_{m\times n},\ c \cdot \mathbf{O} = \mathbf{O},\ 1 \cdot \boldsymbol{A} = \boldsymbol{A}$
L4	$c \cdot (\boldsymbol{A} + \boldsymbol{B}) = c \cdot \boldsymbol{A} + c \cdot \boldsymbol{B}$
L5	$c_1 \cdot \boldsymbol{A} + c_2 \cdot \boldsymbol{A} = (c_1 + c_2) \cdot \boldsymbol{A}$
L6	$c_1 \cdot (c_2 \cdot \boldsymbol{A}) = (c_1 \cdot c_2) \cdot \boldsymbol{A}$
L7	$(c_1 \cdot \boldsymbol{A}) \times (c_2 \cdot \boldsymbol{B}) = (c_1 \cdot c_2) \cdot (\boldsymbol{A} \times \boldsymbol{B})$
L8	$\boldsymbol{A} \times (c \cdot \boldsymbol{B}) = (c \cdot \boldsymbol{A}) \times \boldsymbol{B} = c \cdot (\boldsymbol{A} \times \boldsymbol{B})$
L9	$\boldsymbol{A} \otimes (c \cdot \boldsymbol{B}) = (c \cdot \boldsymbol{A}) \otimes \boldsymbol{B} = c \cdot (\boldsymbol{A} \otimes \boldsymbol{B})$
L10	$\mathbf{O}_{m\times n} \times \boldsymbol{A}_{n\times p} = \boldsymbol{A}_{m\times n} \times \mathbf{O}_{n\times p} = \mathbf{O}_{m\times p}$
L11	$\boldsymbol{I}_m \times \boldsymbol{A}_{m\times n} = \boldsymbol{A}_{m\times n} \times \boldsymbol{I}_n = \boldsymbol{A}_{m\times n}$
L12	$\boldsymbol{A} + \mathbf{O} = \mathbf{O} + \boldsymbol{A} = \mathbf{O}$
L13	$\mathbf{O}_{m\times n} \otimes \boldsymbol{A}_{p\times q} = \boldsymbol{A}_{p\times q} \otimes \mathbf{O}_{m\times n} = \mathbf{O}_{mp\times nq}$
L14	$\boldsymbol{A} \times (\boldsymbol{B} + \boldsymbol{C}) = \boldsymbol{A} \times \boldsymbol{B} + \boldsymbol{A} \times \boldsymbol{C}$
L15	$(\boldsymbol{A} + \boldsymbol{B}) \times \boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{C} + \boldsymbol{B} \times \boldsymbol{C}$
L16	$(\boldsymbol{A} \otimes \boldsymbol{B}) \times (\boldsymbol{C} \otimes \boldsymbol{D}) = (\boldsymbol{A} \times \boldsymbol{C}) \otimes (\boldsymbol{B} \times \boldsymbol{D})$
L17	$\boldsymbol{A} \otimes (\boldsymbol{B} + \boldsymbol{C}) = \boldsymbol{A} \otimes \boldsymbol{B} + \boldsymbol{A} \otimes \boldsymbol{C}$
L18	$(\boldsymbol{A} + \boldsymbol{B}) \otimes \boldsymbol{C} = \boldsymbol{A} \otimes \boldsymbol{C} + \boldsymbol{B} \otimes \boldsymbol{C}$
L19	$(c \cdot \boldsymbol{A})^\dagger = c^* \cdot \boldsymbol{A}^\dagger,\ (\boldsymbol{A} \times \boldsymbol{B})^\dagger = \boldsymbol{B}^\dagger \times \boldsymbol{A}^\dagger$
L20	$(\boldsymbol{A} + \boldsymbol{B})^\dagger = \boldsymbol{A}^\dagger + \boldsymbol{B}^\dagger,\ (\boldsymbol{A} \otimes \boldsymbol{B})^\dagger = \boldsymbol{A}^\dagger \otimes \boldsymbol{B}^\dagger$
L21	${\boldsymbol{I}_m}^\dagger = \boldsymbol{I}_m, \boldsymbol{O}^\dagger_{m\times n} = \boldsymbol{O}_{n\times m}, (\boldsymbol{A}^\dagger)^\dagger = \boldsymbol{A}$
L22	$\|0\rangle^\dagger = \langle 0\|,\ \langle 0\|^\dagger = \|0\rangle,\ \|1\rangle^\dagger = \langle 1\|,\ \langle 1\|^\dagger = \|1\rangle$

scalar and simplify the matrix representation by absorbing ones and eliminating zeros as soon as possible (see the law with label L3). For example, given the term $|1\rangle\langle 1| \otimes \mathrm{X} \times |0\rangle \otimes |0\rangle$, we first check that $|1\rangle\langle 1| \times |0\rangle = 0$ and then it is unnecessary to consider $\mathrm{X} \times |0\rangle$ and return the zero vector immediately. This rewrite strategy significantly improves performance when matrices and vectors corresponding to multiple qubits are used instead of X and $|0\rangle$ as in this simple example. Moreover, the representation of quantum states described in Section 3.1 is specifically designed to work with this strategy, thereby avoiding many unnecessary computations. In this way, automated reasoning about quantum computation using Dirac's bra-ket notation can be achieved through rewriting in Maude.

3.4 Usability of |AS4QC>

We present the usability of `|AS4QC>` to perform automated reasoning about quantum computation in Maude. In our algebraic specification, we define the sorts `Vect`, `CoVect`, and `Mat` to represent vectors, covectors, and matrices,

which correspond to quantum states, bra representations of quantum states, and quantum operations, respectively. The scalar multiplication $\cdot$, addition $+$, multiplication $\times$, tensor product $\otimes$, and conjugate transpose † are formalized in Maude as the operators `_._`, `_+_`, `_x_`, `_(x)_`, `_^+`, respectively. The inner product $\langle \_|\_ \rangle$ and norm square $\|\_\|^2$ operations for vectors are formalized as the operators `<_,_>` and `||_||^2`, respectively. Recall the complex number representation in Section 3.1, we formalize it as the operator `(_,_,_,_,_)`. Note that underscores denote operator parameters.

We support the following quantum operations in Maude for reasoning about quantum computation.

```
sort Gate .
subsort Gate < Mat .
--- single-qubit operations
ops P0 P1 X Y Z H S T Rx Ry Rz : Nat -> Gate .
--- two-qubit operations
ops CX CY CZ SWAP : Nat Nat -> Gate .
--- three-qubit operations
ops CCX CCY CCZ CSWAP : Nat Nat Nat -> Gate .
--- multi-qubit operations
ops MCX MCY MCZ : NatList Nat     -> Gate .
op  MCSWAP      : NatList Nat Nat -> Gate .
```

The sort `Gate` is defined as a subsort of `Mat` to represent quantum operations. The parameters of quantum operations are natural numbers denoting the qubits to which they are applied. For multi-qubit operations, a list of natural numbers is used to specify the control qubits, together with additional parameters of natural numbers denoting the target qubits. The application of quantum operations is carried out automatically through simplification, as described in Section 3.3.

We next define a computation in `|AS4QC>` that initially takes a list of quantum gates and a quantum state given as input as follows.

```
sort Comp .
op [_,_]     : GateList Vect         -> Comp .
op [_,_,_,_] : GateList Vect Nat Nat -> Comp .
op [_,_]     : Vect Float            -> Comp .
```

The sort `GateList` represents a list of quantum operations, and the sort `Comp` represents computations. Given a list `GL` of quantum operations of sort `GateList` and a quantum state `V` of sort `Vect`, we first compute the number of qubits and initialize the global normalization factor k to zero using the following equation. Since the number of qubits is frequently used to construct cylindrical extensions of quantum operations, it is computed once at the beginning and reused later.

```
eq [GL, V] = [GL, V, sizeV(V), 0] .
```

The quantum operations in `GL` are applied to `V` in order from left to right until `GL` becomes `nil`, indicating that no further quantum operations remain. At this point, the final output is obtained and can be analyzed. To handle the nondeterministic nature of quantum measurement, we formalize a set of computations

with the sort CompSet and formalize the nondeterministic choices of quantum operations with the operator _U_ as follows.

```
op _U_ : GateList GateList -> GateList .
eq [(GL1 U GL2) GL, V, N, N']
 = [GL1 GL, V, N, N'], [GL2 GL, V, N, N'] .
```

where GL1, GL2, and GL are variables of sort GateList, V is a variable of sort Vect, N and N' are variables of sort Nat, and the operator _,_ is the constructor of the set of computations. The equation states that both choices are handled as two separate computations.

To support analysis of the final output, we provide the following operations.

```
op analyze  : CompSet -> CompSet .
op analyze  : CompSet Vect -> CompSet .
op check    : CompSet Formula -> CompSet .
```

Given a set of final computations, the operator analyze returns a set of computations where each computation now consists of the final quantum state and its reaching probability. If a basis state is provided as a parameter, each computation consists of the final quantum state and the probability of that basis state in the final quantum state. Given a set of final computations and a formula, the operator check returns a set of computations that do not satisfy the formula. The formulas of sort Formula are formalized as follows.

```
sort Formula .
op P : Nat Vect     -> Formula [ctor] .
op P : Nat Nat Vect -> Formula [ctor] .
op _and_ : Formula Formula -> Formula [comm assoc prec 55] .
```

We formalize projection operators of the form $P(q_i, |\psi\rangle)$ and $\mathrm{P}(q_i, q_j, |\psi'\rangle)$ using the operators P(_,_) and P(_,_,_), respectively, to check whether the final quantum state in a computation belongs to a closed subspace spanned by $\{|\psi\rangle\}$ at qubit q_i and by $\{|\psi'\rangle\}$ at qubits q_i and q_j. We also formalize the conjunction of formulas using the operator _and_.

To use the algebraic specification |AS4QC>, we only need to import the functional module QC, after which all facilities for reasoning about quantum computation become readily available.

4 A Case Study: Quantum Teleportation in |AS4QC>

This section presents how to formally specify and verify Quantum Teleportation in |AS4QC> as a case study. Quantum Teleportation [1] is a quantum communication protocol that enables the teleportation of an arbitrary single-qubit state $|\psi\rangle$ from Alice to Bob. The quantum circuit for this protocol is shown in Figure 2. The protocol begins with the initial quantum state $|\psi\rangle_{q_0} \otimes |0\rangle_{q_1} \otimes |0\rangle_{q_2}$. It first applies the H gate to q_1, followed by the CX gate on q_1 and q_2 to create an entangled state shared by Alice and Bob. Alice then applies the CX gate to q_0 and q_1, followed by the H gate on q_0. Alice then measures q_2 and q_1 in sequence

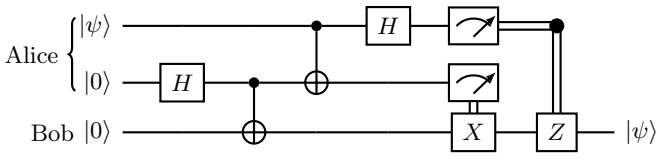


Fig. 2: Quantum Teleportation

and sends the measurement outcomes to Bob. Based on these outcomes, Bob conditionally applies the X and Z gates to recover the state $|\psi\rangle$ on qubit q_2. We want to verify whether $|\psi\rangle$ has been successfully teleported with certainty.

We first specify the functional module `TELEPORT` in `|AS4QC>` for Quantum Teleportation as follows:

```
fmod TELEPORT is
  pr QC .
  ops a b  : -> NzCpx .
  op |psi> : -> Vect  .
  eq |psi> = a . |0> + b . |1> .
  eq  a * a ^* + b * b ^* = (1,0,0,0) .
endfm
```

We define two fresh constants (or symbolic values), `a` and `b`, of sort `NzCpx`, representing nonzero complex numbers. We then define a symbolic quantum state `|psi>` of sort `Vect` as `a.|0> + b.|1>`, where `a` and `b` are subject to the normalization constraint shown in the last equation.

To verify the correctness of Quantum Teleportation, we must consider all possible measurement outcomes that can occur in the protocol. Therefore, we prepare the following command to verify the correctness of the protocol and then ask Maude to reduce it.

```
red analyze([
  H(1) CX(1,2) CX(0, 1) H(0)
  (P0(1) U (P1(1) X(2))) (P0(0) U (P1(0) Z(2))),
  |psi> (x) |0> (x) |0>]) .
```

The list of quantum operations presents the behavior of the protocol where the nondeterministic choices `_U_` and the projection operators `P0` and `P1` for measurement outcomes 0 and 1, respectively, are used to handle the nondeterminism of measurement. The initial quantum state is `|psi>(x)|0>(x)|0>`. Maude completes the command shortly and returns the following result.

```
[|0>(x)|0>(x)(a . |0> + b . |1>), 2.5e-1],
[|1>(x)|0>(x)(a . |0> + b . |1>), 2.5e-1],
[|0>(x)|1>(x)(a . |0> + b . |1>), 2.5e-1],
[|1>(x)|1>(x)(a . |0> + b . |1>), 2.5e-1]
```

The result shows that each computation consists of a final quantum state whose third qubit is `|psi>`, with a reaching probability of 1/4. Consequently, the total

probability of reaching a final quantum state whose third qubit is `|psi>` is 1. We can also use the following command to check the correctness of the protocol.

```
red check([
  H(1) CX(1,2) CX(0, 1) H(0)
  (P0(1) U (P1(1) X(2))) (P0(0) U (P1(0) Z(2))),
  |psi> (x) |0> (x) |0>], P(2,|psi>)) .
```

Maude returns an empty set of computations, meaning that all final computations have a final quantum state whose third qubit is `|psi>`. Therefore, we can conclude the correctness of Quantum Teleportation. Note that we can perform the same analysis when `|psi>` is initialized to either `a.|0>` or `b.|1>` to cover the remaining cases of the quantum state $|\psi\rangle$, and we can obtain the expected result. Using the algebraic specification `|AS4QC>`, we can represent complex numbers symbolically, which allows us to specify arbitrary quantum states for symbolic reasoning. This capability is a major advantage of our approach.

5 Experiments

This section presents the experimental results using `|AS4QC>` to verify the correctness of several case studies. The experiments were conducted using Maude 3.5 on a MacBook Pro machine equipped with an Apple M4 Max chip and 48 GB of RAM. `|AS4QC>` is available at https://github.com/canhminhdo/as4qc.

In the sequel, we present several case studies and their desired properties that we want to verify to demonstrate the effectiveness of our algebraic specification `|AS4QC>` for modeling, reasoning about, and verifying quantum programs.

- Entanglement for constructing an entangled state with multiple qubits.
- Quantum Teleportation [1] for teleporting an arbitrary qubit state $|\psi\rangle$ from a sender to a receiver. Starting from the initial quantum state $|\psi\rangle_{q_0} \otimes |0\rangle_{q_1} \otimes |0\rangle_{q_2}$, we verify whether $|\psi\rangle$ is teleported successfully to qubit q_2 with certainty at the final quantum state.
- Entanglement Swapping [28] for creating a new entangled state $|\Phi^+\rangle$. Given the initial quantum state $|0\rangle_{q_0} \otimes |0\rangle_{q_1} \otimes |0\rangle_{q_2} \otimes |0\rangle_{q_3}$, we verify whether the final joint state of qubits q_0 and q_3 is the EPR state $|\Phi^+\rangle$ at the end.
- Quantum Secret Sharing [14] for teleporting an arbitrary qubit state $|\psi\rangle$ from a sender to a receiver with the help of a third party. From the initial quantum state $|\psi\rangle_{q_0} \otimes |0\rangle_{q_1} \otimes |0\rangle_{q_2} \otimes |0\rangle_{q_3}$, we verify whether $|\psi\rangle$ is teleported successfully to qubit q_3 with certainty at the final quantum state.
- Quantum Relay Scheme [4] for teleporting an arbitrary quantum state $|\psi\rangle$ from one quantum device to another wirelessly, even when these two devices do not share EPR pairs mutually. Starting with the initial quantum state $|\psi\rangle_{q_0} \otimes |0\rangle_{q_1} \otimes |0\rangle_{q_2} \otimes |0\rangle_{q_3} \otimes |0\rangle_{q_4}$, we verify whether $|\psi\rangle$ is teleported successfully to qubit q_4 with certainty at the final quantum state.
- Bidirectional Quantum Teleportation [13] based on EPR pairs and entanglement swapping for simultaneously transmitting arbitrary qubit states $|\psi\rangle$ and $|\phi\rangle$ between two users. From the initial quantum state $|\psi\rangle_{q_0} \otimes |\phi\rangle_{q_1} \otimes$

Table 2: Experimental results with |AS4QC>

Program	Qubits	Unitary	Measurement	Time
Entanglement	100	100	0	53ms
Quantum Teleportation	3	8	2	≈ 0
Entanglement Swapping	4	8	2	≈ 0
Quantum Secret Sharing	4	9	3	≈ 0
Quantum Relay Scheme	5	12	4	1ms
Bidirectional Teleportation	6	12	4	1ms
Quantum Network Coding	14	37	10	268ms
Grover's Search	5	133	0	10ms
	10	1,430	0	1.2s
	15	11,973	0	3.9m

$|0\rangle_{q_2} \otimes |0\rangle_{q_3} \otimes |0\rangle_{q_4} \otimes |0\rangle_{q_5}$, we verify whether $|\psi\rangle$ and $|\phi\rangle$ are simultaneously teleported in qubits q_3 and q_4 with certainty at the final quantum state.

– Quantum Network Coding [21] for the simultaneous generation of EPR pairs using only local operations and classical communication with EPR pairs shared between adjacent quantum repeaters. Given the initial quantum state $|0\rangle_{q_0} \otimes \cdots \otimes |0\rangle_{q_{13}}$, we verify whether the final quantum state contains EPR pairs $|\Phi^+\rangle$ between qubits (q_1, q_4) and (q_0, q_5) at the end.
– Grover's search algorithm [12] for searching an unstructured database with a quadratic speedup by iteratively applying the Grover operator. We verify whether, after approximately $\pi\sqrt{2^n}/4$ iterations of the Grover operator, the correct solution is obtained with high probability, where n and 2^n denote the number of qubits and the database size, respectively.

We successfully verified the correctness of all case studies against their desired properties using |AS4QC>. The experimental results are summarized in Table 2. The columns **Qubits**, **Unitary**, **Measurement** denote the numbers of qubits, unitary operations, and measurements, respectively, while the column **Time** reports the time taken by Maude to verify each program against its specified property. For the Grover's Search case study, measurements are not required because we can directly query the amplitude of the correct solution in the final quantum state supported by |AS4QC>, from which the probability of reaching the correct solution can be computed. Consequently, the number of measurements in Grover's Search is zero. For all case studies except Grover's Search with 15 qubits, we could complete the verification tasks quickly. Especially, we could construct an entangled state with 100 qubits in just 53 milliseconds, comparable with graph-based quantum simulators such as QuIDDPro [26] and MQT Core

DD [30], while array-based quantum simulators such as LIQ$Ui|\rangle$ [27], QX [15], and ProjectQ [25] could not do so as reported in [30]. In the case of Grover's Search with 15 qubits, the verification involves a nontrivial workload with 11,973 applications of quantum operations. As a result, Maude took approximately 3.9 minutes to complete the verification. Notably, |AS4QC> is capable of handling not only concrete and exact values, as presented in Entanglement, Entanglement Swapping, Quantum Network Coding, and Grover's Search, but also symbolic values, as demonstrated in the other case studies. These experimental results demonstrate the effectiveness and practicality of |AS4QC> for modeling, symbolic and exact reasoning about, and verifying quantum programs.

6 Related Work and Conclusion

Symbolic approach to reasoning about quantum computation has been used for theorem proving of quantum circuits in Coq [23] and for reachability analysis, model checking, and equivalence checking of quantum circuits in Maude [8,10,9,7]. However, these approaches can handle simple case studies and a limited number of qubits, mainly due to difficulties in representing complex numbers symbolically and quantum states effectively. Graph-based (or decision diagram-based) approaches [26,30] significantly outperform array-based approaches [27,15,25] for quantum simulation by compactly representing quantum states, exploiting structural redundancies, and providing efficient algorithms for quantum operations. The use of the ring $\mathbb{D}[\omega]$ in decision diagrams to enable symbolic and exact quantum computation was preliminarily investigated in [29] and has since been adopted for the formal verification of quantum circuits and quantum programs in [3,2]. To the best of our knowledge, there is no existing algebraic specification that uses the ring $\mathbb{D}[\omega]$ for symbolic reasoning about quantum computation. Meanwhile, algebraic approaches are expressive and powerful for symbolic reasoning. Thus, |AS4QC> provides a promising algebraic framework for symbolic and exact reasoning about quantum computation in Maude, upon which verification techniques for quantum systems in Maude can be developed in the future.

We have presented |AS4QC>, an algebraic specification for quantum computation in Maude. We have employed the ring $\mathbb{D}[\omega]$ to represent complex numbers in quantum computation without approximation, used Dirac notation to model quantum states and quantum operations, and leveraged a set of laws from quantum mechanics and basis matrix operations to automate reasoning about quantum computation. As case studies, we have used |AS4QC> to confirm the correctness of several nontrivial quantum programs, demonstrating the effectiveness and practicality of our approach. Although a complex number can be algebraically represented by a quadruple (a, b, c, d) and a normalization factor k, many different choices of (a, b, c, d) and k may represent the same complex number. To use model checking techniques effectively, it is therefore necessary to identify a unique representation for this algebraic form. Developing such a unique representation, as suggested in [29], would be one part of our future work, in addition to conducting more complex case studies, such as Shor's algorithm.

References

1. Bennett, C.H., Brassard, G., Crépeau, C., Jozsa, R., Peres, A., Wootters, W.K.: Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. Phys. Rev. Lett. **70**, 1895–1899 (1993). https://doi.org/10.1103/PhysRevLett.70.1895
2. Chen, Y.F., Chung, K.M., Hsieh, M.H., Huang, W.J., Lengál, O., Lin, J.A., Tsai, W.L.: AutoQ 2.0: From verification of quantum circuits to verification of quantum programs. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 87–108. Springer Nature Switzerland (2025). https://doi.org/10.1007/978-3-031-90660-2_5
3. Chen, Y.F., Chung, K.M., Lengál, O., Lin, J.A., Tsai, W.L.: AutoQ: An automata-based quantum circuit verifier. In: Computer Aided Verification. pp. 139–153. Springer Nature Switzerland (2023). https://doi.org/10.1007/978-3-031-37709-9_7
4. Cheng, S.T., Wang, C.Y., Tao, M.H.: Quantum communication for wireless wide-area networks. IEEE Journal on Selected Areas in Communications **23**(7), 1424–1432 (2005). https://doi.org/10.1109/JSAC.2005.851157
5. Dirac, P.A.M.: A new notation for quantum mechanics. Mathematical Proceedings of the Cambridge Philosophical Society **35**(3), 416–418 (1939). https://doi.org/10.1017/S0305004100021162
6. Do, C.M., Ogata, K.: Symbolic model checking quantum circuits in Maude. In: The 35th International Conference on Software Engineering and Knowledge Engineering, SEKE 2023. pp. 103–108 (2023). https://doi.org/10.18293/SEKE2023-014
7. Do, C.M., Ogata, K.: Equivalence checking of quantum circuits based on Dirac notation in Maude. In: Rewriting Logic and Its Applications. pp. 84–103. Springer (2024). https://doi.org/10.1007/978-3-031-65941-6_5
8. Do, C.M., Ogata, K.: An executable operational semantics of quantum programs and its application. In: Software Fault Prevention, Verification, and Validation. pp. 15–31. Springer Nature Singapore (2024). https://doi.org/0.1007/978-981-96-1621-3_2
9. Do, C.M., Ogata, K.: Symbolic model checking quantum circuits in Maude. PeerJ Comput. Sci. **10**, e2098 (2024). https://doi.org/10.7717/PEERJ-CS.2098
10. Do, C.M., Takagi, T., Ogata, K.: Automated quantum protocol verification based on Concurrent Dynamic Quantum Logic. ACM Trans. Softw. Eng. Methodol. **34**(6) (Jul 2025). https://doi.org/10.1145/3708475
11. Giles, B., Selinger, P.: Exact synthesis of multiqubit Clifford+T circuits. Phys. Rev. A **87**, 032332 (Mar 2013). https://doi.org/10.1103/PhysRevA.87.032332
12. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing. p. 212–219. STOC '96, Association for Computing Machinery (1996). https://doi.org/10.1145/237814.237866
13. Hassanpour, S., Houshmand, M.: Bidirectional quantum teleportation and secure direct communication via entanglement swapping (2014). https://doi.org/10.48550/arXiv.1411.0206
14. Hillery, M., Bužek, V., Berthiaume, A.: Quantum secret sharing. Phys. Rev. A **59**, 1829–1834 (1999). https://doi.org/10.1103/PhysRevA.59.1829
15. Khammassi, N., Ashraf, I., Fu, X., Almudever, C., Bertels, K.: QX: A high-performance quantum computer simulation platform. In: Design, Automation &

Test in Europe Conference & Exhibition (DATE). pp. 464–469 (2017). https://doi.org/10.23919/DATE.2017.7927034
16. Kliuchnikov, V., Maslov, D., Mosca, M.: Fast and efficient exact synthesis of single-qubit unitaries generated by Clifford and T gates. Quantum Info. Comput. **13**(7–8), 607–630 (Jul 2013). https://doi.org/10.26421/QIC13.7-8-4
17. M. Clavel, et al.: All About Maude – A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, vol. 4350. Springer (2007). https://doi.org/10.1007/978-3-540-71999-1
18. Meseguer, J.: Twenty Years of Rewriting Logic. J. Log. Algebraic Methods Program. **81**(7-8), 721–781 (2012). https://doi.org/10.1016/j.jlap.2012.06.003
19. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press (2010). https://doi.org/10.1017/CBO9780511976667
20. Paykin, J., Rand, R., Zdancewic, S.: QWIRE: A core language for quantum circuits. SIGPLAN Not. **52**(1), 846–858 (jan 2017). https://doi.org/10.1145/3093333.3009894
21. Satoh, T., Gall, F.L., Imai, H.: Quantum network coding for quantum repeaters. Physical Review A **86**(3) (sep 2012). https://doi.org/10.1103/physreva.86.032331
22. Selinger, P.: Towards a quantum programming language. Mathematical. Structures in Comp. Sci. **14**(4), 527–586 (aug 2004). https://doi.org/10.1017/S0960129504004256
23. Shi, W., Cao, Q., Deng, Y., Jiang, H., Feng, Y.: Symbolic reasoning about quantum circuits in Coq. J. Comput. Sci. Technol. **36**(6), 1291–1306 (2021). https://doi.org/10.1007/s11390-021-1637-9
24. Shor, P.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science. pp. 124–134 (1994). https://doi.org/10.1109/SFCS.1994.365700
25. Steiger, D.S., Häner, T., Troyer, M.: ProjectQ: an open source software framework for quantum computing. Quantum **2**, 49 (Jan 2018). https://doi.org/10.22331/q-2018-01-31-49
26. Viamontes, G., Markov, I., Hayes, J.: High-performance QuIDD-based simulation of quantum circuits. In: Proceedings Design, Automation and Test in Europe Conference and Exhibition. vol. 2, pp. 1354–1355 Vol.2 (2004). https://doi.org/10.1109/DATE.2004.1269084
27. Wecker, D., Svore, K.M.: LIQUi|>: A software design architecture and domain-specific language for quantum computing (2014). https://doi.org/10.48550/arXiv.1402.4467
28. Żukowski, M., Zeilinger, A., Horne, M.A., Ekert, A.K.: "Event-ready-detectors" Bell experiment via entanglement swapping. Phys. Rev. Lett. **71**, 4287–4290 (1993). https://doi.org/10.1103/PhysRevLett.71.4287
29. Zulehner, A., Niemann, P., Drechsler, R., Wille, R.: Accuracy and compactness in decision diagrams for quantum computation. In: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 280–283 (2019). https://doi.org/10.23919/DATE.2019.8715040
30. Zulehner, A., Wille, R.: Advanced simulation of quantum computations. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **38**(5), 848–859 (2019). https://doi.org/10.1109/TCAD.2018.2834427

The Timed *P* Transformation for Distributed Real-Time Systems

José Meseguer[1] and Peter Csaba Ölveczky[2]

[1] University of Illinois, Urbana-Champaign, USA
[2] University of Oslo, Oslo, Norway

Abstract. Both *qualitative* and *quantitative* properties are crucially important in most distributed systems. Analyzing these two kinds of properties often requires using two different system models, necessitating substantial efforts in developing and evolving two quite different models and raising the question of their *semantic consistency*. For *untimed* actor models of distributed systems this problem has been solved in [17] by the P *transformation*, which maps a rewrite theory $\mathcal{R}$ modeling an actor system to a *probabilistic* real-time rewrite theory $\mathcal{R}_\Pi$ semantically consistent with $\mathcal{R}$ and amenable to quantitative analysis by statistical model checking. This paper extends the P transformation to distributed *real-time* systems communicating by asynchronous message passing. We define a *timed* P transformation whose input theories are real-time rewrite theories that extend timed actor systems. We prove that the timed P transformation enjoys desired semantic properties, including amenability to statistical model-checking analysis of the transformed theory.

1 Introduction

Distributed real-time systems communicating by message passing are intrinsically nondeterministic because, besides being concurrent, the delays involved in message transmission are unpredictable. For most of these systems, both *qualitative* properties, such as safety properties, and *quantitative* properties, such as latency and throughput, are crucially important. Analyzing both qualitative and quantitative properties is challenging because the models used for each kind of analysis are often different: a concurrent model will typically be used for verifying qualitative properties by model checking or theorem proving, whereas a probabilistic model such as a stochastic timed automaton may be used to analyze quantitative properties by statistical model checking (see, e.g., [11]). This leads to a *semantic gap* between the different models developed for analyzing different properties, since there may be no clear semantic relation between them.

For *untimed* distributed systems communicating by message passing in a (generalized) version of the actor model [2] the gap has been bridged in [17] by the so-called P transformation. The key idea is to quantify by means of continuous probability distributions Π the nondeterminism inherent in message transmission by the underlying network. Since distributed systems can be naturally modeled as rewrite theories [19], which are supported by the Maude language [10], the

inputs to the P transformation are rewrite theories $\mathcal{R}$ formally specifying such distributed systems, and P is a theory transformation $P : \mathcal{R} \mapsto \mathcal{R}_{\Pi}$ mapping $\mathcal{R}$ to a *probabilistic* real-time rewrite theory $\mathcal{R}_{\Pi}$.

The semantic gap between models for qualitative and quantitative analysis is bridged by the P transformation because: (i) all qualitative analysis can be performed on $\mathcal{R}$ itself; (ii) all quantitative analysis, typically by statistical model checking [3,17], is carried out on $\mathcal{R}_{\Pi}$, which is a refinement of $\mathcal{R}$ that quantifies by means of Π the nondeterminism inherent in $\mathcal{R}$ and, since it just restricts $\mathcal{R}$'s nondeterministic behaviors, is *semantically consistent* with $\mathcal{R}$ itself [17].

This paper extends the P transformation in [17] to a *timed* P transformation whose inputs are real-time rewrite theories specifying distributed *real-time* systems that significantly extend timed actor systems. This makes the class of input theories very general, to support a wide range of applications.

We provide some prerequisites in §2, and describe in §3 the class of real-time rewrite theories that are the inputs to the timed P transformation, which is defined in §4. In §5 we provide two key semantic properties of the timed P transformation ensuring that for any rewrite theory $\mathcal{R}$ and initial state meeting the input requirements: (i) when $\mathcal{R}_{\Pi}$ is viewed as a nondeterministic system, its behaviors from the given initial state are semantically consistent with those in $\mathcal{R}$; and (ii) $\mathcal{R}_{\Pi}$ is *purely probabilistic* and therefore analyzable by standard statistical model-checking algorithms. By bridging the semantic gap between quantitative and qualitative system analysis we obtain a *unified formal methodology*, outlined in §6, for designing distributed real-time systems in rewriting logic and analyzing them with respect to both their qualitative and quantitative properties. §7 discusses related work, and §8 gives some concluding remarks. We refer to our longer report [21] for more detail, explanations, proofs, and three examples illustrating our methodology.

2 Prerequisites

2.1 Rewriting Logic, Maude, and Real-Time Rewrite Theories

Rewriting Logic [19,20] is a logic well suited to formally specify and program concurrent systems. A *rewrite theory* is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$, where:

- Σ is an algebraic *signature*, i.e., a set of *sorts*, *subsorts*, and *function symbols*.
- (Σ, E) is an *order-sorted equational logic theory* [13] specifying the system's data types, with E a set of (possibly conditional) equations and axioms.
- L is a set of rule *labels*.
- R is a collection of *labeled conditional rewrite rules* $[l] : t \to t'$ **if** *cond*, with t, t' Σ-terms and $l \in L$, that specify the system's local transitions.

Maude [10] is a formal specification language and analysis tool for distributed systems. A Maude module specifies a rewrite theory $\mathcal{R}$ as a *system module* `mod` $\mathcal{R}$ `endm`. We summarize Maude's syntax and refer to [10] for details. Operators

are declared `op` f `:` $s_1 \ldots s_n$ `->` s and can have user-definable syntax, with '`_`' denoting argument positions, as in `_+_`. (Unconditional and conditional) equations and rewrite rules are introduced with, resp., the keywords `eq` and `ceq`, and `rl` and `crl`. Mathematical variables are declared with the keywords `var` and `vars`.

Maude provides *syntactic sugar* to support object-oriented specification: A declaration `class` C `|` att_1 `:` s_1`, ..,` att_n `:` s_n declares an *object class* C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C is a term `<` o `:` C `|` $att_1 : val_1, \ldots, att_n : val_n$ `>`, where o (of sort `Oid`) is the object's *identifier*, and val_1 to val_n are the current values of the attributes att_1 to att_n. *Messages* are terms of sort `Msg`. A state of an object-oriented system often has the sort `Configuration` and represents a multiset of objects and messages.

Real-Time Rewrite Theories. Real-time systems can be modeled in rewriting logic as real-time rewrite theories [24], which can be analyzed in Maude [7]. Such theories contain a data type `Time` of (discrete or continuous) time values. Their rules are ordinary rewrite rules, which model *instantaneous change*, and *tick* rewrite rules `[`*tick*`]` `:` $\{t\} \rightarrow \{t'\}$ `in time` τ *if cond* which model *time advance*: the system evolves from state $\{t\}$ to state $\{t'\}$ in time τ. The states have the form $\{u\}$, so that time advances uniformly in all parts of the state.

2.2 Nondeterministic Rules and Probabilistic Rewrite Theories

The variables in v and *cond* in a rewrite rule $[l] : u \rightarrow v$ *if cond* are usually a *subset* of the variables $\boldsymbol{x}$ in u. However, it is possible to model transitions that exhibit *nondeterministic choice* using rewrite rules of the form $[l] : u(\boldsymbol{x}) \rightarrow v(\boldsymbol{x}, \boldsymbol{y})$ *if* $cond(\boldsymbol{x})$. For each matching substitution ρ of the variables $\boldsymbol{x}$ in $u(\boldsymbol{x})$ that satisfies the condition $cond(\boldsymbol{x})\rho$, the resulting state is not a single state $v(\boldsymbol{x})\rho$, but a possibly infinite family of states $\{v(\boldsymbol{x}, \boldsymbol{y})\rho \uplus \tau\}_\tau$, indexed by the choice of substitution τ instantiating the extra variables $\boldsymbol{y}$. For example, the rewrite rule $[choose2] : \langle n, m\rangle \rightarrow \langle n, k\rangle$ transforms a pair $\langle n, m\rangle$ of natural numbers into a pair $\langle n, k\rangle$ where k can be *any* natural number. We might choose k according to a *probability distribution* on the natural numbers. This leads to the notion of a *probabilistic rewrite rule*, which is a rewrite rule of the form

$$[l] : u(\boldsymbol{x}) \rightarrow v(\boldsymbol{x}, \boldsymbol{y}) \text{ if } cond(\boldsymbol{x}) \text{ with probability } \boldsymbol{y} := \pi(\boldsymbol{x}).$$

For each matching substitution ρ of the variables $\boldsymbol{x}$ in $u(\boldsymbol{x})$ that satisfies the condition $cond(\boldsymbol{x})\rho$, the substitution τ instantiating the extra variables $\boldsymbol{y}$ is chosen according to a probability distribution $\pi(\boldsymbol{x})\rho$ on the product of data types of the variables $\boldsymbol{y}$, where the probability distribution $\pi(\boldsymbol{x})\rho$ may be parametric on the choice of values ρ instantiating the variables $\boldsymbol{x}$. A *probabilistic rewrite theory* is a rewrite theory where *some* of its rewrite rules are probabilistic.

A nondeterministic rewrite rule $[l] : u(\boldsymbol{x}) \rightarrow v(\boldsymbol{x}, \boldsymbol{y})$ *if* $cond(\boldsymbol{x})$ is *nonexecutable* (`[nonexec]` in Maude) because a matching substitution ρ of the variables $\boldsymbol{x}$ in u does *not* determine the choice of an extra substitution τ for the extra

variables $\boldsymbol{y}$ in v. By transforming such a nondeterministic rule into a probabilistic rule $u(\boldsymbol{x}) \to v(\boldsymbol{x}, \boldsymbol{y})$ *if* $cond(\boldsymbol{x})$ *with probability* $\boldsymbol{y} := \pi(\boldsymbol{x})$ the rule becomes executable by *simulation* based on *sampling* the probability distribution $\pi(\boldsymbol{x})\rho$ for each matching substitution ρ of the rule's left-hand side.

Even without a probability distribution $\pi(\boldsymbol{x})$, it is still possible to execute nondeterministic rules, which can be useful for *debugging* purposes. Let the nondeterministic variables $\boldsymbol{y} = y_1, \ldots, y_k$ of rule l range over respective sorts $s_1, \ldots, s_k$. Assume that we have added to our theory corresponding sorts $Set[s_1], \ldots, Set[s_k]$ of finite sets of elements in $s_1, \ldots, s_k$, where set union is an associative-commutative binary operation `_,_` with `empty` as identity element. Assume also that we have specified *finite* sets $A_1, \ldots, A_k$ in $Set[s_1], \ldots, Set[s_k]$. Then, our nondeterministic rule l becomes executable in Maude as the rule

$$[l] : u(\boldsymbol{x}) \to v(\boldsymbol{x}, \boldsymbol{y}) \text{ if } cond(\boldsymbol{x}) \wedge y_1 \text{ , } X_1 := A_1 \wedge \ldots \wedge y_k \text{ , } X_k := A_k$$

by adding the *matching conditions* [10] y_i **,** $X_i := A_i$, $1 \leq i \leq k$, where the variables $X_1, \ldots, X_k$ range over respective sorts $Set[s_1], \ldots, Set[s_k]$. Only *some* behaviors of the nondeterministic rule can be tested this way, although if $s_1, \ldots, s_k$ are all *finite* data types, then *all* behaviors can be tested in this way.

3 The Input Real-Time Rewrite Theories

To support a wide range of applications, we specify a quite general class of real-time rewrite theories—extending timed actor systems—as inputs to the timed P transformation. We describe their basic features in §3.1 and explain additional features in §3.2. Further semantic requirements are discussed in §3.3.

3.1 Object-Oriented Real-Time Rewrite Theories

We first explain the basic features of the real-time rewrite theories that are inputs to the timed P transformation. We use *object-oriented* real-time rewrite theories, with local clocks and timers, and whose states include the current configuration (a multiset) of objects, messages, and delayed messages *en route*.

The time domain `Time` denotes the nonnegative real numbers $\mathbb{R}_{\geq 0}$, and its supersort `TimeInf` adds an infinity time value ∞ (written `inf` in Maude).

An object may have *multiple timers*. Timers are used to trigger actions: some rule must be applied when a timer expires, i.e., it becomes zero. That rule should either reset the timer to a non-zero natural number, or turn the timer off (i.e., setting it to the infinity value `inf`) while possibly resetting another timer.

The set of timers of an object could change dynamically. For example, in a formalization of the domain name system (DNS), the resolver may have a timer for each user query, e.g., to periodically resubmit an unanswered query to a name server. We define the following datatype `Timers` for sets of timers:

```
sort Timer Timers TimerData .   subsort Timer < Timers .
op [_;_;_] :  TimeInf NzNat TimerData -> Timer [ctor] .
op noTimer : -> Timers [ctor] .
op __ : Timers Timers -> Timers [ctor assoc comm id: noTimer] .
```

The state of a timer is modeled as a term [*timeToExpiration* ; *period* ; *data*], where *timeToExpiration* denotes the expiration time of the timer, *period* denotes the period of the timer, and *data* is the data associated to the timer.

We assume that all the timers of an object are contained in an attribute `timers` of sort `Timers`. An object need not have such an attribute `timers`. An object *may* also have an attribute `clock` of sort `Time`, which models its local clock that is assumed to be perfect. We assume that no object has other time-dependent attributes besides `clock` and/or `timers`.

The timed behavior of the system is given by the following tick rewrite rule:[3]

```
sort DlyMsg .   subsorts Msg < DlyMsg < Configuration .
op dly : Msg Time -> DlyMsg [ctor] .

[tick] : {STATE} => {timeEffect(STATE, T)} in time T
                    if T := mte(STATE) /\ T > 0 [/\ cond] .
```

The function `mte` defines how much time may elapse in a state before a timer expires or a message arrives. Time cannot advance in a state with sending tasks (see below) present, which means that the rule `sendDelayed` in §3.2 must be applied before time advances. The function `timeEffect` decreases the values of all active timers and all message delays, and increases the local clocks, according to the elapsed time. See [21] for the definition of these functions.

3.2 Admissible Rewrite Rules in the Input Rewrite Theories

The admissible instantaneous rewrite rules of an input real-time rewrite theory $\mathcal{R}$ to the timed P transformation are described in this section, and satisfy additional requirements described in §3.3. These requirements on $\mathcal{R}$ should ensure that the transformed probabilistic real-time rewrite theory $P(\mathcal{R}, \Pi, init)$ satisfies the *absence of nondeterminism* (AND) property for all states reachable from *init*.

An important novelty, not present in the original P transformation in [17], is that the input theory $\mathcal{R}$ may include several kinds of *nondeterministic rewrite rules* detailed in §3.2.1–3.2.3. This makes the input theories quite general. The reason for the generality afforded by nondeterministic rules is that, as suggested in §2.2, by adding a probability distribution, any nondeterministic rewrite rule becomes a probabilistic one, modeling some random phenomenon.

A main requirement of an input rewrite theory is that only the expiration of a timer or the arrival of a message can trigger an instantaneous rewrite rule.

A key invariant we want to maintain is that, after applying the timed P transformation to an input real-time rewrite theory, in any reachable state of the resulting probabilistic rewrite theory (described in §4), the expiration times of all timers and the arrival times of all messages are all different. To achieve this, an application of a rewrite rule can (re)set at most one timer, to a natural number value, and may turn off other timers. Allowing timers to be (re)set to real numbers could break this invariant, as the following example shows:

[3] We do not show variable declarations, but follow the convention that variables are written in all capital letters.

Example 1. When object A receives a *start* message (and it has a local clock), it records the time when the *start* message arrives and starts a timer. A can later send this value as a message to object B, which stores this value. When it is time to reset B's timer, it can use this stored "offset of A" and set its timer to this offset plus some natural number. Now, the timers of A and B have the same offset, which could lead to them expiring at the same time.

For example, if A receives a message at time 1.23456, it could then set its timer to 100 and send a message to B saying that its timer will expire at time 101.23456. When B receives this message, it can set its own timer to also expire at time 101.23456, which leads to undesirable nondeterminism. □

For the sake of a simpler exposition we exclude rules that create new objects dynamically. However, there should be no problems with generating new objects (with or without timers) as long as at most one timer is (re)set in any rule.

We now describe the admissible rewrite rules in real-time rewrite theories that are inputs to the timed P transformation. As in [17], we assume non-nested objects, i.e., objects that do not contain other objects in some of their attributes.

Besides the *tick* rule in §3.1 we have four types of instantaneous rewrite rules:

1. *Deterministic message-triggered rules* of the form:

   ```
   [l] : (to o [from o'] : mp) < o : Cl | atts >
     =>                        < o : Cl | atts' >  [msgs]  [if cond] .
   ```

   or of the form:

   ```
   [l] : (to o [from o'] : mp) < o : Cl | atts >
     =>                        < o : Cl | atts' >     [if cond] .
   ```

   if no messages are sent. *If* the object o has a `timers` attribute, the rule can (re)set at most one of its timers, to a natural number if the rule does not generate messages and to a non-zero natural number otherwise, and some timers may by turned off (set to `inf`) or removed. The rule may also add new timers, as long as at most one timer (existing or new) is (re)set.
   In the term `[`*msgs*`]` of sort `STask` (for *sending task*) appearing in some of the rules, *msgs* is a term of sort `MsgList` of lists of messages:

   ```
   sorts MsgList STask .  subsort Msg < MsgList .
   subsort STask < Configuration .
   op nil : -> MsgList [ctor] .
   op _;_ : MsgList MsgList -> MsgList [ctor assoc id: nil] .
   op [_] : MsgList -> STask [ctor] .
   ```

2. *Deterministic timer-triggered* rules of the form:

   ```
   [l] : < o : Cl | timers : [0 ; p ; td] otherTimers , atts >
     => < o : Cl | timers : [v ; p' ; td] otherTimers' , atts' > [msgs] [if cond] .
   ```

   or of the same form without '`[`*msgs*`]`' if no messages are sent. The new (relative) expiration time v of timer *td* is either a non-zero natural number (typically p) or `inf`. In the latter case, one other timer in *otherTimers may* be reset to a non-zero natural number.

3. *Nondeterministic network message transmission* rules model the message-sending behavior of a network using the following instantaneous rules:

```
[sendDelayed] : [M ; ML] => dly(M, T)  [ML]  [nonexec] .
[sendNil] : [nil] => none .
```

   These rules evaluate a sending task to a corresponding multiset of $n \geq 0$ of *delayed* messages, where the delay of each message is nondeterministically chosen in the nondeterministic rule `sendDelayed`.
4. *Other nondeterministic* rewrite rules discussed in §3.2.1–3.2.3.

3.2.1 Modeling Message Arrival and Message Loss. If the underlying network guarantees that no messages are lost, message arrival is modeled by the rule `[msgArrival] : dly(M,0) => M .` If the network is lossy, message arrival can be modeled by the following *nondeterministic* rewrite rule:

```
[msgArrival] : dly(M,0)  => if B then M else none fi [nonexec] .
```

where `B` is a variable of sort `Bool`, and `none` denotes the empty configuration.

The above rule is *context-free*, which makes it hard to model that the *state* of the network—for example, its congestion—may affect message loss. We can make the `msgArrival` rule *context-sensitive*, to later be able to quantify message loss depending on context, by involving the entire state of objects and messages:

```
[msgArrival] : {dly(M,0) C} => if B then {M C} else {C} fi [nonexec] .
```

where `C` is a variable of sort `Configuration` that matches *the rest of the configuration*, i.e., the *context* in which message `M` is ready to arrive. As explained in §2.2 and §4, the probability distribution associated to a rule in a probabilistic rewrite theory may be *parametric* on the left-hand side's matching substitution. Therefore, having the entire configuration available in `msgArrival` makes it possible for the probability of message loss to vary depending on the context.

In cases where the possibility of messages being lost varies depending on the kinds of messages being sent, instead of a single `msgArrival` rule we may have one such rule for each kind of message by replacing the variable `M` by message expression patterns characterizing each such class.

3.2.2 Modeling Sensors in Cyber-Physical Systems. The original P transformation [17] has as input untimed non-probabilistic systems with no nondeterministic rewrite rules. Carolyn Talcott has suggested investigating how to allow input real-time rewrite theories that specify distributed cyber-physical systems with objects that periodically read some sensors in their environment.

This is a pertinent question, since the values read by sensors from their environment are unpredictable, i.e., they correspond to nondeterministic inputs to the system amenable to subsequent quantification by probabilities. That is, the input real-time rewrite theory could model the nondeterminism of sensor inputs, and the timed P transformation could then quantify them probabilistically.

The values read from the environment can be modeled by new, nondeterministic variables in a rule's right-hand side. For example, in a plant the periodic reading of values from its uncertain environment and the corresponding actions can be modeled by an instantaneous nondeterministic rewrite rule of the form:

```
[read-sensor-l] :
   < O : Plant | timers : [0 ; p ; td] TIMERS, values : V, atts >
=> < O : Plant | timers : [p ; p ; td] TIMERS, values : NEW-V, atts' > [msgs]
       [nonexec] .
```

or of the same form without '`[msgs]`', `V` denotes the previous value(s), the nondeterministic variable `NEW-V` describes the "new" value(s) read from the environment at the expiration of the current period, *atts* are the remaining attributes of the *Plant* object, which may change to *atts*′ after the transition, and `[msgs]` is the messages to be sent, whose payloads may depend on *atts*, `V`, and `NEW-V`.

3.2.3 Rules with Finitary Nondeterminism. Our last type of nondeterministic rules are message-triggered and timer-triggered rewrite rules where the extra variables in their right-hand sides range over some *finite* sorts. We can therefore say that these nondeterministic rules exhibit *finitary nondeterminism*.

Such finitary nondeterminism arises naturally in systems involving objects that, after receiving a message or upon the expiration of a timer, can *choose* from a finite set of *actions*. For example, some objects in an e-commerce system may model *users* who, at various points in the shopping process, may choose among several possible interactions with the system. We consider nondeterministic rules with finitary nondeterminism that the timed P transformation will refine into useful probabilistic rules for many applications.

For the sake of concreteness let us consider a specific class of rules of this kind in systems where one of its finite sorts, say `FinState`, denotes a finite set Q of states of a transition system defined by a function $next : Q \to \mathcal{P}(Q)$ where $q' \in next(q)$ iff q can make a one-step transition to q'. Assume that both *next* and the membership predicate $\in$ have been equationally defined. Now consider a class `Cl` with an attribute `finState` with values in `FinState`. Some message-triggered and timer-triggered rules for objects of class `Cl` may exhibit finitary nondeterminism, and may have one of the following respective forms:

```
vars S S'  :  FinState .
[l] : (to o [from o'] : mp) < o : Cl | finState : S, atts >
  =>  l-act[S,S',mp,atts] if S' ∈ next(S) = true [and cond] [nonexec] .
```

and

```
[l] : < o : Cl | finState : S,  timers : [0 ; p ; td] timers,  atts >
  =>  l-act'[S,S',td,timers,atts] if S' ∈ next(S) = true [and cond] [nonexec] .
```

where *l*`-act` is an equationally-defined function such that for each $q, q' \in Q$,

– if $q' \in next(q) = false$, then, *l*`-act`$[q, q', mp, atts] =$ `none`

– if $q' \in next(q) = true$, then,

  l-`act[`q`,`q'`,`mp`,`$atts$`] = < `o `: Cl | finState :` q'`, `$atts$-$q.q'$ ` > [`$msgs$-$q.q'$`]`

  or has the same form without '`[`$msgs$`]`' if no messages are sent for the case $q.q'$. All variables appearing in the attributes $atts$-$q.q'$ and in $msgs$-$q.q'$ must occur in either mp or $atts$.

and where `l-act'` is an equationally-defined function such that for each $q, q' \in Q$,

– if $q' \in next(q) = false$, then, l-`act'[`q`,`q'`,`td $timers$`,`$atts$`] = none`
– if $q' \in next(q) = true$, then,

  l-`act'[`q`,`q'`,`td`, `$timers$`,`$atts$`] =`
  `< `o `: Cl | finState :` q'`, timers : [`$period$-$q.q'$ `; `p `; `td`] `$timers$-$q.q'$`,`
  $atts$-$q.q'$ `> [`$msgs$-$q.q'$`]`

  or of the same form without '`[`$msgs$`]`' if no messages are sent for the case $q.q'$. $period$-$q.q'$ is either the non-zero natural number p or `inf`. In the latter case, at most one other timer in $timers$ *may* be reset to its non-zero natural number period. All variables appearing in $timers$-$q.q'$, $atts$-q', and $msgs$-$q.q'$ must appear in $timers$ or $atts$.

3.3 Requirements on Input Theories and Initial States

An initial state `{`$initconf$`}`, where $initconf$ is a ground term of sort `Configuration`, of a real-time rewrite theory in our input class consists of a set of objects with distinct names and an optional sending task, with a list of messages to be sent with undetermined delays, of the form

`< `o_1 `: `C_1 ` | `$atts_1$ ` > ... < `o_n `: `C_n ` | `$atts_n$ ` >`
`[(to `o_{i_1} `[from `o_{l_1}`] : `mp_{i_1}`) ; ... ; (to `o_{i_k} `[from `o_{l_k}`] : `mp_{l_k}`)]`

such that the value of each local `clock` attribute is `0`. The rewrite theory and the initial configuration $initconf$ together satisfy the following requirements:

1. At most one timer in $initconf$ in a single object o_j is turned on with a timer value a natural number, which cannot be 0 unless the initial state does not have a sending task. Indeed, if there is no sending task a single object o_j must be turned on, since otherwise $initconf$ would be a deadlock state.
2. Any object *enabled* by a message-triggered rule to receive a given message addressed to it, is so enabled by the application of a *unique* message-triggered rule with a *unique* substitution of its left-hand side variables.
3. Any object with a single expired timer and enabled to perform a transition by a timer-triggered rule (including such rules with finitary nondeterminism and sensor-reading rules), is so enabled by a *unique* rule among those for that timer, with a *unique* substitution of its left-hand side variables.

4. Any (message-triggered or timer-triggered) transition may set the value of at most *one* existing or new (added by the rule) timer to a natural number, and may turn off some other timers. If the transition was triggered by a timer or the transition generates a sending task, then the reset timer must be reset to a non-zero natural number; furthermore, if the set timer is not the one that expired, then the expired timer must be turned off.
5. All addresses of all messages in the initial states or generated by the timer- and message-triggered rules are names of objects in the initial configuration.

4 The Timed P Transformation

The *timed P transformation* transforms an input real-time rewrite theory and an initial state satisfying the requirements in §3 by using a family Π of parametric probability distributions for:

- delays of messages,
- probability of message losses (if the network is lossy),
- the values read by sensors in "sensor rules" for cyber-physical systems, and
- choosing the next "state" q' in any rule with finitary nondeterminism

into a probabilistic rewrite theory. Formally, the timed P transformation is a mapping $P : (\mathcal{R}, \mathit{init}, \Pi) \mapsto (\mathcal{R}_\Pi, \mathit{init}_\Pi)$ where:

- $\mathcal{R} = (\Sigma, E, L, R)$ is a real-time rewrite theory meeting the input requirements in §3, where all the rules have distinct labels, and where $\texttt{init} \notin L$.
- init is an initial system state satisfying the requirements in §3.3.
- Π is a partial function from the set $L \cup \{\texttt{init}\}$ to parametric probability distributions or, in some cases, pairs and/or families of such distributions. Π is defined as follows:
  1. For a message-triggered or timer-triggered rule l that is not a rule with finitary nondeterminism or a sensor rule, and that has a sending task in its right-hand side, Π has a map $l \mapsto \pi_l(\boldsymbol{x})$, where $\boldsymbol{x}$ are the variables in the left-hand side of rule l and $\pi_l(\boldsymbol{x})$ is a parametric *continuous* probability distribution on the non-negative reals $\mathbb{R}_{\geq 0}$, defining the probability distribution over the delays of the messages generated by the rule l as a function of the state of the sending object and (for message-triggered rules) the received message. Further assumptions on the message delay distributions $\pi_l(\boldsymbol{x})$ are detailed in §4.4.
  2. When a lossy network is assumed, the mapping in Π for the rule label `msgArrival` is $\texttt{msgArrival} \mapsto \pi_{\texttt{msgArrival}}(\boldsymbol{x})$, where $\boldsymbol{x}$ are the variables in the rule's left-hand side and $\pi_{\texttt{msgArrival}}(\boldsymbol{x})$ is a parametric probability distribution on the Booleans, defining the probability that a message is lost as a function of the message or of the message and the entire state (depending on which version of the rule `msgArrival` is used).
  3. For a sensor-reading rule `read-sensor-`l that generates a message-sending task and whose left-hand side variables are $\boldsymbol{x}$, the mapping in Π is

$$\texttt{read-sensor-}l \mapsto (\pi_{\texttt{read-sensor-}l.val}(\boldsymbol{x}), \pi_{\texttt{read-sensor-}l}(\boldsymbol{x}))$$

where $\pi_{\texttt{read-sensor-}l.val}(\boldsymbol{x})$ is a parametric probability distribution on the data type of `Values`, and $\pi_{\texttt{read-sensor-}l}(\boldsymbol{x})$ is a parametric continuous probability distribution on $\mathbb{R}_{\geq 0}$. If the rule does not generate a sending task, then the mapping is just $\texttt{read-sensor-}l \mapsto \pi_{\texttt{read-sensor-}l.val}(\boldsymbol{x})$.

4. For a rule l with finitary nondeterminism that uses a finite transition system $\mathcal{Q} = (Q, \rightarrow_{\mathcal{Q}})$, and with left-hand side variable `S` of sort `FinState`, the mapping in Π has the form

$$l \mapsto (\pi_{l.\mathcal{Q}}(S), \{\pi_{l_{q.q'}}(\boldsymbol{x} \setminus \{\texttt{S}\})\}_{q \in Q, q' \in next(q) \wedge generatesMsgs(q,q')})$$

where $\boldsymbol{x}$ are the variables in the rule's left-hand side. $\pi_{l.\mathcal{Q}}(S)$ is the parametric probability distribution $\pi_{l.\mathcal{Q}}(S) = \{\pi_{l.\mathcal{Q}}(S)\}_{S \in Enabled(Q)}$, where $Enabled(Q)$ denotes the complement of the set of terminating states in $\mathcal{Q}$ and each $\pi_{l.\mathcal{Q}}(q)$ is a *discrete* probability distribution on $next(q)$ in $\mathcal{Q}$. Therefore, $\pi_{l.\mathcal{Q}}(S)$ defines a *Markov chain* quantifying the nondeterminism of the transition system $\mathcal{Q}$. $\{\pi_{l_{q.q'}}(\boldsymbol{x} \setminus \{\texttt{S}\})\}_{q \in Q, q' \in next(q) \wedge generatesMsgs(q,q')}$ is a family of parametric continuous probability distributions over $\mathbb{R}_{\geq 0}$ providing probability distributions for the delays of the messages generated when the rule l is applied in "state" q and chooses q' as its next state, but only for those $q.q'$-pairs where a sending task is generated.

5. Finally, Π contains a map $\texttt{init} \mapsto \pi_{\texttt{init}}$, where $\pi_{\texttt{init}}$ is a continuous probability distribution on $\mathbb{R}_{\geq 0}$ defining the probability distribution of the delays of the messages in the initial state.

– $\mathcal{R}_{\Pi} = (\Sigma_{\Pi}, E_{\Pi}, L_{\Pi}, R_{\Pi})$ is the probabilistic rewrite theory defined below.
– $init_{\Pi}$ is the timed P-transformed initial state defined in §4.3.

In what follows, we define in detail the probabilistic theory $\mathcal{R}_{\Pi}$ and the initial state $init_{\Pi}$ obtained by applying the timed P transformation to $(\mathcal{R}, init, \Pi)$.

4.1 The Equational Theory (Σ_{Π}, E_{Π})

The equational theory (Σ_{Π}, E_{Π}) extends (Σ, E), and also contains the functions required to define the continuous probability distributions in Π. The timed P transformation removes the subsort declaration `STask < Configuration` and adds a new sort `LSTask` (for "labeled sending task") as a subsort of `Configuration`.

The P transformation also adds, for each rule l that may generate new messages and whose left-hand side variables are $\boldsymbol{x} = x_1 : s_1, \ldots, x_n : s_n$, an operator

```
op msgDly_l : s_1 ... s_n STask -> LSTask [ctor] .
```

However, if the rule l is a rule with finitary nondeterminism whose left-hand side has variables `S` and $\boldsymbol{x} = x_1 : s_1, \ldots, x_n : s_n$, then we instead add an operator

```
op msgDly_{l_{q.q'}} : s_1 ... s_n STask -> LSTask [ctor] .
```

for each transition $q \rightarrow q'$ "in" l that may generate messages. We also declare the function `op msgDly`$_{\texttt{init}}$ `: STask -> LSTask [ctor]` used for probabilistically quantifying the delays in sending tasks for initial states.

The equations for l-`act` and l-`act'` are also modified, as explained under the treatment of rules with finitary nondeterminism in §4.2.

The equations for `mte` and `timeEffect` are modified by removing the equations for sending tasks, and adding the equations

```
var LSTASK : LSTask .  eq mte(LSTASK) = 0 .  eq timeEffect(LSTASK, T) = LSTASK .
```

4.2 The Rewrite Rules R_{Π}

This section explains how each kind of the rewrite rules § 3.2 is transformed by the timed P transformation, which leaves the tick rule in §3.1 unchanged.

Deterministic Message-Triggered Rewrite Rules. The timed P transformation transforms each deterministic message-triggered rewrite rule in § 3.2 that generates a sending task into the corresponding rewrite rule

```
[l] : (to o [from o'] : mp) < o : Cl | atts >
    =>                      < o : Cl | atts' >  msgDly_l(x, [msgs]) [if cond] .
```

where $\boldsymbol{x}$ are the variables in the left-hand side of rule l. Deterministic message-triggered rules that do not generate new messages are not modified by P.

Deterministic Timer-Triggered Rules. Each deterministic timer-triggered Rule l that may generate messages is transformed by P into the timer-triggered rule

```
[l] : < o : C | timers : [0 ; p ; td] otherTimers, atts >
  => < o : C | timers : [v ; p' ; td] otherTimers, atts' > msgDly_l(x, [msgs]) [if cond] .
```

Such rules that do not generate messages are not modified by P.

Network Transmission Rules. P transforms the rule `sendDelayed` into a *family* of *probabilistic* rewrite rules parameterized by those rule labels $l \in L$ that are in the domain of Π and where l generates a message task as follows. If l is not a rule with finitary nondeterminism, then the corresponding rule is:

```
[sendDelayed_l] :
 msgDly_l(x, [M ; ML]) => dly(M, T) msgDly_l(x, [ML]) with probability T := π_l(x) .
```

If l labels a rule with finitary nondeterminism, there are as many probabilistic rules as transitions $q \to q'$ such that when l changes q to q' some messages can be sent. Each such probabilistic rule has the form:

```
[sendDelayed_{l_{q.q'}}] :
msgDly_{l_{q.q'}}(x, [M ; ML]) => dly(M,T) msgDly_{l_{q.q'}}(x,[ML]) with probability T := π_{l_{q.q'}}(x) .
```

The rule `sendNil` is transformed by P into `[sendNil`$_l$`]` : `msgDly`$_l$`(`$\boldsymbol{x}$`,[nil]) => none` for each l as above, and a similar rule for each "message-generating" $_{l_{q.q'}}$. If the initial state contains a sending task, R_{Π} also contains the rewrite rules

```
[sendDelayed_init] :
 msgDly_init([M ; ML]) => dly(M, T) msgDly_init([ML]) with probability T := π_init .
[sendNil_init] : msgDly_init([nil]) => none .
```

Message Arrival and Loss. There are three versions of the rule `msgArrival`. P leaves the first unchanged, and transforms either of the other two into the corresponding *probabilistic* rewrite rule

```
[msgArrival] :
 dly(M,0) => if B then M else none fi with probability B := π_msgArrival(M) .
    or
[msgArrival] :
{dly(M,0) C}  => if B then {M C} else {C} fi
                 with probability B := π_msgArrival(M,C) .
```

Sensor Rules. The timed P transformation transforms each *sensor* rule `read-sensors-`l that generates messages into the *probabilistic* rewrite rule

```
[read-sensor-l] :
   < o : Plant | values : V,     timers : [0 ; P ; td] TIMERS, atts >
=> < o : Plant | values : NEW-V, timers : [P ; P ; td] TIMERS, atts' >
   msgDly_read-sensors-l(x, [msgs]) [if cond]
       with probability NEW-V := π_read-sensors-l.val(x) .
```

where $\boldsymbol{x}$ are the variables in the left-hand side of the rule, which include the variable `V` for the previously read sensor value. A sensor rule that does not generate messages is transformed in a similar way by P.

Rules with Finitary Nondeterminism. P transforms a message-triggered rule l with finitary nondeterminism in §3.2.3 into the *probabilistic* rewrite rule

```
[l] : (to o from o' : mp)  < o : C | finState: S, atts >
 =>   l-act[S,S',mp,atts]  [if cond]  with probability S' := π_l.Q(S) .
```

and transforms the *equations* involving sending tasks into the equations

```
eq l-act[q,q',atts] = < o : C | finState : q', atts-q' > msgDly_l_q.q'(x,[msgs-q.q'])
```

where $\boldsymbol{x}$ are the variables in the rule l except for `S`, which are present in *atts* in this equation. If the above equation does not have a sending task ([*msgs-q.q'*]) in its right-hand side, then the equation is not modified by P. The timed P transformation is similar for timer-triggered rules with finitary nondeterminism.

4.3 The Initial State $\boldsymbol{init_\Pi}$

The timed P transformation transforms an initial state *init* of the form {*objects* [*messageList*]} into the initial state $init_\Pi$, which is the term {*objects* $\mathtt{msgDly}_{\mathtt{init}}$([*messageList*])}.

4.4 Assumptions on the Message Delay Distributions in Π

The message delay distributions $\pi_l(\boldsymbol{x}_l)$ are continuous distributions on the non-negative reals $\mathbb{R}_{\geq 0}$ parametric on the left-hand side variables $\boldsymbol{x}_l$ of rule l. Likewise, the distribution $\pi_{\texttt{init}}$ is a continuous distribution on $\mathbb{R}_{\geq 0}$. These continuous distributions model the network's probabilistic behavior when sending the messages. The following assumptions from [17] will be used:

1. For each message- and timer-triggered rule l that generates messages and each ground substitution $\theta = \{\boldsymbol{x}_l \mapsto \boldsymbol{a}\}$, instantiating the parameters $\boldsymbol{x}_l$ of l's left-hand side (resp. for $l = init$) there is a piecewise continuous *probability density function* $f_l(\boldsymbol{a}) : \mathbb{R} \to [0, +\infty)$ (resp. f_{init}) such that—since message delays are always non-negative—for $x < 0$, $f_l(\boldsymbol{a})(x) = 0$ (resp. $f_{init}(x) = 0$).
2. Each such density function defines a *probability distribution function* $\pi_l(\boldsymbol{a}) : \mathbb{R} \to [0, 1]$ (resp. $\pi_{init} : \mathbb{R} \to [0, 1]$) as a continuous and almost everywhere differentiable function of the form $\pi_l(\boldsymbol{a}) = \lambda x \in \mathbb{R}.\ \int_{-\infty}^{x} f_l(\boldsymbol{a})(t)\ dt$ (likewise for π_{init}). By the assumptions on $f_l(\boldsymbol{a})$ and f_{init}, $\pi_l(\boldsymbol{a})(x) = 0$ and $\pi_{init}(x) = 0$, for $x \leq 0$. More generally, yet equivalently ([16], Theorem 1.88), $f_l(\boldsymbol{a})$ defines a *probability measure* $\mu_l(\boldsymbol{a}) : \mathcal{B}(\mathbb{R}) \to [0, 1]$ on the Borel σ-algebra of $\mathbb{R}$, defined by $\mu_l(\boldsymbol{a}) = \lambda B \in \mathcal{B}(\mathbb{R}).\ \int_{t \in B} f_l(\boldsymbol{a})(t)\ dt$.
3. In what follows, $X_{f_l(\boldsymbol{a})}$, which we call the *support* of $f_l(\boldsymbol{a})$, will denote the set $X_{f_l(\boldsymbol{a})} = \{x \in \mathbb{R} \mid f_l(\boldsymbol{a})(x) > 0\}$, and $\overline{X}_{f_l(\boldsymbol{a})}$ will denote its topological closure, obtained by adding to $X_{f_l(\boldsymbol{a})}$ its limit points. The set $\overline{X}_{f_l(\boldsymbol{a})}$ provides a useful "envelope" for the values obtained by sampling $\pi_l(\boldsymbol{a})$. Indeed, by openness, for any $x \in \mathbb{R} \setminus \overline{X}_{f_l(\boldsymbol{a})}$ there is an open interval $(a, b) \subseteq \mathbb{R} \setminus \overline{X}_{f_l(\boldsymbol{a})}$ with $x \in (a, b)$, so that $\mu_l(\boldsymbol{a})((a, b)) = \pi_l(\boldsymbol{a})(b) - \pi_l(\boldsymbol{a})(a) = 0$. Therefore, any real number r obtained by sampling the distribution $\pi_l(\boldsymbol{a})$ must be inside the envelope $\overline{X}_{f_l(\boldsymbol{a})}$. Notice, furthermore, that the assumption on $f_l(\boldsymbol{a})$ and f_{init} forces the inclusions $\overline{X}_{f_l(\boldsymbol{a})} \subseteq [0, +\infty)$ and $\overline{X}_{f_{init}} \subseteq [0, +\infty)$.

5 Correctness of the Timed P Transformation

The timed P transformation maps a real-time rewrite theory $\mathcal{R}$ meeting P's input requirements into the probabilistic rewrite theory $\mathcal{R}_\Pi$. The purpose of $\mathcal{R}_\Pi$ is to both *quantify* and perhaps *restrict* the nondeterministic behaviors of $\mathcal{R}$. But is P correct? Two key correctness requirements are that: (1) the *behaviors* of $\mathcal{R}_\Pi$ are always semantically related to those of $\mathcal{R}$; this is proved in §5.1; and (2) $\mathcal{R}_\Pi$ is a *purely probabilistic system* and can therefore be formally analyzed by statistical model checking; this is proved in §5.2.

5.1 Faithful Behavioral Correspondence between $\mathcal{R}_\Pi$ and $\mathcal{R}$

$\mathcal{R}_\Pi$ is a non-executable mathematical model. To talk about its "behaviors" we need to *simulate* $\mathcal{R}_\Pi$ by *sampling* the probability distributions in its probabilistic rewrite rules. But there is a simpler way of comparing $\mathcal{R}$ and $\mathcal{R}_\Pi$ than via

sampling: we can use the so-called *nondeterministic envelope* $NdEnv(\mathcal{R}_\Pi)$ of $\mathcal{R}_\Pi$, a nondeterministic real-time rewrite theory that contains all the behaviors that could be obtained by sampling $\mathcal{R}_\Pi$ without any need to bring sampling techniques into the picture. Due to space limitations we sketch $NdEnv(\mathcal{R}_\Pi)$'s construction and refer to [21] for full details. The key idea is to replace each probabilistic rule in $\mathcal{R}_\Pi$ by a nondeterministic rule such that its choices of values are those obtainable by sampling the probabilistic rule's distribution. For example, each rule of the form $\texttt{sendDelayed}_l$ is replaced by the nondeterministic rule:

$$\texttt{[sendDelayed}_l\texttt{] :}$$
$$\texttt{msgDly}_l\texttt{(}\boldsymbol{x}\texttt{, [M ; ML]) => dly(M,T) msgDly}_l\texttt{(}\boldsymbol{x}\texttt{, [ML])}\ \textit{if}\ \texttt{T} \in \overline{X}_{f_l}(\boldsymbol{x})\ \texttt{[nonexec] .}$$

where, as explained in §4.4, $\overline{X}_{f_l}(\boldsymbol{a})$ is the *envelope* for the values obtained by sampling the distribution $\pi_l(\boldsymbol{a})$ for each instantiation of $\boldsymbol{x}$ to ground values $\boldsymbol{a}$.

We can relate the behaviors of $NdEnv(\mathcal{R}_\Pi)$ and those of $\mathcal{R}$ by means of a *simulation map* $h : NdEnv(\mathcal{R}_\Pi) \to \mathcal{R}$, which, by definition, is a function h mapping states of $NdEnv(\mathcal{R}_\Pi)$ to states of $\mathcal{R}$ in such a way that, for any state u of $NdEnv(\mathcal{R}_\Pi)$ that can perform a rewrite transition $u \to u'$ in $NdEnv(\mathcal{R}_\Pi)$, the state $h(u)$ can perform a rewrite transition $h(u) \to h(u')$ in $\mathcal{R}$. The fact that qualitative properties of $\mathcal{R}$ and the quantitative properties of $\mathcal{R}_\Pi$ are properties of the *same* system is ensured by the function h defined in [21] that, by the theorem below, is a simulation map as proved in [21].

Theorem 1. *The function h defines a simulation map of rewrite theories $h : NdEnv(\mathcal{R}_\Pi) \to \mathcal{R}$.*

5.2 Absence of Nondeterminism (AND)

To analyze the quantitative properties of $\mathcal{R}_\Pi$ by statistical model checking it is highly desirable that the sampling-based simulation of $\mathcal{R}_\Pi$ from an initial state $init_\Pi$ is a *purely probabilistic system*. This just means that for any state reachable from $init_\Pi$ at most one rewrite transition (probabilistic or not) is possible, since otherwise the system would exhibit a form of nondeterminism. In more detail, this means that the following *absence of nondeterminism* (AND) property holds:

Definition 1. *(AND) [17]. With probability 1, for any state reachable from $init_\Pi$ by a sequence of sampling-based rewrite steps (i.e., by a Monte Carlo simulation of $\mathcal{R}_\Pi$), there is at most one rewrite rule applicable, at a unique position and with a unique matching substitution θ for its left-hand side; i.e., two different rules, or the same rule, can never be applied to a reachable state either at different positions or with different matching substitutions.*

The proof of the following theorem can be found in [21].

Theorem 2. *If the real-time rewrite theory $\mathcal{R}$ and the initial state init satisfy the input requirements of the timed P transformation, then $\mathcal{R}_\Pi$ from initial state $init_\Pi$ satisfies the AND property.*

6 A Design and Analysis Methodology for Distributed Real-Time Systems

As this work has shown, a distributed real-time system design expressed as a real-time rewrite theory often includes nondeterministic rules that are non-executable. This poses challenges for debugging such designs *prior* to analyzing their quantitative properties, since the design may contain bugs and design flaws that are hard to uncover without testing the system. Furthermore, verification of qualitative properties, such as safety properties, may also be important. This section outlines a design and analysis methodology addressing these challenges:

1. **Design Capture** of the intended system as a real-time rewrite theory $\mathcal{R}$.
2. **Execution and Testing** of $\mathcal{R}$ following the methodology outlined in §2.2: By choosing finite subsets of values for nondeterministic variables, all rules can become executable by adding matching conditions to the rules to select specific values in the domain. After testing and debugging $\mathcal{R}$ through execution, Maude's `search` command can exhaustively explore all behaviors from an initial state that satisfy the restrictions imposed by the choices of values.
3. **Symbolic Execution and Reachability Analysis**. The original specification $\mathcal{R}$, while non-executable, is *symbolically* executable by *rewriting modulo SMT* [25]. The idea is that we can symbolically execute, not concrete states, but symbolic ones of the form $u(\boldsymbol{x}) \mid \varphi(\boldsymbol{x})$ where $\varphi(\boldsymbol{x})$ is a formula in a decidable theory constraining the values of the variables $\boldsymbol{x}$. For example, some of those variables may range over the real numbers, and $\varphi(\boldsymbol{x})$ may contain constraints over those variables in an SMT-decidable theory such as linear arithmetic. Rewriting modulo SMT makes it possible to verify *safety properties* by reachability analysis. For formal analyses of real-time systems using rewriting modulo SMT, see [6,5,7].
4. **Simulation of** $\mathit{Sim}(\mathcal{R}_{\Pi})$. The *Sim* transformation [17] makes the probabilistic rewrite theory $\mathcal{R}_{\Pi}$ executable by *sampling* the probability distribution of each rule. This allows users to *simulate* the probabilistic behavior of $\mathcal{R}_{\Pi}$.
5. **Statistical Model Checking of** $\mathit{Sim}(\mathcal{R}_{\Pi})$. Thanks to Theorem 2, it is possible to analyze the behaviors of $\mathcal{R}_{\Pi}$ by *statistical model checking*. This can be done using, e.g., the QMaude tool [26] for properties expressed in the QuaTEx probabilistic temporal logic.

7 Related Work

The timed P transformation extends the P transformation in [17], whose inputs are *untimed* generalized actor rewrite theories. The (timed) input theories in this paper support timers and message delays, which make no sense for untimed models. The input theories in [17] do not support nondeterminism in the rules themselves, whereas we support nondeterministic sensor rules, rules with finitary nondeterminism, and nondeterministic rule(s) for message loss/delivery. On the other hand, the input theories in [17] support support both message-triggered

rules (which we also do) and more general "object-triggered rules," whereas in this work we focus on *timer-triggered* rules, which are the kind of object-triggered rules most typical in real-time systems.

Lingua Franca [18] and Timed Rebeca [1] are two well-known timed actor languages. Lingua Franca also supports both message-triggered and timer-triggered (re)actions. Lingua Franca provides "deterministic concurrency:" the only source of nondeterminism are external "physical actions," which can take place at "any" time and have any value. In contrast to our input theories, the delay of all messages between the same pair of ports is the same, timer periods do not change, and there are no message losses. On the other hand, the time- and value-nondeterministic physical actions in Lingua Franca are only indirectly supported by our theories, in which such an action can be simulated by sending a message with nondeterministic delay, setting a sensor timer to 0 when it arrives, which triggers a `read-sensor` rule to obtain a value nondeterministically.

Timed Rebeca [1] extends the actor language Rebeca [28] to discrete real-time systems. Timed Rebeca supports delay statements, messages with delay intervals, and nondeterministic assignment of state variables to any value in a finite set. Unlike in our input theories, the following are not supported: timers (although periodic behavior can be simulated by an actor sending a message to itself); "sensor rules" sampling infinite domains; message loss; and dense time.

The main contribution of our paper is to define a transformation that maps a nondeterministic timed actor system into a purely probabilistic timed one, and proving a semantic relationship between the two models.

Although probabilistic timed versions of Rebeca exist [15,14], statistical model checking is only supported to choosing the next state and assign variable values from discrete distributions. Continuous distributions, important for performance analysis, are not supported. The main contrast with our paper is that there is no work mapping a Timed Rebeca model to a probabilistic timed model.

We are not aware of any probabilistic version of Lingua Franca, or of any mapping from Lingua Franca programs to probabilistic ones.

Other pairs of related nondeterministic and probabilistic formalisms are discussed in [17]; examples include timed [4] and probabilistic and stochastic timed automata [12], and BIP [8] and $\mathcal{S}$BIP [22]. However, we are unaware of any work mapping qualitative timed distributed systems to quantitative ones.

Recent work [23] by Olarte et al. provides a "probabilistic" Maude semantics to probabilistic Event-B to support statistical model checking of such Event-B models using `umaudemc`. Like our finitary nondeterminism and sensor rules, their model supports "external inputs [...] uniformly chosen from finite sets" and "probabilistic assignments to variables [which] model uncertainty in 'external' data." The differences between our finitary nondetermism and sensor rules are that [23] only supports non-parametric *uniform* discrete distributions, and does not support user-defined (discrete or continuous) distributions which, furthermore, in our case could be parametric on the current state. The main difference between our work and the work in [23] is that the timed P transformation takes as inputs and yields as outputs *timed* systems, whereas all models in [23] are un-

timed (and therefore do not involve timers and message delays). We prove that our resulting models are fully probabilistic "by construction," whereas Olarte et al. obtain fully probabilistic models by adding (discrete) weights to the different events, akin to how weights are added to rewrite rules (and substitutions and contexts) in QMaude [26] and PSMaude [9] to obtain fully probabilistic models.

8 Concluding Remarks

This work is a further step in the effort initiated in [17] to bridge the semantic gap between the different models used to verify qualitative and quantitative properties of distributed systems. In [17] this was done for *untimed* systems. The main contribution of this paper has been to extend the P transformation to a general class of distributed *real-time* systems that extends timed actor systems. The *soundness* of the quantitative analysis of distributed real-time systems supported by our *timed* P transformation crucially depends on its correctness. We have proved two key correctness results. The first proves the semantic consistency between the input real-time rewrite theory $\mathcal{R}$ and its P-transformed theory $\mathcal{R}_{\Pi}$, thus ensuring that $\mathcal{R}$ and $\mathcal{R}_{\Pi}$ are models of the *same* system at different levels of abstraction. The second shows that $\mathcal{R}_{\Pi}$ is a purely probabilistic model and makes possible the automated quantitative analysis of $\mathcal{R}_{\Pi}$ by statistical model checking. We have also discussed in §6 how our timed P transformation supports a *formal design and verification methodology* for distributed real-time systems.

Much work remains ahead to bring such a design and verification methodology to fruition. First of all, the *automation* of the timed P transformation needs to be implemented. Our goal is to make this transformation available for users of the QMaude tool [26]. This will greatly facilitate the access to QMaude by real-time distributed system designers. Second, this should be done in conjunction with substantial experimentation, resulting in a suite of case studies demonstrating the usefulness of the timed P transformation and its supporting tools in the design and both qualitative and quantitative verification of distributed systems.

Acknowledgments. We are grateful to Joud Khoury, Minyoung Kim, Narciso Martí-Oliet, Christophe Merlin, Carolyn Talcott, and the anonymous reviewers for helpful comments on an earlier version of this paper, and to Rubén Rubio for his help with the QMaude tool. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No HR001125CE019.

References

1. Aceto, L., Cimini, M., Ingólfsdóttir, A., Reynisson, A.H., Sigurdarson, S.H., Sirjani, M.: Modelling and simulation of asynchronous real-time systems using Timed Rebeca. In: FOCLASA. EPTCS, vol. 58, pp. 1–19 (2011)
2. Agha, G.: Actors. MIT Press (1986)

3. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. **28**(1), 6:1–6:39 (2018)
4. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2), 183–235 (1994)
5. Arias, J., Bae, K., Olarte, C., Ölveczky, P.C., Petrucci, L.: A rewriting-logic-with-SMT-based formal analysis and parameter synthesis framework for parametric time Petri nets. Fundamenta Informaticae **192**(3-4), 261–312 (2024)
6. Arias, J., Bae, K., Olarte, C., Ölveczky, P.C., Petrucci, L., Rømming, F.: Symbolic analysis and parameter synthesis for networks of parametric timed automata with global variables using Maude and SMT solving. Science of Computer Programming **233**, 103074 (2024)
7. Bae, K., Olarte, C., Ölveczky, P.C.: Modeling and analyzing real-time systems in rewriting logic. In: Concurrent Programming, Open Systems and Formal Methods: Essays Dedicated to Gul Agha to Celebrate His Scientific Career. pp. 494–535. Springer Nature Switzerland, Cham (2026)
8. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Softw. **28**(3), 41–48 (2011)
9. Bentea, L., Ölveczky, P.C.: A probabilistic strategy language for probabilistic rewrite theories and its application to cloud computing. In: Proc. WADT 2012. pp. 77–94. Lecture Notes in Computer Science, Springer (2012)
10. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude – A High-Performance Logical Framework. Springer LNCS Vol. 4350 (2007)
11. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B.: Uppaal SMC tutorial. Int. J. Softw. Tools Technol. Transf. **17**(4), 397–415 (2015)
12. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B.: Uppaal SMC tutorial. Int. J. Softw. Tools Technol. Transf. **17**(4), 397–415 (2015)
13. Goguen, J., Meseguer, J.: Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theoretical Computer Science **105**, 217–273 (1992)
14. Jafari, A., Khamespanah, E., Kristinsson, H., Sirjani, M., Magnusson, B.: Statistical model checking of Timed Rebeca models. Comput. Lang. Syst. Struct. **45**, 53–79 (2016)
15. Jafari, A., Khamespanah, E., Sirjani, M., Hermanns, H., Cimini, M.: PTRebeca: Modeling and analysis of distributed and asynchronous systems. Science of Computer Programming **128**, 22–50 (2016)
16. Klenke, A.: Probability Theory. Springer (2006)
17. Liu, S., Meseguer, J., Ölveczky, P.C., Zhang, M., Basin, D.A.: Bridging the semantic gap between qualitative and quantitative models of distributed systems. Proc. ACM Program. Lang. **6**(OOPSLA2), 315–344 (2022)
18. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a Lingua Franca for Deterministic Concurrent Systems. ACM Transactions on Embedded Computing Systems (TECS) **20**(4), 1–27 (2021)
19. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)
20. Meseguer, J.: Twenty years of rewriting logic. J. Algebraic and Logic Programming **81**, 721–781 (2012)
21. Meseguer, J., Ölveczky, P.C.: The timed P transformation for real-time distributed systems (longer report) (2026), http://olveczky.se/timed-p-techrep.pdf

22. Nouri, A., Mediouni, B.L., Bozga, M., Combaz, J., Bensalem, S., Legay, A.: Performance evaluation of stochastic real-time systems with the SBIP framework. Int. J. Crit. Comput. Based Syst. **8**(3/4), 340–370 (2018)
23. Olarte, C., Osorio, D., Ramírez, C., Rocha, C.: Algorithmic analysis of Event-B in rewriting logic. In: Proc. NASA Formal Methods. pp. 275–293. Springer Nature Switzerland, Cham (2025)
24. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theoretical Computer Science **285**(2), 359–405 (2002)
25. Rocha, C., Meseguer, J., Muñoz, C.A.: Rewriting modulo SMT and open system analysis. Journal of Logic and Algebraic Methods in Programming **86**, 269–297 (2017)
26. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: QMaude: Quantitative specification and verification in rewriting logic. In: Chechik, M., Katoen, J., Leucker, M. (eds.) Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14000, pp. 240–259. Springer (2023)
27. Rubio, R., Riesco, A., Martí-Oliet, N.: PMaude revisited through probabilistic strategies. In: Concurrent Programming, Open Systems and Formal Methods: Essays Dedicated to Gul Agha to Celebrate His Scientific Career. pp. 475–493. Lecture Notes in Computer Science, Springer (2026)
28. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundamenta Informaticae **63**(4), 385–410 (2004)

A semantic framework for symbolic execution in Maude

Rafael Morales-Palacios, Rubén Rubio, and Ignacio Fábregas

Universidad Complutense de Madrid, Madrid, Spain
{rafmor02, rubenrub, fabregas}@ucm.es

Abstract. Symbolic execution is a mainstream automated technique for analyzing computer programs. However, most implementations are developed for a specific programming language and without taking formal semantics into account. This paper introduces a general framework for specifying operational semantics in the Maude specification language, based on rewriting logic, and analyzing their programs using symbolic execution. User-defined semantics are translated to the appropriate setting for concrete execution, symbolic execution, or combinations of them, like concolic testing. Symbolic reasoning is mainly implemented through SMT solvers, but the possibilities of narrowing are also explored.

1 Introduction

Symbolic execution [4] is a widely-used approach for semiautomated program analysis, bug finding, test case generation, and sometimes functional verification. Its basic non-trivial idea is to run a program with symbolic values instead of actual inputs to cover the whole or a wider range of executions. Several techniques and tools have been developed to deal with specific programming languages, applications, and bug categories. Even the mainstream compilers for C and C++, GCC and Clang, include built-in analyzers based on symbolic execution. However, most of these tools are ad-hoc implementations not built on top of a formal definition of the language semantics, as already pointed out by the authors of $\mathbb{K}$ [27], a general framework for specifying formal semantics that is able to run symbolic executions for them. Some limitations of symbolic methods are the path explosion problem and the complexity of dealing with loops, complex or custom datatypes, memory, or the environment, which are not alien to other verification approaches. Hybridization with concrete executions, like concolic testing, can mitigate some of these problems.

Maude [9] is a high-level, high-performance specification and programming language based on rewriting logic [21]. It has been extensively used as a logic and semantic framework [20,31], where deduction and computation are naturally expressed by means of rules on a custom signature of terms modulo equations and structural axioms, like associativity and commutativity. Semantics for several languages have been specified and exploited in Maude [14,17,33]. Indeed, the already mentioned $\mathbb{K}$ framework was first developed using Maude. However, its

support for symbolic computation has substantially improved since then [12,22], with connections to SMT solvers [26] and symbolic reachability analysis via narrowing [11]. Moreover, the MaudeSE tool [34] has extended the built-in SMT support in Maude.

In this work, we develop a generic semantic infrastructure and some reflective transformations to run SMT-based symbolic executions on programs within Maude. Given a user-defined executable semantics satisfying some constraints, we derive a module where symbolic reachability problems can be solved by rewriting modulo SMT with MaudeSE or the Maude `search` command. Moreover, an extension of this transformation can be used for concolic testing, which involves simultaneous concrete and symbolic execution. Since concolic testing requires obtaining models from SMT formulae, which is not supported by Maude, we have developed an extension of the Maude SMT interface via the Maude Python library [28] for this purpose. Finally, a third translation shows that the framework can also be used for symbolic execution with narrowing. We illustrate the framework by detecting bugs and checking properties in some small examples written in a simple imperative language.

The paper is organized as follows: after discussing related work in Section 2 and some preliminaries in Section 3, the generic semantic infrastructure and its transformations are respectively presented in Section 4 and Section 5, and Section 6 illustrates it with some examples. Source code, these and more examples can be found in [23].

2 Related work

Some applications of symbolic execution are available in many end-user development tools. For instance, the widespread GNU compiler has recently incorporated a built-in bug finder based on coverage-guided symbolic execution for C programs; the Clang compiler already used it in its static analyzer for C, C++, and Objective-C; and Microsoft's Visual Studio is able to generate test cases by symbolic methods.[1] Other well-known open-source and commercial tools are KLEE [6], Facebook's Infer [7], GNAT SAS for Ada, or SonarQube. Some examples in Section 6 portray some of the typical simple checks available in these tools.

Symbolic execution tools are not usually based on a formal symbolic semantics of the underlying programming language, or this semantics is developed separately from its concrete counterpart. In [32], the authors propose a rule format to simultaneously specify both concrete and symbolic big-step operational semantics, and illustrate it with the specification of the typical *While* language. We use a similar *While* language in the examples of this paper, although with a small-step semantics. Other works have formalized symbolic execution in the context of denotational and operational semantics [18,5]. However, the closest reference for program analysis based on a given formal semantics is the $\mathbb{K}$ Framework [27], which actively maintains a symbolic execution engine implemented in

[1] See `gcc.gnu.org`, `clang-analyzer.llvm.org`, and `visualstudio.microsoft.com`.

Haskell. The theoretical foundations for symbolic execution in $\mathbb{K}$ and a first prototype in Java were developed by Lucanu, Rusu, and Arusoaie [19,3]. It relies on syntactic unification and the Z3 SMT solver for applying the semantic rules on program configurations with variables, and uses a program transformation to integrate this symbolic search into $\mathbb{K}$. This is similar to the approach we follow in Section 4. That tool was used for Hoare and reachability logic verification, and bounded LTL model checking with some examples, while concolic testing is mentioned as future work. At that time, $\mathbb{K}$ used Maude as a rewriting engine, so model checking was done by the Maude LTL model checker [13]. The support for symbolic execution parameterized on the semantics in $\mathbb{K}$ is much more developed than the work we present in this paper, but we are doing it using the general-purpose symbolic features of Maude and applying other methods like concolic testing and equational narrowing.

As mentioned in the introduction, symbolic reasoning features in Maude have experimented important advances in its latest versions [12,22], but they were available since a long time. The most significant application of narrowing is the Maude-NPA tool [16] for cryptographic protocol analysis, but we can also mention a symbolic model checker for quantum circuits [10]. Regarding the interaction with SMT solvers, networks of parameterized timed automata [2] and parametric time Petri nets [1] have recently been analyzed using the external extension MaudeSE [34]. However, we do not know about any attempt to apply the narrowing features and connections with SMT solvers in Maude to analyze programs parameterized on their language semantics.

3 Preliminaries

In this section, we recall some basic notions on rewriting logic and Maude [9], rewriting modulo SMT [26], symbolic execution, and concolic testing [30]. We assume the reader is familiar with order-sorted signatures Σ over a poset of sorts $(S, \leq)$, the free algebra $T_\Sigma(X)$ of terms over a signature Σ with variables X, substitutions $\sigma : X \to T_\Sigma(X)$, and their homomorphic extension to terms $\sigma : T_\Sigma(X) \to T_\Sigma(X)$, which, abusing notation, we denote by the same symbol.

Symbolic execution. Symbolic execution generalizes standard execution by operating on symbolic values representing arbitrary inputs rather than concrete data. In this context, every program state maintains a path condition ϕ, a first-order logic formula that accumulates constraints over the symbolic inputs. Upon encountering a conditional statement with guard ψ, the execution bifurcates into two paths, updating the respective path conditions with $\phi \wedge \psi$ and $\phi \wedge \neg \psi$. An execution state is deemed feasible if and only if its associated path condition is satisfiable, a verification step typically performed by an SMT solver.

Concolic testing. Concolic testing is a hybrid technique designed to address the limitations of pure symbolic execution, such as the path explosion problem and the handling of external system calls, by running both concrete and symbolic

executions at the same time. The process operates on a *dual state*, maintaining both concrete values and their symbolic counterparts. The execution begins with a concrete input that drives the program along a specific path. Simultaneously, the symbolic part collects the constraints satisfied by the concrete values to build a path condition ϕ. To systematically explore the program's state space and discover new paths, the algorithm selects a decision point along the current trace, negates its constraint, and asks an SMT solver for a model. The solver then generates a new concrete input that satisfies this modified condition, ensuring that the next execution deviates into an unexplored branch, thus combining the efficiency of concrete execution with the coverage of symbolic reasoning.

Rewriting logic and Maude. Maude [9] specifications correspond to rewrite theories $\mathcal{R} = (\Sigma, E \cup A, R)$ in rewriting logic [21]. We assume some knowledge on rewriting logic and Maude, but recall that terms are considered modulo some equations E and structural axioms A, and non-deterministic transformations on these terms are specified by rewrite rules R. Specifications are executable and terms can be reduced to normal form with the `reduce` command, and rewritten with rules with `rewrite`. In addition to matching and rewriting, Maude supports variant-based unification and narrowing modulo for terms with variables, as a quite general form of symbolic computation. We explain other details when they are required in the paper and refer to the Maude manual [9] for further information.

Rewriting modulo SMT. Rewriting modulo SMT combines term rewriting with SMT solving to allow symbolic reasoning on top of the first-order theories supported by SMT solvers, which usually include linear arithmetic on both integers and rationals. Given a quantifier-free formula ϕ over a set of variables X in a theory $\mathcal{T}$, we say φ is satisfiable if there is a model M within $\mathcal{T}$ (i.e., an assignment of constants to the variables in X compatible with all the axioms of $\mathcal{T}$) that satisfies φ. For its integration into term rewriting, we distinguish a *built-in subsignature* $\Sigma_0 \subseteq \Sigma$ in the order-sorted signature Σ, where $T_{\Sigma,s_0}(X_0) = T_{\Sigma_0,s_0}(X_0)$ for every sort $s_0 \in \Sigma_0$ and every set of built-in variables X_0. This means that all subterms of any built-in term are built-in, so that they can be interpreted in the associated theory $\mathcal{T}$ of the SMT solver. In this context, the system state is represented by *constrained terms* of the form (t, ϕ), where t is a term in $T_\Sigma(X)$ and ϕ is a Boolean formula over Σ_0 representing constraints on the variables in t. A rewrite rule $l \to r$ if ψ in $\mathcal{R}$ can be applied to a constrained term (t, ϕ) if there exists a substitution θ such that t matches $\theta(l)$ modulo A, and the accumulated constraint $\phi \wedge \theta(\psi)$ is satisfiable in $\mathcal{T}$. The result of such a transition is the new constrained term $(\theta(r), \phi \wedge \theta(\psi))$. Thus, execution paths correspond to sequences of satisfiable constraints, enabling symbolic reachability analysis. MaudeSE [34] supports rewriting modulo SMT with this kind of rules and keeps the accumulated constraints in its internal states.

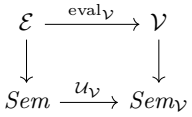


Fig. 1. General diagram of the transformations.

4 Infrastructure for the specification of semantics

In this section, we describe a simple infrastructure to specify small-step operational semantics in Maude, so that programs can be executed both concretely and symbolically as explained in Section 5. Figure 1 gives a general picture of the framework, where *Sem* represents the user-defined syntactic and semantic definition of the desired language. As usual [20], although on top of the provided infrastructure, the user should specify the following elements of the language semantics:

1. the syntax of the programming language as a protected extension of a predefined signature $\mathcal{E}$ of Boolean, integer, and rational expressions with constants, variables, and the usual arithmetic operators. These expressions should be interpreted as bare syntax that may be evaluated to different target types $\mathcal{V}$ as required;
2. the execution states of the semantics using a predefined specification of variable stores, i.e., mappings from variable names to values of the corresponding target type in $\mathcal{V}$;
3. and the semantic rules over these execution states, which may use the `eval` function provided by the framework to evaluate expressions to the target types. A typical rule has the form

   `crl` S `=>` S'[... `eval(`e'`,` s`)` ...] `if` `eval(`e`,` s`)` `=` `true` `.`

   where `eval` may be used with a Boolean expression in the condition, or in the right-hand side to set the values of the store. Unconditional rules and rules with other shapes are allowed too.

The cornerstone of the framework is the abstraction of the semantic values $\mathcal{V}$ of the basic expressions and their evaluation with `eval`. In the concrete execution setting, Boolean, integer, and rational expressions are evaluated to the predefined `Bool`, `Int`, and `Rat` sorts in Maude, so that programs can be executed as usual. However, they can also be evaluated to the sorts `Boolean`, `Integer`, and `Rat` of SMT formulae, as we do in Section 5.1; or to `BoolFVP` and `IntFVP` for narrowing in Section 5.3; among others.

Since semantic rules will need to be transformed reflectively, instead of a parameterized skeleton, the infrastructure is provided as a module `CONCRETE-FW` gathering the specifications of $\mathcal{E}$, $\mathcal{V}$, and $\mathrm{eval}_{\mathcal{V}}$ for concrete executions. The latter two are reflectively transformed for the different symbolic execution settings.

The signature $\mathcal{E}$ consists of three sorts `BExp`, `IExp`, and `RExp` for Boolean, integer, and real-valued expressions, respectively. Literals and variables of each type are wrapped with the `val` and x`v` constructors as

```
op val : Bool -> BExp [ctor] .   op bv : Qid -> BExp [ctor] .
op val : Int  -> IExp [ctor] .   op iv : Qid -> IExp [ctor] .
op val : Rat  -> RExp [ctor] .   op rv : Qid -> RExp [ctor] .
```

using the predefined Maude sort for constants, and the sort `Qid` of quoted identifiers for variable names. The signature of each type of expression is extended with the usual operators of the data type like conjunction, disjunction, order relations, arithmetic operations, etc.

The predefined variable stores come in the same three flavors, `BStore`, `IStore`, and `RStore`, and each is defined as a `Map{Qid,`s`}` from the variable names to the corresponding target sort s. Then, we combine all three in a structure `Stores`:

```
op _|_|_ : IStore RStore BStore -> Stores [ctor] .
```

Finally, an evaluation function is defined for each category of expressions, which take the generic form of

```
op eval : XExp Store -> XVal .
```

where `X`*`Exp`* is one of the expression sorts, *Store* is the store needed for evaluating `X`*`Exp`*, and *XVal* is its target value. These functions receive the variable stores, from which they obtain the value of the variables in the expression. In the concrete setting, and therefore in the predefined module, they are defined as

```
op eval : BExp Stores -> Bool .
op eval : IExp IStore -> Int .
op eval : RExp RStore -> Rat .
```

These functions are recursively defined on the syntax tree of the expressions. The `eval` function for `BExp` receives the full set of type `Stores` as Boolean expressions may contain integer or real-valued subexpressions. Those for `IExp` and `RExp` only receive the store they need, since they do not contain nested subexpressions of a different type. In Section 5, we show how `eval` functions are transformed to operate with symbolic data.

In the concrete setting, the definition of `eval` is straightforward: it unwraps the literals from the `val` constructor, obtain variable values from the store, and translate operators to their identical counterparts in the target sort. Later, we will discuss that, on the translations for symbolic and concolic execution, `eval` functions translate literals from concrete values to their SMT representation.

4.1 A simple *While* language

Let us illustrate the framework with the *While* language, a simple imperative programming language with assignments, conditionals, and loops. Its semantics is given as a system module `WHILE-MAUDE` that follows the requirements mentioned in the previous section, and extends the module `CONCRETE-FW` with the infrastructure of the concrete semantics. Instructions are terms of sort `Inst` representing the basic statements of the language, and programs are sequences of instructions of sort `Prog`. They are defined with the following operators:

```
mod WHILE-MAUDE is
  protecting CONCRETE-FW .
  sorts Inst Prog .
  subsort Inst < Prog .

  op skip : -> Inst [ctor] .
  op _;_  : Prog Prog -> Prog [ctor assoc id: skip] .
  op while_do{_} : BExp Prog -> Inst [ctor] .
  op if_then{_}else{_} : BExp Prog Prog -> Inst [ctor] .
  op if_then{_}        : BExp Prog -> Inst .

  var P : Prog . var B : BExp .
  eq if B then {P} = if B then {P} else {skip} .
```

In WHILE-MAUDE, we also define the constructor of an execution state as

```
  op <_|_> : Prog Stores -> State [ctor] .
```

and two additional functions, `getStore` and `updateStore`, to obtain and update the store of a given state, that are needed for the concolic transformation in Section 5.2.

```
  op getStore    : State -> Stores .
  op updateStore : State Stores -> State .
```

Semantic rules are represented by unconditional and conditional rewrite rules with conditions of the form $t = \text{true}$, where t is a Boolean term or the result of some evaluation function. For instance, the rules for the Boolean assignment, the conditional, and the `while` loop are the following:

```
  vars P P1 P2 : Prog . var STR : Stores .
  var  Q       : Qid  . var B   : BExp   .
   rl [assign-B] : < bv(Q) := B ; P | STR >
                => < P | insert(Q, eval(B, STR), STR) > .

  crl [if-then] : < if B then {P1} else {P2} ; P | STR >
               => < P1 ; P | STR > if eval(B, STR) = true .

  crl [if-else] : < if B then {P1} else {P2} ; P | STR >
               => < P2 ; P | STR >
               if not eval(B, STR) = true .

  crl [while-loop] : < while B do { P1 } ; P | STR >
                  => < P1 while B do { P1 } ; P | STR >
                  if eval(B, STR) = true .

  crl [while-exit] : < while B do { P1 } ; P | STR >
                  => < P | STR >
                  if not eval(B, STR) = true .
  *** [...]
endm
```

Conditional rules may also include sort-membership t `:` s, matching t `:=` t', or rewrite t `=>` t' conditions, which are left unchanged by the translations. Hence, the `eval` functions should not occur on them.

4.2 Concrete program execution

Semantics specified as explained in the previous sections are already in the concrete setting and can be directly executed with the `rewrite` command, traced step by step with the `search` command, or even model checked with concrete inputs. Of course, the specification should satisfy the usual admissibility requirements [9] like confluence and termination of equations, coherence of rules, etc.

For example, using the *While* language in Section 4.1, consider we want to execute the following program P

```
if iv('x) < val(0) then { iv('y) := val(0); }
else { iv('y) := val(2) * iv('x); }
```

when `x` takes value 1. Using the state constructor, the initial state becomes $S =$ `<` P `| 'x |->I 1 | empty | empty >` and the final state of the program can be computed with `rewrite`:

```
Maude> rewrite S .
result State: < nil | 'x |->I 1, 'y |->I 2 | empty | empty >
```

More interesting execution examples are discussed in Section 6.

5 Translations for symbolic analysis

In order to make any semantics specified as explained in the previous section amenable for symbolic execution, we need to transform them so that computation is applied on symbolic semantic values instead of concrete ones. Different types of analyses and techniques require related but different transformations. This section presents three transformations (1) for direct symbolic execution by rewriting modulo SMT with the MaudeSE tool, (2) for concolic testing, and (3) for direct symbolic execution using narrowing. All these program transformations, but the last one, are reflectively implemented in Maude.

5.1 Symbolic execution with SMT

In Section 4, we explained that our framework works on a user-defined semantics *Sem* extending a signature $\mathcal{E}$ of logical and numerical expressions, and using some predefined variable stores. In the concrete execution setting, the semantic values $\mathcal{V}$ (i.e., the result of the expression evaluation functions, and the values bound to variables in the store) are the concrete predefined Maude types `Bool`, `Int`, and `Rat`. In this section, we present the SMT-based symbolic setting, where the semantic values are SMT formulae on the Maude SMT sorts `Boolean`, `Integer`, and `Real`, defined in the `smt.maude` file of the official distribution. For executing

the programs, standard rewriting is replaced by rewriting modulo SMT [26], as implemented by MaudeSE [34]. Hence, the initial values of the program variables or arguments can be symbolic expressions with variables instead of simple constants.

Once the variable stores keep symbolic values and the evaluation functions yield them, symbolic semantic rules are obtained by an almost identical transformation

```
crl S => S' if eval evalS(e, s) = (true).Bool (true).Boolean .
```

where `eval` is replaced by the symbolic evaluation function `evalS`, `true` is now of `Boolean` type, and S' may include evaluations of the `evalS` function to be saved in the variable store. In general, these rules may include other condition fragments, and `eval` may be nested inside expressions. These rules are ready for rewriting modulo SMT, as explained in Section 3, with MaudeSE.

As explained in Section 4, the user-defined semantics is already in the concrete setting, and we translate it to a different setting using a reflective module transformation. This transformation consists of (1) including the predefined SMT modules and some useful extensions for converting the literals in the expressions to SMT constants, (2) replacing variable stores to concrete values with variable stores to symbolic values, (3) adapting the signature of the `eval` functions to yield SMT types instead of the standard ones, use symbolic stores, and perform the conversion of literals. For convenience, we use different names for sorts like `BStoreS`, `IStoreS`, and `RStoreS`, and evaluation functions, since symbolic execution will have to coexist with concrete execution in Section 5.2. The signature of the transformed evaluation functions is

```
op evalS : BExp Stores -> Boolean .
op evalS : IExp IStoreS -> Integer .
op evalS : RExp RStoreS -> Real .
```

and variable stores are now `Map{Qid,Boolean}` instead of `Map{Qid,Bool}`, and so on.

We can run programs under the symbolic semantics using the `metaSmtSearch` function from MaudeSE, which is the metalevel counterpart of its `smt-search` command. It searches through all SMT-rewriting paths, like the `search` command does over the standard rewriting paths. While MaudeSE provides significant advantages with respect to the built-in SMT support, like folding and extensibility, we can alternatively implement rewriting modulo SMT in Maude itself. This can be achieved by extending the state with an accumulated path constraint

```
op _{_} : State Boolean -> SEState [ctor] .
```

that is extended by the semantic rules as MaudeSE does under the hood:

```
crl S {C} => S' {C'}
  if C' := C and evalS(e, STR)
  /\ metaCheck(['REAL-INTEGER], upTerm(C')) .
```

where `metaCheck` is used to check whether the constraints are satisfiable using the SMT solver on the real and integer arithmetic theory in the predefined `REAL-INTEGER` module. Operator `op [_] : Qid → Module` generates a new module that imports the target one efficiently. Moreover, `metaCheck` takes the metarepresentation of the SMT formula representing the constraints, and we use the `upTerm` meta-level function to obtain it from its object level term. As an example, the rule that takes the positive branch of a conditional will extend the constraint with the evaluation of its condition and will be applied only when this extended constraint is satisfiable, i.e., when taking the positive branch is possible. This explicit translation will be used for concolic testing in Section 5.2.

We can symbolically execute program P from Section 4.2 with a Maude search over the transformed module. After one step, we have two states, one for each branch of the conditional.

```
search < P | ('x |->Is x:Integer | empty | empty) > {true}
    =>1 < P':Program | STR:Stores > {CST:Boolean} .

Solution 1 (state 1)
STR:Stores --> 'x |->Is x:Integer | empty | empty
CST:Boolean --> true and x:Integer < 0
P':Program --> iv('y) := val(0) ;

Solution 2 (state 2)
STR:Stores --> 'x |->Is x:Integer | empty | empty
CST:Boolean --> true and not x:Integer < 0
P':Program --> iv('y) := val(2) * iv('x) ;
```

Once the transformed conditional rule is applied, each state has accumulated the corresponding path constraint, as expected.

In order for symbolic execution to be useful, it must soundly and/or completely simulate concrete executions. Let Σ_0 be the built-in subsignature of the SMT theory of linear integer and real arithmetic in Maude; recall from Section 3 that X_0 denotes the set of built-in variables; $\mathcal{L}$ the subsignature of standard Boolean, integer, and rational constants in Maude; $\mathrm{Var}(t)$ denote the set of variables in t; and $\mathcal{U}$ the symbolic transformation (overloaded for modules, terms, rules, etc.). The implementation details on $\mathcal{U}$ can be found in the code repository.

Lemma 1. *For any concrete store $\sigma_c : V \to \mathcal{L}$, symbolic store $\sigma_s : V \to T_{\Sigma_0}(X_0)$, and expression $e \in T_{\mathcal{E}}(V)$, $\mathrm{Var}(\mathrm{eval}_s(e, \sigma_s)) \subseteq \bigcup_{x \in V} \mathrm{Var}(\sigma_s(x))$ and for every $\theta : X_0 \to T_{\Sigma_0}(\emptyset)$ such that $\mathcal{U}(\sigma_c(x)) =_{\mathcal{T}} \theta(\sigma_s(x))$ for all $x \in V$, $\mathcal{U}(\mathrm{eval}_c(e, \sigma_c)) =_{\mathcal{T}} \mathrm{eval}_s(e, \sigma_s)$.*

Definition 1. *A concrete state c and a symbolic state (s, ϕ) are* paired *if c and s only differ in their stores, respectively, σ_c and σ_s, $\mathrm{dom}(\sigma_c) = \mathrm{dom}(\sigma_s)$, and there is a substitution $\theta : X_0 \to T_{\Sigma_0}(\emptyset)$ such that $\mathcal{T} \vDash \theta(\phi)$ and $\mathcal{U}(\sigma_c(x)) =_{\mathcal{T}} \theta(\sigma_s(x))$ for all $x \in \mathrm{dom}(\sigma_c)$.*

Notice that (s, ϕ) is satisfiable iff it is paired with some concrete state.

Proposition 1. *1. For any concrete states c and c' such that $c \to_R c'$, and any symbolic state (s, ϕ) paired with c, there is a symbolic state (s', ϕ') paired with c' such that $(s, \phi) \to_{\mathcal{U}(R)} (s', \phi')$.*
2. For any symbolic states (s, ϕ) and (s', ϕ') such that $(s, \phi) \to_{\mathcal{U}(R)} (s', \phi')$, and any concrete state c' paired with (s', ϕ'), there is a concrete state c paired with (s, ϕ) such that $c \to_R c'$.

Corollary 1. *Symbolic execution on $\mathcal{U}(Sem)$ is sound and complete, i.e.,*

- *For any symbolic run $(s_1, \phi_1) \to_{\mathcal{U}(R)} \cdots \to_{\mathcal{U}(R)} (s_n, \phi_n)$ where ϕ_n is satisfiable, there is a concrete run $c_1 \to \cdots \to c_n$ where c_k is paired with (s_k, ϕ_k) for all $1 \leq k \leq n$.*
- *For any concrete run $c_1 \to \cdots \to c_n$, there is a symbolic run $(s_1, \phi_1) \to_{\mathcal{U}(R)} \cdots \to_{\mathcal{U}(R)} (s_n, \phi_n)$ where c_k is paired with (s_k, ϕ_k) for all $1 \leq k \leq n$.*

5.2 Concolic testing

Symbolic execution suffers from the path explosion problem, especially when loops or recursion are involved, and cannot be easily applied to complex programs. Concolic testing addresses this problem by combining repeated symbolic and concrete executions, as explained in Section 3. Moreover, this procedure requires a custom interaction with the SMT solver that cannot be achieved with the `smt-search` command of MaudeSE. Hence, our implementation of concolic testing deals with rewriting modulo SMT in Maude itself as explained at the end of the previous section. Our execution states are now of the form

```
op [_][_{_}][_] : State State Boolean State
                  -> ConcolicState [ctor].
```

where *State* is the state sort defined in the semantics. The first and second arguments are respectively a concrete and symbolic execution state with the accumulated path constraints inside braces. These arguments will be paired (as in Definition 1) during concolic executions. Finally, the fourth argument contains a copy of the initial symbolic state to be used when the program is restarted.

Unconditional `rl` S `=>` S' and conditional `crl` S `=>` S' `if` C rules in the user specification are translated to rules of the form

```
vars CST CST' : Boolean . var SMSInit : State .
 rl [S][U(S) {CST}][SMSInit] => [S'][U(S') {CST}][SMSInit] .
crl [S][U(S) {CST}][SMSInit] => [S'][U(S') {CST'}][SMSInit]
 if C /\ CST' := CST and U(C) .
```

where conditional rules check the concrete condition C and accumulate its symbolic translation to the path constraint. Notice that we do not have to check the satisfiability of `CST'` because the symbolic state is paired with the concrete one. For example, the rules for the Boolean assignment and the positive branch of conditional of the *While* language become

```
rl [assign-B] : [< bv(Q) := B ; P | STR >]
   [< bv(Q) := B ; P | STRS > {CST}][SMSInit]
=> [< P | insert(Q, eval(B, STR), STR) >]
   [< P | insert(Q, evalS(B, STRS), STRS) {CST}][SMSInit] .

crl [if-then] : [< if B then {P1} else {P2} ; P | STR >]
    [< if B then {P1} else {P2} ; P | STRS > {CST}][SMSInit]
 => [< P1 P | STR >][< P1 P | STRS > {CST'}][SMSInit]
 if eval(B, STR) = true /\ CST' := CST and evalS(B, STRS) .
```

where actions are independently replicated in both states.

While these new rules on the extended states allow running paired concrete and symbolic executions, it remains to implement the control logic of the concolic testing. Given an initial concrete state S, an initial symbolic store s_0 on the inputs of the program, and an initial constraint ϕ_0, the variant of concolic testing we consider proceeds as follows:

1. Generate concrete values for the inputs of the program using s_0 and ϕ_0, and build an initial concrete variable store s with them. If the stores in S already contain initial (concrete) assignments, they are kept, and the store is completed with assignments satisfying ϕ_0 for the remaining variables. The SMT solver can be used to generate a model for ϕ_0 as we will explain later in this section.
2. Build an initial concolic state `[`S'`][`O $\{\phi_0\}$`][`O`]` where $S' =$ `updateStore(`S`,` s`)` and $O =$ `updateStore(`S`,` s_0`)` using the user-defined `updateStore` function.
3. Run the concolic state with the transformed rules until no more rewrites are possible or a depth bound is reached. Moreover, when a conditional semantic rule is encountered, if its concrete condition is not satisfied, check whether the path constraint extended with the symbolic condition of the rule is satisfied. If so, obtain a model for that constraint ϕ, build an initial concrete variable store for it as in (1) (with ϕ instead of ϕ_0), and restart the execution from the beginning. The initial symbolic state is recovered from the last argument of the concolic state, and the corresponding concrete state is generated from it.

Notice that the second part of the third step spawns a new execution path, but the current execution can still continue with other semantic rules that succeed in the concolic state. Since every feasible road is taken in every execution cross-road, the concolic procedure is expected to test a wide and significant range of behaviors. Then, properties can be checked on the execution paths or the visited states.

The procedure above can be implemented as a standard Maude search on the transformed module if we add some additional rules. For each conditional semantic rule `crl` S `=>` S' `if` C, we generate a second transformed rewrite rule

```
crl [S][U(S) {CST}][SMSInit]
 => [S'][SMSInit {CST'}][SMSInit]
```

```
if not C /\ CST' := CST and not U(C)
/\ metaCheck(['REAL-INTEGER], upTerm(CST'))
/\ S' := updateStore(SMSInit,
                     initSTR(getStore(SMSInit), CST')) .
```

where `S'` is the new initial concrete state with the inputs obtained from the SMT solver as a model of `CST'`. For example, in the positive conditional rule, `if-then`, S would be `< if B then {P1} else {P2} ; P | STR >`, $\mathcal{U}(S)$ would be the same but with `STRS` instead of `STR`, C would be `eval(B, STR)`, and $\mathcal{U}(C)$ would be `evalS(B, STRS)`. Observe that we are checking that `CST'` is satisfiable with the `metaCheck` metalevel function of the official distribution. Moreover, `initSTR` needs to obtain a model for `CST'` in order to initialize the concrete variables of `S'`, but the Maude interface to the SMT solver only returns a Boolean answer. We have solved this problem by implementing a custom hook that communicates directly to the Z3 [24] SMT solver and returns a substitution:

```
op get-SMTassignment : Boolean -> AssignmentSMT
    [special (id-hook SpecialHubSymbol)] .
```

where `AssignmentSMT` is a comma-separated list of assignments `_<--_` whose arguments are either `Boolean`, `Integer`, or `Real`. This hook is implemented in Python using the Maude Python library [28]. Notice that the path constraint `CST'` is checked twice by an SMT in the transformed rule, first by the `metaCheck` function and then by `get-SMTassignment`, if it is satisfiable. Even though the invocation of `metaCheck` is redundant, it appears to reduce the execution time in the examples in Section 6. This might be caused by the more direct and efficient integration of `metaCheck` in Maude with respect to the Python-based hook, or the additional cost of generating a model.

In summary, the user-defined semantics *Sem* are translated to a version amenable for concolic testing Conc(*Sem*) by (1) including the infrastructure for both concrete and symbolic execution of the previous translations, with bidirectional conversions of constants between regular and SMT types, (2) duplicating the `eval` functions and the variable stores, and transforming the definition of symbolic copy as in Section 5.1, (3) defining a concolic state from the original state definition, (4) translating the original rules to advance the concolic execution state, (5) adding alternative rules for restarting the execution with different concrete inputs when a condition semantic rule cannot be applied, and (6) implementing a hook and some auxiliary definitions (both in Maude and Python) to obtain models from SMT formulae, as required by the method.

Since Conc(*Sem*) is produced at the metalevel, we are forced to interact with it from the metalevel too. Moreover, since it uses a Python-implemented hook, it needs to be run through the Maude Python library. In order to overcome these problems, which may appear when using other reflective transformations, we have developed `maude-shell` [29], a prototype of an extended Maude interpreter inspired in Full Maude [8], but implemented in Python. This tool allows loading Python hooks and selecting metamodules with the `select` command to execute regular commands on them, although not all commands are currently supported.

For instance, to check whether a property P is satisfied by concolic testing, we run

```
$ maude-shell maudeSMTHook.py \
              while.maude semantics-analysis-tr.maude
Maude> select concTr(upModule('WHILE-MAUDE, true), 'State)
Maude> search initial =>* CS s.t. not P .
```

and check there are no solutions. As claimed before, we have reduced concolic testing to a Maude search, where only the hook `get-SMTassignment` is implemented outside Maude. This allows using all command and features of Maude on the transformed model, like bounding the depth of the search, inspecting the rewrite graph, etc.

5.3 Symbolic execution via narrowing

In addition to SMT, Maude supports symbolic computation through variant-based equational unification and narrowing on user-defined theories satisfying the *finite variant property* [11]. This class of theories is more general than the repertory available in SMT solvers, although the general procedure would not be as efficient as the specialized decision procedures of these tools. We show here, as a proof of concept, how to adapt our framework for symbolic reasoning with narrowing on specifications using as semantic values the sorts `BoolFVP` of the Booleans and `IntFVP` of the Presburger integer arithmetic, which we have taken or extended from [15]. Both satisfy the finite variant property. As in previous transformations, we modify `eval` to produce terms in the `BoolFVP` and `IntFVP` sorts, assuming that rationals and unsupported operators (like product) are absent from the programs. However, since narrowing only works on top and for unconditional rules in Maude, the translation of semantic rules is more involved and produces for every conditional rule `crl` S `=>` S' `if eval(`e`, STR) = `b two narrowing-enabled rules using a temporary extended state

```
rl S => < S | eval(e, STR) > [narrowing] .
rl < S | b' > => S' [narrowing] .
```

where b' is either `tt` or `ff`, the Boolean constants in `BoolFVP`, if b was respectively `true` or `false`. This way, the Boolean formula that results from `eval(`e`, STR)` is unified against `tt` or `ff` modulo the variant equations in the specification and the state is narrowed accordingly. Then, one can symbolically execute programs using narrowing with the `vu-narrow` command, as we show in Section 6.1.

6 Case studies

In this section, we illustrate the proposed transformation and its associated techniques with some simplified examples. Although the examples here are defined with the *While* language specified in Section 4.1, tests with a simple functional recursive language can be found in the repository [23].

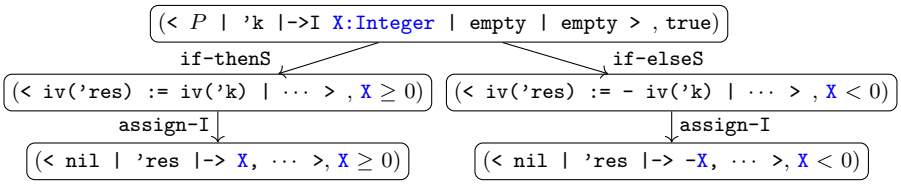


Fig. 2. Graph of a symbolic execution of P_1.

6.1 Checking functional properties

Consider a program P_1 in the *While* language of Section 4.1 that calculates the absolute value of an input variable `k` as

```
if iv('k) ≥ val(0) then { iv('res) := iv('k); }
else { iv('res) := - iv('k); }
```

In the concrete setting, we can fix the initial value of `k` and execute the program with the `rewrite` command or obtain an execution trace with `search`. For instance, if `k` takes value 0, and assuming arbitrary real-valued and Boolean stores `RS` and `BS`, the execution will follow this path

```
Maude> search < P1 | 'k |->I 0 | RS | BS > =>! S:State .
...
Maude> show search graph .
state 0, State: < P1 | 'k |->I 0 | RS | BS >
arc 0 ===> state 1 crl ... [if-then]
state 1: < iv('res) := iv('k) ; | 'k |->I 0 | RS | BS >
arc 0 ===> state 2 rl ... [assign-I]
state 2: < nil | ('k |->I 0, 'res |->I 0) | RS | BS >
```

For this small example, its behavior can be easily characterized by the symbolic execution graph in Figure 2, where the value of `res` can be either `X` or `-X` at the end of the program, where `X` is the initial value of `k`. This is the SMT search graph in the transformed module $\mathcal{U}(Sem)$ presented in Section 5.1. The `semantics-analysis-ext.py` script available in the companion repository [23] facilitates loading a given semantics, transforming it for the desired analysis, and executing commands on it. For instance, we can check that `res` is always positive at the end of P_1 with a command equivalent to

```
> smt-search < P1 | 'k |->I X:Integer | RS | BS >
          =>! < nil | 'res |->I Y:Integer, IS | RS | BS >
          s.t. not (Y:Integer ≥ 0) .
```

that will have no solution, and it takes around 180ms.

Alternatively, we can check that the final value of `res` is always positive by using narrowing with the procedure in Section 5.3. Unlike the other transformations, this is not implemented in Maude, so we have translated the module by hand. Since the property holds, if we look for a final state where `res` is bound to a negative value (represented by the pattern `- Y:NzNatFVP`) from an initial state where `k` takes an arbitrary integer value `X:IntFVP`, we must find no solution.

```
Maude> vu-narrow {delay, filter} < P1 | 'k |->I X:IntFVP >
           =>* < skip | 'res |->I - Y:NzNatFVP, IS > .
No solution.
rewrites: 147 in 3ms cpu (3ms real) (49000 rewrites/second)
```

Moreover, if we replace `- Y:NzNatFVP` with `Y:NzNatFVP` in the pattern, i.e., a negative result with a positive one, the same query provides two solutions where `X` respectively equals `Y` and `- Y`, as expected. In other words, we have calculated the preimages of the absolute value under different constraints, and this can also be applied to other functions.

6.2 Finding bugs by symbolic execution

We can use symbolic execution to find bugs or undesired situations in programs, for example, a division by 0. Consider a program P_2 where the denominator of a division may be 0 at some point in execution. P_2 is defined as

```
rv('y) := val(20); rv('i) := val(5);
while rv('k) < rv('y) do {
  if rv('k) > val(10) then { rv('z) := rv('y) / rv('i); }
  rv('k) := rv('k) + val(1);
  rv('i) := rv('i) - val(1); }
```

where `k` is a symbolic rational variable. We can execute this program symbolically and look for a state matching the pattern `< RV:RVar := RE:RExp / rv('i); P:Prog | IS:IStoreS | RS:RStoreS, 'i |->Rs 0/1 | BS:BStoreS >` where the denominator variable is zero. The result of the SMT-search, which took around twice the time as for P_1, indicates that the state number 25 matches the pattern, and it is reachable when $10 < \mathtt{k} < 20$ after 5 iterations of the loop.

6.3 Finding bugs in more complex programs by concolic testing

As we have already mentioned, direct symbolic execution may become unfeasible when the program becomes more complex due to path explosion. In a program like the following

```
iv('i) := val(5);
while iv('k) < iv('y) do {
  if iv('k) > val(10) then {
    iv('x) := iv('k) * iv('y);
    iv('a) := iv('x) - iv('i);
    iv('z) := iv('y) / iv('a); }
  iv('k) := iv('k) + val(1); iv('i) := iv('i) * val(2); }
```

where `k` and `y` are symbolic variables, the conditional inside the loop forks the path in each iteration, making the symbolic graph grow exponentially. With concolic execution, we can mitigate this problem, as it executes the program concretely several times. Similarly to the previous examples, using the transformed module, we can search over the concolic state graph for a state where

we are dividing by some variable with value 0 at that execution point. This can be done with the extended Maude interpreter presented in Section 5.2 and the standard `search` command. Maude finds a solution at state 3147 after around 1.3 seconds. We can retrieve the sequence of rules applied with the `show path` command, and check that the bug was reached after 7 iterations of the loop. The resulting store shows that the counterexample corresponds to the assignment: $k \mapsto 9$, $y \mapsto 20$.

7 Conclusions

In this paper, we have presented a generic framework for symbolic and concolic execution of programs parameterized on the semantics of the language in Maude. This work is built on top of the existing support for symbolic computation in Maude and MaudeSE, using mainly rewriting modulo SMT but also narrowing. We have implemented metalevel translations from a user-defined semantics with some restrictions to modules ready for symbolic and concolic execution. For the latter, we have also implemented a concolic execution algorithm with the transformed semantic rules, aided by a custom Maude hook in Python that generate initial values for concrete execution of the algorithm via SMT. In addition, we provide auxiliary Python scripts to simplify using the framework, and a general prototype inspired in Full Maude to run object-level commands on metalevel modules with potential Python hooks [29]. These translations are generic for any semantic specification, and have been tested with a simple imperative language and a functional recursive language. Examples of both can be found in the code repository [23]. Finally, we run a small benchmark the different symbolic execution approaches.

As future work, we consider incorporating narrowing and narrowing modulo SMT as a way of dealing with more complex data types and custom data structures, as well as trying symbolic model checking to verify more complex properties. We also want to apply the framework on semantics for more realistic programming languages, like WebAssembly [25].

Acknowledgments. We would like to thank Narciso Martí-Oliet for his comments on this manuscript. This work is partially supported by MCIU/AEI/10.13039/50110-0011033/FEDER,UE through project ProCode 10 (PID2023-149943OB-I00). Rubén Rubio is partially supported by Comunidad de Madrid/UCM through project COMFIA (PR17/24-31913).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the contents of this article.

References

1. Arias, J., Bae, K., Olarte, C., Ölveczky, P.C., Petrucci, L.: A rewriting-logic-with-SMT-based formal analysis and parameter synthesis framework for parametric time Petri nets. Fundam. Informaticae **192**(3-4), 261–312 (2024). https://doi.org/10.3233/FI-242195

2. Arias, J., Bae, K., Olarte, C., Ölveczky, P.C., Petrucci, L., Rømming, F.: Symbolic analysis and parameter synthesis for networks of parametric timed automata with global variables using Maude and SMT solving. Sci. Comput. Program. **233**, 103074 (2024). https://doi.org/10.1016/j.scico.2023.103074
3. Arusoaie, A.: A Generic Framework for Symbolic Execution: Theory and Applications. Ph.D. thesis, Alexandru Ioan Cuza University, Iaşi, Romania (2014), https://tel.archives-ouvertes.fr/tel-01094765
4. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3), 50:1–50:39 (2018). https://doi.org/10.1145/3182657
5. de Boer, F.S., Bonsangue, M.M.: Symbolic execution formally explained. Formal Aspects Comput. **33**(4-5), 617–636 (2021). https://doi.org/10.1007/S00165-020-00527-Y, https://doi.org/10.1007/s00165-020-00527-y
6. Cadar, C., Nowack, M.: KLEE symbolic execution engine in 2019. Int. J. Softw. Tools Technol. Transf. **23**(6), 867–870 (2021). https://doi.org/10.1007/s10009-020-00570-3
7. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: NFM 2011. LNCS, vol. 6617, pp. 459–465. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_33
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: Full Maude: Extending Core Maude. In: All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, LNCS, vol. 4350, pp. 559–597. Springer (2007). https://doi.org/10.1007/978-3-540-71999-1_18
9. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.: Maude Manual (version 3.5.1) (2025), https://maude.lcc.uma.es/maude-manual/
10. Do, C.M., Ogata, K.: Symbolic model checking quantum circuits in Maude. PeerJ Comput. Sci. **10**, e2098 (2024). https://doi.org/10.7717/peerj-cs.2098
11. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.L.: Equational unification and matching, and symbolic reachability analysis in Maude 3.2 (system description). In: IJCAR 2022. LNCS, vol. 13385, pp. 529–540. Springer (2022). https://doi.org/10.1007/978-3-031-10769-6_31
12. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.L.: Programming and symbolic computation in Maude. J. Log. Algebraic Methods Program. **110** (2020). https://doi.org/10.1016/j.jlamp.2019.100497
13. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker and its implementation. In: SPIN 2003. LNCS, vol. 2648, pp. 230–234. Springer (2003). https://doi.org/10.1007/3-540-44829-2_16
14. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. In: Field, J., Hicks, M. (eds.) POPL 2012. pp. 533–544. ACM (2012). https://doi.org/10.1145/2103656.2103719
15. Escobar, S.: Extensions of logic programming in Maude (2023), https://logicprogramming.org/2023/02/extensions-of-logic-programming-in-maude/
16. Escobar, S., Meadows, C., Meseguer, J.: Maude-NPA: Cryptographic protocol analysis modulo equational properties. In: FOSAD 2007/2008/2009. LNCS, vol. 5705, pp. 1–50. Springer (2007). https://doi.org/10.1007/978-3-642-03829-7_1
17. Farzan, A., Chen, F., Meseguer, J., Rosu, G.: Formal analysis of Java programs in JavaFAN. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 501–505. Springer (2004). https://doi.org/10.1007/978-3-540-27813-9_46

18. Kneuper, R.: Symbolic execution: A semantic approach. Sci. Comput. Program. **16**(3), 207–249 (1991). https://doi.org/10.1016/0167-6423(91)90008-L
19. Lucanu, D., Rusu, V., Arusoaie, A.: A generic framework for symbolic execution: A coinductive approach. J. Symb. Comput. **80**, 125–163 (2017). https://doi.org/10.1016/j.jsc.2016.07.012
20. Martí-Oliet, N., Meseguer, J.: Rewriting logic as a logical and semantic framework. In: Meseguer, J. (ed.) RWLW 1996. ENTCS, vol. 4, pp. 190–225. Elsevier (1996). https://doi.org/10.1016/S1571-0661(04)00040-4
21. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992). https://doi.org/10.1016/0304-3975(92)90182-F
22. Meseguer, J.: Symbolic computation in Maude: Some tapas. In: Fernández, M. (ed.) LOPSTR 2020. LNCS, vol. 12561, pp. 3–36. Springer (2020). https://doi.org/10.1007/978-3-030-68446-4_1
23. Morales, R.: Symbolic execution in Maude (2026), https://github.com/rafamor-exe/Symbolic_Execution_in_Maude
24. de Moura, L.M., Björner, N.S.: Z3: an efficient SMT solver. In: TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
25. Palacios, R.M.: Semantics of WebAssembly in Maude. Master's thesis, Facultad de Informática. Universidad Complutense de Madrid (2025), https://hdl.handle.net/20.500.14352/125016
26. Rocha, C., Meseguer, J., Muñoz, C.A.: Rewriting modulo SMT and open system analysis. J. Log. Algebraic Methods Program. **86**(1), 269–297 (2017). https://doi.org/10.1016/j.jlamp.2016.10.001
27. Rosu, G., Serbanuta, T.: An overview of the K semantic framework. J. Log. Algebraic Methods Program. **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
28. Rubio, R.: Maude as a library: An efficient all-purpose programming interface. In: WRLA 2022. LNCS, vol. 13252, pp. 274–294. Springer (2022). https://doi.org/10.1007/978-3-031-12441-9_14
29. Rubio, R.: Maude Shell: a extended Maude interpreter prototype (2026), https://github.com/fadoss/maude-shell
30. Steinhöfel, D.: Symbolic Execution: Foundations, Techniques, Applications, and Future Perspectives, pp. 446–480. Springer (2022). https://doi.org/10.1007/978-3-031-08166-8_22
31. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. J. Log. Algebraic Methods Program. **67**(1-2), 226–293 (2006). https://doi.org/10.1016/j.jlap.2005.09.008
32. Voogd, E., Johnsen, E.B., Kløvstad, Å.A.A., Rot, J., Silva, A.: Correct and complete symbolic execution for free. In: IFM 2024. LNCS, vol. 15234, pp. 237–255. Springer (2024). https://doi.org/10.1007/978-3-031-76554-4_13
33. Xiao, L., Zhu, H.: UTP semantics for the MCA ARMv8 architecture. J. Syst. Archit. **125**, 102438 (2022). https://doi.org/10.1016/j.sysarc.2022.102438
34. Yu, G., Bae, K.: A flexible framework for integrating Maude and SMT solvers using Python. In: WRLA 2024. LNCS, vol. 14953, pp. 179–192. Springer (2024). https://doi.org/10.1007/978-3-031-65941-6_10

On the translation of Maude programs to modern imperative languages

Rubén Rubio, Beatriz Alcaide García, and Adrián Riesco

Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain
{rubenrub, balcaide, ariesco}@ucm.es

Abstract. Maude is a high-level specification and programming language based on rewriting logic. Although expressiveness is its primary design goal, Maude achieves a high level of performance. Nevertheless, it is not intended to compete with lower-level imperative languages in terms of raw performance. Moreover, such languages offer greater interoperability with existing software systems. For these reasons, it is worth exploring the mapping of Maude specifications to other programming languages. In this paper, we present a compiler from a subset of Maude to Rust, C++, Python, and Dafny, with particular emphasis on their support for pattern matching and algebraic data types. We evaluate the usefulness of the proposed approach through several simple yet illustrative examples, and assess it by means of differential testing.

1 Introduction

Maude [5] is a high-level specification and programming language based on rewriting logic [20]. Despite its expressivity, its interpreter is also a high-performance rewriting engine. Indeed, in the latest Rewrite Engine Competition [9], Maude comes second just between Haskell and the compiled version of OCaml. In most verification and programming applications of Maude, Maude programs are as efficient as the intended model can be or efficient enough for its purpose.

However, generating semantically-equivalent code in a more conventional programming language may be worth in some situations. One of them is the integration of Maude-based components into other software systems. Maude's built-in support for external objects (sockets, processes, files, etc.) and its Python library [27] are very useful for this purpose, but still need a Maude interpreter process behind the scenes. In embedded or low-resource systems, like robots [18], generating a lightweight standalone program might be required. In this direction, an important reason for compilation is performance. However, compiling even subsets of such a general language like Maude is notably hard and does not guarantee a better performance [7,9]. Compilation has been widely studied and put in practice in the context of declarative programming languages, where different analyses and optimization techniques have been studied. Some intermediate languages, like Prolog's Warren Abstract Machine, have been developed as a compilation target or as a step towards C or machine code. Other rewriting-based languages include compilers like ASF+SDF [4] and ELAN [21].

There have also been abandoned efforts to build a compiler for Maude itself, which are still available in the Maude codebase.

In the verification field, a potentially interesting use case for compilation is statistical model checking [1], where a model with random behavior is executed many times to estimate quantitative values by the Monte Carlo method. In this context, the absence of unquantified nondeterminism (required for the soundness of the statistical analysis) simplifies compilation efforts, which are also rewarded by reusing the same generated code in many executions. Moreover, simulations in statistical model checking are easy to distribute and parallelize, and lightweight specialized code may facilitate exploiting the resources of the host machines.

In this paper, we present an experimental compiler for a subset of Maude specifications to some imperative programming languages, namely Rust, C++, Python, and Dafny. All of these languages but C++ have some support for algebraic datatypes and pattern matching we want to leverage on. While we could have targeted a lower-level language, like LLVM, as compilation target, we wanted to use as much high-level constructs and data structures as possible to simplify the translation and approach the generated code to the original one. Our translation is implemented in two steps: first, an abstract syntax tree for the Maude code is obtained as intermediate representation. Then the tree is used to generate the imperative code. Even though the translation should preserve the semantics by construction, we use differential testing [19] to assess the equivalence of the generated code. For this purpose, we have updated the MUnit framework [26] to work in Maude 3. Some performance evaluations have been carried out with promising results.

The rest of the paper is organized as follows: Section 3 introduces some preliminaries, Section 4 describes the translation from Maude programs to other languages, and Section 5 discusses some case studies. Finally, Section 6 concludes and outlines some lines of future work. The source code of the compiler, examples, and tests are available at `github.com/fadoss/maudec`.

2 Related work

In the early years of rewriting logic, compilation of patterns and rewrite programs attracted much attention. Several abstract machines were proposed like the Reduce ELAN Machine [21], the Rewrite Rule Machine [16], and CafeOBJ's Term Rewriting Abstract Machine [24]. The ELAN language [13] counted with a compiler that produces C code and supports matching modulo theories like AC, rule rewriting, and rewrite strategies. ASF+SDF [4] was also endowed with a compiler. Compilation from a subset of Maude called Simple Maude to SIMD or MIMD machines is discussed in [15]. Maude is an interpreted language, but owes much of its performance to the careful semicompilation of its pattern matching [8]. In this paper, we do not aim to translate Maude to an abstract machine or to a low-level programming language, but to leverage on high-level programming constructs of modern imperative languages.

Soufflé [12] implements a transformation from Datalog [11], a logic programming language used for program analysis in deductive databases, into C++. The main motivation for their translation is improving the performance of the analyses while preserving a declarative interface from the user's perspective. This idea is fully aligned with the approach presented in this paper, which seeks to support rapid system specification alongside efficient execution in widely adopted programming languages.

Regarding the translation to Dafny, we have presented a translation of Maude specifications into the theorem prover Lean [23] in [30]. Dafny and Lean occupy complementary points in the verification space: Dafny emphasizes automation and scalability, while Lean emphasizes expressiveness and proof control. By supporting both, our approach allows users to balance verification effort and assurance, and to apply the most appropriate reasoning technique to each class of properties. Actually, the Maude approach is also complementary to both of them, supporting, for example, model checking and symbolic reachability analysis.

Finally, the Heterogeneous Tool Set (Hets) [22] proposes an approach based on translations, but with a very different nature and purpose. Hets focuses on translations between tools for formal analysis (including Maude), whereas our methodology targets translations between verification-oriented and high-performance languages. Moreover, since the logics supported by Hets are formalized as institutions, translations are formally defined as comorphisms. In contrast, our translations are not formally defined and instead rely on operational techniques such as differential testing. This lack of formalization stems from the nature of the target languages, which do not have a well-established mathematical foundation.

3 Preliminaries

We present here the basic notions of the languages involved in the translation.

3.1 Maude

Maude [6] is a high-performance logical framework based on rewriting logic [20], a logic of change where state transitions are modeled by means of rewrite rules. Rewriting logic is parameterized by an equational logic; in the case of Maude, this logic is membership equational logic [3] which, in addition to equations, allows one to state membership axioms characterizing the elements of a sort.

Maude functional modules [6, Chap. 4], introduced with syntax `fmod ... endfm`, are executable membership equational specifications that allow the definition of sorts (by means of keyword `sort`(`s`)); subsort relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`), where conditions have three

variants: (i) equational conditions (t `=` t'), (ii) membership conditions (t `:` s), and (iii) matching conditions (t `:=` t'). In matching conditions the term t possibly includes new variables and holds if t', once reduced by means of equations, matches t. Matching is performed modulo the equational axioms stated in operators, giving Maude specifications a high expressive power. It is worth mentioning that each connected component of sorts by the subsort relation is called a *kind*.

3.2 Some imperative programming languages

C++. C++ [31] is a general-purpose, object-oriented programming language built on top of its precursor C. C++ leverages existing C libraries while gaining additional expressive power and flexibility by means of encapsulation, inheritance, and polymorphism and keeping very good performance. For these reasons, C++ is a widely used language for high-performance applications and large-scale software development. There is no support for pattern matching in the language (although its addition has been proposed), neither for algebraic datatypes, but limited forms can be achieved using types as `std::variant`.

Python. Python [17] is also a general-purpose, object-oriented programming language known for its readability and extensive ecosystem. In recent years, it has gained popularity as the dominant language for artificial intelligence and machine learning, due to its simplicity and the availability of powerful libraries and frameworks. Pattern matching is, in this case, natively supported through structural pattern matching (`match`/`case`).

Rust. Rust [2] is a systems programming language designed to provide memory safety and concurrency without sacrificing performance. It has gained popularity for building reliable and efficient software, particularly in contexts where safety and low-level control are critical. In the case of Rust, pattern matching is a core language feature, deeply integrated via `match`, `if let`, and `while let`.

Dafny. Dafny [25] is a verification-aware programming language with native support for Hoare logic specifications and an SMT-based static program verifier. Pattern matching is supported through `match` statements and expressions.

3.3 Pattern matching support in mainstream languages

Several modern general-purpose imperative programming languages support some form of algebraic data types (ADTs) and pattern matching. Generalizing the traditional `switch` construct, for multiple case distinction in integer values, pattern matching statements and expressions are available in languages like Python and Java since 2021, C# since 2019, Scala, Rust, and Dafny, among others. However, their generality and features differ from one language to the other. Some characteristics that will be relevant when translating Maude statements are:

1. *Nesting* of patterns is supported by all languages in the previous list.

2. *Linearity* proscribes multiple occurrences of the same variable in the pattern. All programming languages mentioned above (and even functional languages like Haskell or OCaml) require their patterns to be linear. Otherwise, the substructures matched by the multiple occurrences of the same variable would need to be deeply checked for equality, which would imply a significant and somehow hidden amount of work. However, a non-linear pattern can be translated to a linear one and some equality checks.
3. *Guards* are conditions that can be attached to each branch of the matching construct to be selected. They cannot be simply replaced by nested conditional statements, because this would avoid the execution of alternative branches of the matching construct. Guards are not supported by Dafny.
4. *Bindings inside guards*, i.e., the possibility of declaring variables in the guards, is only supported by Python (and Maude). However, this only makes a difference with languages like Rust or C++, where variable assignment means potentially non-trivial copy or movement of values.
5. *Disjunction of patterns* allows assigning the same body to different patterns with the same variables, usually with the syntax p_1 `|` $\cdots$ `|` p_n. However, its semantics may differ between languages when combined with a guard. In Rust, matching with the disjunctive pattern succeeds if there is a matching subpattern that satisfies the condition, while in Python, this happens only when the first matching subpattern satisfies the condition. In Dafny, disjunctive patterns cannot bind variables.
6. *Matching on predefined data structures*, like tuples or lists, is supported with limitations by most languages. For instance, Rust supports *slice patterns* `[`p_1`,` p_2`,` r `@ ..,` p_3`]` with some element subpatterns and at most one slice capture. This is similar to sequence patterns in Python p_1`,` p_2`, *`r`,` p_3, where at most one starred element (representing a slice) is allowed. These restrictions are reasonable, since having multiple slice components will imply a search and increase the complexity of pattern matching. Dafny does not support list patterns.

Pattern matching is usually applied on instances of algebraic datatypes. Out of the languages mentioned before, only Scala, Rust, and Dafny support algebraic datatypes, often called *sum types* or *enumerations* with fields. For example, in Rust, a type that either holds a value of type `T` or nothing is declared as:

```
enum Option<T> { None, Some(T), }
```

In other languages, ADTs can be implemented as tagged unions using enumeration constants, classes, records, or tuples. In C and C++, they are often implemented with `union` and a tag of an `enum` type, but C++17 introduced them in the standard library with `std::variant`. Another key difference between languages is whether the responsibility of memory management is on the user (i.e., garbage collection, explicit allocation and deallocation of resources, etc.). This makes implementing recursive data structures easy in Python, Java, or Dafny, but not in C++ or Rust. Indeed, Rust sum types cannot be recursive unless explicit (smart) pointers are used.

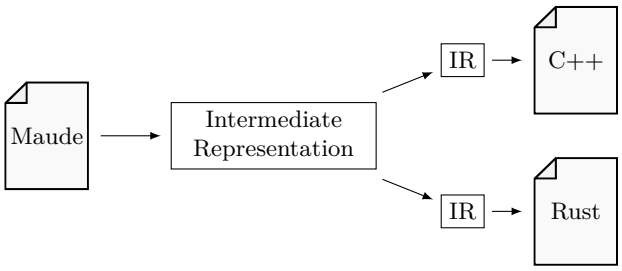


Fig. 1. Architecture of the translation.

In the implemented tool [29], whose principles are described in Section 4, Rust, Python, C++, and Dafny are supported as target languages. Moreover, for simplicity, we will use Rust as the main language for the examples in this paper.

3.4 Testing

We briefly describe the unit testing frameworks available for the languages involved in the translation. MUnit [26] is a framework for Maude specifications that we have reimplemented to work in Maude 3.[1] Although it provides assertions specific to Maude, here we are interested in `assertTrue(`t`)` (the given term t is reduced to `true`), `assertFalse(`t`)` (the term t is reduced to `false`), `assertEquals(`t`, `t'`)` (the normal forms of t and t' are the same) and `assertLeqSort(`t`, `s`)` (the normal form of t has sort s).

MUnit is inspired by Python's `unittest`,[2] so they are very similar. It also provides `assertTrue`, `assertFalse`, and `assertEqual`, with the expected meaning. In turn, Rust's unit test infrastructure [14, § 11] provides the assertions `assert!(`t`)` and `assert_eq!(`t`, `t'`)`, along with syntax to define functions as tests. Regarding Dafny, it also includes built-in support for writing test as functions annotated with the `:test` attribute and also to generate them based on the specification. Finally, for C++, we implement the tests ourselves.

4 Code generation from Maude programs

In this section, we explain the proposed code generation from a subset of Maude programs, which goes through an intermediate internal representation. Only a subset of Maude is implemented in the tool [28], so we will indicate what is supported and how the transformation can be extended beyond the implementation. Given a Maude module, the translation consists of several steps, as illustrated in Figure 1:

[1] The new version is available at https://github.com/ariesco/MUnit.
[2] https://docs.python.org/es/3/library/unittest.html

1. The Maude module is analyzed to index module elements (kinds, sorts, operators, etc.), identify constructors, locate built-in types, characterize sorts, calculate the kind-containment relation, etc. This information is a valuable base for the next steps.
2. An abstract syntax tree is built in an intermediate representation (explained in Section 4.1) from the Maude module. This process is described in Section 4.2.
3. Some transformations on the intermediate representation are applied to simplify it or adapt it to the desired target language. For instance, `match` statements can be removed for languages like C++. We explain it in Section 4.3.
4. Code in the final language is generated from the intermediate representation, as explained in Section 4.4.

The module information is obtained through the Maude Python library [27] and it is later processed in Python. One relevant relation that is calculated at this stage is the mutual dependency or containment between kinds. A kind k directly depends on another k' if there is an operator with range in k and a sort of kind k' in its domain. The non-trivial strongly-connected components of this relation are the sets of recursive or mutually-recursive kinds.

4.1 Intermediate representation

Using an intermediate representation is a common practice in compilers and language processors, not only when multiple inputs or outputs are planned, but also to apply transformations or optimizations in a more convenient setting. Our intermediate representation is designed to take advantage of the support for algebraic datatypes and pattern matching in the potential target languages, while providing alternatives to circumvent the lack of pattern matching after an internal transformation.

A program in this intermediate representation is a collection of sum type declarations, function declarations, and their bodies, internally represented as objects in some hierarchies `Node` of syntactic nodes, `Pattern` of patterns, and `AstType` of types (see Figure 2). The definitions of these elements are the expected ones: function declarations are top-level nodes with a name, a sequence of argument names and types, and a range type; expressions of type `Expr` are a particular category of nodes that yield a value (and have a type) and comprises constants, variables, function, operator, and datatype constructor calls, and conditional expressions, among others. Other syntactic nodes that are not expressions are variable declarations, conditional blocks, and return statements. Types can be simple (`NamedType`) or compound (`PointerType`, `SequenceType`, etc.), where named types of name `bool`, `int`, `nat`, `float`, and `string` are intended to match the corresponding built-in types in Maude and the target language. User-defined algebraic datatypes are also named types, but they are declared in a `SumTypeDecl` with a name and sequence of `Variant`s, which in turn are given by a name and a sequence of argument types. For instance, an algebraic datatype with two variants that contains either an integer or none is written as

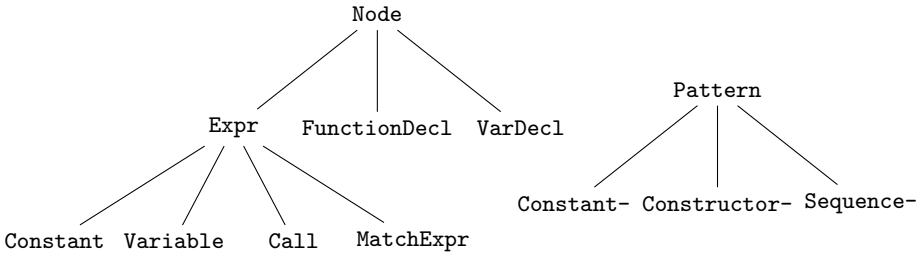


Fig. 2. Summary of the main elements of the intermediate language.

```
SumTypeDecl('IntOption', [Variant('None')],
      Variant('Some', [NamedType('int', builtin=True)])])
```

Sum types are destructured by pattern matching with the `MatchExpr`, i.e., a pattern-matching expression like those we have discussed in Section 3.3. It is given by an input expression and a sequence of branches, where each branch consists of a pattern of type `Pattern` and an optional Boolean guard of type `Expr`. There are several subclasses of patterns: `VariablePattern`, `ConstantPattern`, `ConstructorPattern` for patterns on algebraic datatypes, `OrPattern` for disjunctions of patterns, `TuplePattern`, and `SequencePattern` for patterns on lists. Most of these patterns exist in the target languages, although with some limitations, as pointed out in Section 3.3.

With these rudiments, we build the translations from Maude and to the target languages as explained in the following sections.

4.2 From Maude to the intermediate representation

The translation from a Maude module to the intermediate representation and to the target language aims to be as direct as possible, although this is limited by the lower generality of this representation and the target languages. The unit of translation is a flattened Maude module and the modular structure is not preserved. In summary, kinds are translated to types, constructor operators become data constructors for those types, equationally-defined operators are translated to functions whose body is a `match` expression, and terms are recursively translated to expressions on built-in constants, variables, operators, function, and data constructors calls. Let us describe the details element by element.

For each kind k, we declare a sum type $T(k) =$ `NamedType(`k`)` with a variant of domain $T^*([s_1]) \cdots T^*([s_n])$ for each constructor `op` f `:` $s_1 \cdots s_n$ `->` s `[ctor]` in the kind. We denote by $[s]$ the kind of the sort s, and $T^*(k') =$ `PointerType(`$T(k')$`)` if k and k' are mutually-recursive kinds or $T^*(k') = T(k')$ otherwise. For example, the Maude code

```
op mt : -> Stack [ctor] .
op push : Int Stack -> Stack [ctor] .
op empty : Stack -> Bool .
```

gives the following sum type

```
SumTypeDecl('Stack', [Variant('mt', []),
    Variant('push', [NamedType('nat', builtin=True),
                     PointerType(NamedType('Stack'))])])
```

Built-in types and operators with structural axioms are handled differently:

- The constructors of the built-in sorts `Bool`, `Nat`, `Int`, `Float`, and `String` are recognized by their attributes (since names may have changed by a renaming) to translate their instances to Boolean, integer, and string constants in the intermediate representation. If the built-in type constructors are the only one in their kinds, no sum type is defined and the type is used unwrapped. Otherwise, the built-in type is one of the variants of the sum type.
- Commutative constructors are represented as in the general case, but a function is defined to normalize them by sorting their arguments. When the operator is applied on non-ground arguments, this function is called instead of the sum type constructor.
- Iterative symbols `op` f `:` s `->` s `[iter]` are translated to a variant with signature $T^*([s])$ `NamedType('nat', builtin=True)`, where the second argument is the exponent. A normalizing function is similarly defined.
- Associative operators (with or without unity) are translated to sequences or lists. When the identity element is a nullary constant (in Maude, identity elements can be any term), it is identified with the empty sequence. If the associative constructor cannot be nested in other constructors of the kind, $T(k)$ is defined without that constructor and `SequenceType(`$T(k)$`)` is used in place of sorts containing the operator. Otherwise, a variant of that sequence type is added to the sum type. Associative operators are not fully supported in the implementation, specially when matching with unsupported patterns in the target languages.
- Associative and commutative (AC) operators can be represented as multisets in the same way associative operators are translated to sequences. However, unlike for sequences, there is no support for matching inside multisets in the considered imperative languages, so it must be implemented explicitly. Notice that efficient matching modulo AC is complicated [13,8], but some usual patterns can be more easily, yet not trivially, handled with a search. Our implementation does not support these operators yet.

Examples with commutative and associative operators can be found in Section 5.

Even though data constructors are defined at the kind level, sorts need to be taken into account for matching and sort-membership tests. Moreover, they are useful for simplification of the generated code. Explicit $t :: s$, $t : s$, and implicit sort-membership tests are usually translated to calls to `hasSort_`s predicates. These predicates are defined by a match expression with the constructor declarations and membership axioms in the sort, and potentially using the predicates for other sorts. For built-in types, specialized inline tests are generated, like $x \geq 0$ for `Nat` (if not the only maximal sort) or $x > 0$ for `NzNat`. We do not attach

sort information to the objects, which will make sort testing more efficient, so programs relying on complex sort calculations will not work efficiently.

Each operator with equations is translated to a function, whose domain and result type are obtained by translating the corresponding Maude kinds. It is defined by a single `match` expression on a tuple with the function arguments, and with a case for each equation that has that symbol on the top of its left-hand side. The case patterns are obtained by translating the left-hand side of the equations, and the case body is the translation of the right-hand-side term. Conditions (if any) are inserted into the case guard using the equality operator of the datatype and the aforementioned sort-membership functions. Matching conditions introduce variable declarations in the case body for the free variables in their left-hand sides. For example, the `empty` function declared above can be defined in Maude with

```
eq empty(mt) = true .
eq empty(push(N:Int, S:Stack)) = false .
```

This is translated to a function in the internal representation equivalent to the more easily readable Rust code

```
fn empty(s: &Stack) -> bool {
  match s { Stack::Mt => true,
            Stack::Push(_, _) => false, }
}
```

In case a subterm occurs multiple times in an equation, we use an auxiliary variable declaration to avoid recomputing the subterm multiple times (in place of the hash consing of Maude). Built-in operators (called *special* in the Maude terminology) are translated to calls that will be eventually translated to built-in operators or library function calls in the target language. This includes Boolean and arithmetic operators, trigonometric functions, etc. Structural axioms are taken into account when constructing patterns. For example, the left-hand side $f(p_1, p_2)$ where f is a commutative operator is translated to the disjunctive pattern $(p_1, p_2) \mid (p_2, p_1)$.

Support for rules in the current version of the translation is limited to deterministic rewriting, although it can be extended to deal with nondeterminism. We introduce a function for each kind that applies any possible rule on top of the given term. The definition of this function is similar to those of equationally-defined operators. For example, the following two rules describe the Collatz sequence, whose termination is only a conjecture as of today.

```
crl N => N quo 2   if N rem 2 = 0 .
crl N => 3 * N + 1 if N rem 2 = 1 .
```

They are translated to an intermediate representation that will generate the following Rust code

```
fn rlapp_Nat(term: i32) -> Option<i32> {
  match term {
    n if n % 2 == 0 => Some(n / 2),
```

```
      n if n % 2 == 1 => Some(3 * n + 1),
      _ => None,
    }
  }
```

Notice that it returns an `Option<i32>` instead of the raw value, so that we are able to recognize when rules cannot be applied. Then, we also define some functions, called `arlapp_`k, to rewrite the innermost leftmost regex within the given term. They delegate on `rlapp_`k and other `arlapp_`k' to rewrite on top and inside subterms, respectively. Rewriting conditions t `=>` t' are currently not supported, but they can be implemented by a nested search. For introducing nondeterminism, the rule-application functions can be defined as coroutines to incrementally provide solutions. In that case, a single `match` expression would exclude rewrites, since only the first matching case is activated, so patterns should be checked one by one. Rewrite strategies can then be translated to functions that invoke these rule functions.

4.3 Transformations within the intermediate representation

The intermediate representation generated from the Maude code can be further processed, simplified, or optimized. For instance, match expressions on tuples (which are used for the argument of a function) are simplified by removing those variables in which there is no case distinction. In addition to the generic transformations, the syntax tree may be also adapted to a particular target language or family of them. Some of these transformations are:

- The *dematch* transformation completely removes the `match` expressions and replaces them by equivalent code with conditionals that explictly checks in which variant the object is and extracts the information from it, using the `VariantCheck` and `VariantAccess` expressions of the intermediate language. This is useful for C++, where pattern matching is not available.
- Rust does not support patterns that traverse pointers (although there are proposed features like `box_patterns` and `defer_patterns` in unstable Rust to overcome this problem). Hence, we replace the pointer subpattern by a variable and translate it as a condition appended to the guard and some variable declarations.
- Since the semantics of pattern disjunction with guards in Python differ from our intended meaning, a transformation for eliminating disjunctive patterns is applied. It translates a guarded branch with a pattern disjunction into several branches, after resolving all disjunctions that deeply appear in the pattern. This implies copying the guard and body several times.

An example of the *dematch* transformation is shown in Section 5.

4.4 From the intermediate representation to the target languages

Once the intermediate representation is available and conveniently transformed, code generation in the target language is almost straightforward. Some adaptations are required for languages that do not support name overloading (Python

and Rust), in which the declaration order is relevant (C++, a topological order of function dependencies is calculated), etc. However, the most critical concern at this point is memory management, which is explicit in C++ and Rust. For convenience, we take arguments by constant reference (in C++) or borrow them (in Rust), always return copies of objects, and use smart pointers in recursive data types. When reducing with equations or rewriting with rules, modifying the term in place would be more efficient and should be considered as future work. The general principle of the translation is to generate code as conventional as possible in the target language, so implementing a significant runtime for term manipulation or *hash consing* [10] is a non-goal.

4.5 Testing the translation

Differential testing [19] is a testing technique in which the same input is executed on multiple independent implementations of a system, and their outputs are compared to detect inconsistencies. We have written several Maude specifications along with some MUnit tests. On the one hand, these tests are executed by the MUnit tool on the Maude specification. On the other hand, our code generator takes them as input, recognizes `assertTrue`, `-False`, `-Equal`, and `-LeqSort` tests, and generates tests in the target language using the frameworks mentioned in Section 3.4. A simple script `tester.py` is provided to generate the code from the Maude specification, build it, and run the tests in the target languages.

5 Examples

This section illustrates the translation of Section 4 with several examples. We have run small experiments for some of them to compare the performance between the different target languages and Maude. The experiments ran under Linux in an Intel Xeon machine, with the official build of Maude 3.5.1, Rust 1.92.0 (without or with the `-O` optimization flag), GCC 13.3.0 as the C++ compiler (without or with the `-O3` flag), CPython 3.12.3, and PyPy 7.3.20.

5.1 Numeric functions

The `Bool`, `Nat`, `Int`, and `Float` types in Maude are translated to the corresponding native types of the target language. This can be observed with the following Fibonacci function:

```
op fib : Nat -> Nat .
eq fib(0) = 0 .  eq fib(1) = 1 .
eq fib(s s N) = fib(s N) + fib(N) .  var N : Nat .
```

Then, the code generator can be run with the following command

```
$ maudec tests/nat-funs.maude --target rust
```

to produce the following Rust code

	Maude	Rust	`-O`	C++	`-O3`	CPython	PyPy
`fib`	3 min 57 s	3.4 s	893 ms	2.65 s	560 ms	1 min 9.9 s	10.85 s
`eeuclid`	9.58 s	121.6 ms	60.4 ms	269.5 ms	57.9 ms	1.22 s	247 ms
`minimum`	26.68 s	226.4 ms	2.1 ms	86.3 ms	1.4 ms	108.8 ms	107.1 ms
`palindr`	50 ms	14.2 ms	3 ms	30.2 ms	7.5 ms	52.34 s	61.16 s

Table 1. Performance comparison of various examples in the different languages.

```
fn fib(a1: i32) -> i32 {
  match a1 {
    0 => 0, 1 => 1,
    n if n >= 2 => fib((n - 2) + 1) + fib(n - 2),
    _ => panic!("case not covered"),
  }
}
```

Observe that the guard `n >= 2` and the fallback case could be removed if we found out that they are redundant (i.e., the equations are complete). However, we cannot drop the fallback case without removing the guard, because the Rust compiler will complain that the cases are not exhaustive. Moreover, the guard is the result of translating the pattern `s s N` of the third equation, while the occurrences of `N` in the right-hand side have been translated to `n - 2`.

We have measured the time for computing all Fibonacci numbers from 1 to 40. Table 1 shows that this takes almost 4 minutes in Maude, 839 ms in an optimized build of the generated Rust code, 560 ms with an optimized C++ build, and almost 11 seconds for the Python program run with PyPy. We should take into account that the C++ and Rust translation uses machine integers, while Maude and Python use big integers, which are slower. This example is artificial, since memoization can drastically reduce the computation effort. In effect, with the `memo` flag in the `fib` operator [5, §3.4.8], Maude runs the experiment in 1 ms.

A more interesting and complex example is the following specification of the extended Euclidean algorithm.

```
sort EuclidResult .
op euc : Int Int Int -> EuclidResult [ctor] . *** result
op eeuclid : Int Int -> EuclidResult .
op eeuclid : Int Int Int Int Int Int -> EuclidResult .
vars N M U1 V1 U2 V2 : Int .
eq eeuclid(M, N) = eeuclid(M, 1, 0, N, 0, 1) .
eq eeuclid(M, U1, V1, 0, U2, V2) = euc(M, U1, V1) .
eq eeuclid(M, U1, V1, N, U2, V2) =
     eeuclid(N, U2, V2, M - (M quo N) * N,
             U1 - (M quo N) * U2, V1 - (M quo N) * V2) .
```

The result of `eeuclid(`m`, `n`)` is a term `euc(`d`, `x`, `y`)` with the greatest common divisor d of m and n, and the Bézout coefficients x and y. The tool yields the following Rust code:

```
#[derive(Clone, PartialEq, PartialOrd)]
enum EuclidResult { Euc(i32, i32, i32), Err, }

fn eeuclid(m: i32, u1: i32, v1: i32,
           n: i32, u2: i32, v2: i32) -> EuclidResult {
  match n {
    0 => EuclidResult::Euc(m, u1, v1),
    n => { let aux1 : i32 = m / n;  // repeated subterm
      eeuclid(n, u2, v2, m - n * aux1,
              u1 - u2 * aux1, v1 - v2 * aux1)
    },
}}
fn eeuclid_1(m: i32, n: i32) -> EuclidResult {
  eeuclid(m, 1, 0, n, 0, 1)
}
```

where `eeuclid_1` and `eeuclid` are the two overloads of `eeuclid` in the Maude program, and the `euc` constructor is now the `Euc` variant of the sum type. `EuclidResult` also contains an *error term* `Err` for partial functions. Inserting this is avoided for kinds that are considered error-free by Maude.

In this case, we have measured the time to compute `euclid` on every input in $\{1, \ldots, 1000\}^2$, and the results appear in the second row of Table 1. Maude takes around 9.5 seconds, while the optimized Rust and C++ builds take approximately 60 ms. It seems that numerical functions can be significantly accelerated with the translation.

5.2 Commutative operator

The minimum of two numbers can be defined in Maude with this simple conditional equation on a commutative operator:

```
op minimum : Nat Nat -> Nat [comm] .
ceq minimum(M, N) = M if M <= N .  vars M N : Nat .
```

This is translated to the following Rust code

```
fn minimum(n: i32, m: i32) -> i32 {
  match (n, m) { (n, m) | (m, n) if m <= n => m, }
}
```

where commutativity is handled by the disjunction of patterns, so that the two possible orderings of the arguments are tried. We can also produce C++ code:

```
int minimum(int N, int M) {
  if (M <= N) { return M; } if (N <= M) { return N; }
}
```

The body of the function has been *dematched* in the internal representation to the two conditional statements. Both translations are artificial, but correct and relatively efficient, as shown in the third row of Table 1 for its evaluation on $\{1, \ldots, 5000\}^2$. Given the high number of inputs, the measurement for Maude

may be affected by the recursive function that iterates over them to call the `minimum`, which is implemented by two loops in the other languages.

Since this operator is not a constructor, we do not generate both a variant constructor and a normalization function, but only the function itself. See the `unordered-pair` test in [29] for an example of a commutative constructor.

5.3 Stacks

In Section 4.2, we introduced stacks of integer values and defined an `empty` predicate on them. Such data structures are usually specified as parameterized modules, like `fmod STACK{X :: TRIV}`. However, we have not considered translating parameterized types to generic types in the target language, so we only work with instances of these types. The definition of the datatype will be

```
#[derive(Clone, PartialEq, PartialOrd)]
enum Stack { Mt, Push(i32, Box<Stack>), Err, }
```

where `Mt` represents the empty list, `Push` appends an element to the stack, and `Err` is the error term. Observe that the recursive argument of `Push` is wrapped into a `Box`, which is a unique pointer to a heap allocated object. Hence, the implementation has to deal with this pointer explicitly.

In addition to the `empty` predicate defined in Section 4.2, we can define a partial function `pop` to remove the top element of the stack.

```
op pop : Stack ~> Stack .
eq pop(push(N, S)) = S . var N : Int . var S : Stack .
```

This is translated to the Rust code

```
fn pop(a1: &Stack) -> Stack {
  match a1 {
    NatStack::Push(_, paux) => (**paux).clone(),
    _ => Err,
  }
}
```

where a copy of the nested stack is returned (the copy can be avoided with a more sophisticated memory management). Moreover, there is a fallback case that returns the term `Err` when the partial function is invoked out of its domain.

5.4 Palindromes

Consider the following module with a function that checks whether a word on some vowel symbols is a palindrome:

```
fmod PALINDROME is
  sorts Symbol List . subsort Symbol < List .
  ops a e i o u : -> Symbol [ctor] .
  op nil : -> List [ctor] .
  op __ : List List -> List [ctor assoc id: nil] .
```

	Maude	Rust	-O	C++	-O3	CPython	PyPy
fib	7.63 Mb	1.74 Kb	1.74 Kb	73.73 Kb	73.73 Kb	1.23 Mb	107.73 Mb
eeuclid	7.65 Mb	1.74 Kb	1.74 Kb	73.73 Kb	73.73 Kb	1.35 Mb	16.57 Mb
minimum	7.05 Mb	1.74 Kb	1.74 Kb	73.73 Kb	73.73 Kb	1.23 Mb	13.92 Mb
palindr	23 Mb	66.1 Kb	66.1 Kb	198 Kb	198 Kb	7.69 Gb	9.12 Gb

Table 2. Peak memory usage of the same examples and languages in Table 1.

```
  op palindrome : List -> Bool .
  vars S S1 S2 : Symbol . var L : List .
  eq palindrome(nil) = true . eq palindrome(S) = true .
  eq palindrome(S L S) = palindrome(L) .
  eq palindrome(S1 L S2) = false [owise] .
endfm
```

Our tool translates it to the following type declaration and function using the slice patterns of Rust (which are similar to Python's).

```
#[derive(Clone, PartialEq, PartialOrd)]
enum Symbol { A, E, I, O, U, }

fn palindrome(s: &[Symbol]) -> bool {
  match s { [] => true, [_] => true,
    [s1, l @ .., s2] if s1 == s2 => palindrome(l),
    [_, .., _] => false,
  }
}
```

Because the list constructor cannot appear in other constructors of the kind, it has been excluded from the datatype definition and a sequence type in Rust (`&[Symbol]`) is used, as explained in Section 4.2.

In Table 2, we can observe a significant reduction in memory usage when translating to Rust and C++, but a huge rise when translating to Python. Slices in Rust and C++ are views over a memory segment of the original list (essentially a pointer and a length), while they are full copies in Python. Table 1 also shows a much greater execution time for Python.

5.5 Rule application

As explained in Section 4, the translation defines some functions to execute single-step rewrites within the given term for those kinds where a rewrite is possible. Here we illustrate them with a small example, whose original Maude code is the following:

```
mod NESTED-REWRITE is
  sorts Foo Bar .
```

```
  op f : Bar Foo -> Foo [ctor] .
  ops a b c : -> Bar [ctor] .

  rl [ab] : a => b .
  rl [bc] : b => c .
endm
```

In Rust, the compiler generates the following data definitions:

```
#[derive(Clone, PartialEq, PartialOrd, Debug)]
enum Bar { A, B, C, }
#[derive(Clone, PartialEq, PartialOrd, Debug)]
enum Foo { F(Bar, Box<Foo>), }
```

and a function `rlapp_Bar` that applies either `ab` or `bc` (i.e., all rules defined on the kind `Bar`) on top of the given term:

```
fn rlapp_Bar(term: &Bar) -> Option<Bar> {  // on top
  match term {
    Bar::A => Some(Bar::B),
    Bar::B => Some(Bar::C),
    _ => None,
  }
}
```

The return value can be either `Some(`t'`)` for a rewritten term t', or `None` if no rewrite is possible. For rewriting anywhere within a term, two more functions are provided, `arlapp_Bar` and `arlapp_Foo`, one for each kind.

```
fn arlapp_Bar(term: &Bar) -> Option<Bar> {
  rlapp_Bar(term)  // only possible on top
}

fn arlapp_Foo(term: &Foo) -> Option<Foo> {
  match term {
    Foo::F(t0, t1) => {
      if let Some(subt) = arlapp_Bar(t0) {
        return Some(Foo::F(subt, (*t1).clone()));
      }
      if let Some(subt) = arlapp_Foo(&**t1) {
        return Some(Foo::F((*t0).clone(), subt.into()));
      }
    },
  }
  None  // no rewrite possible on top
}
```

The `arlapp_Foo` function decomposes the terms of kind `Foo`, tries to apply the `arlapp` functions recursively on the arguments, and finally returns the recomposed rewritten term if the nested rewrite succeeded. The case of the second argument is more involved, since this argument is recursive and mediated by a pointer (`Box<Foo>`).

6 Conclusions and ongoing work

In this paper we have described a translation from Maude functional modules to different modern imperative languages. The translation starts from a Maude specification and builds an abstract syntax tree, which is used as intermediate language. This tree is later used for generating code in C++, Rust, Python, and Dafny, which provide features that facilitate a natural translation, despite the significantly higher expressiveness of Maude specifications. We have also implemented a translation for test cases, hence supporting differential testing and providing a higher level of confidence. Our experiments with simple examples show that the obtained implementations are significantly faster than the original Maude code. Moreover, the translation to Dafny expands the spectrum of verification techniques.

The current implementation establishes a solid foundation for extensions. We are currently extending the subset of Maude specifications supported by the translation. It is especially interesting to analyze how to integrate ACU operators and rewrite rules. Finally, we aim to assess our approach using a larger set of benchmarks. These benchmarks should include a wide range of applications, including larger specifications than the ones shown in this paper, the new features under development, and also specifications that require proving properties with Dafny, illustrating how Maude verification techniques are complemented.

We also consider studying the usage of artificial intelligence (AI) tools in different directions. A promising application of AI lies in the automated generation of test cases, where it can efficiently explore a wide range of input scenarios. We will take advantage of this fact for obtaining larger test suites. Moreover, AI can be asked to improve the efficiency of the generated code, as it works better with mainstream programming languages. In this case, we would analyze how the code evolves and take advantage of tests for assessing equivalence.

Acknowledgments. We would like to thank Narciso Martí-Oliet for his comments on this manuscript. This work is partially supported by MCIU/AEI/10.13039/50110-0011033/FEDER,UE through project ProCode 10 (PID2023-149943OB-I00) and by Comunidad de Madrid/UCM through project COMFIA (PR17/24-31913).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the contents of this article.

References

1. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. **28**(1), 6:1–6:39 (2018). https://doi.org/10.1145/3158668
2. Blandy, J., Orendorff, J., Tindall, L.F.S.: Programming Rust: Fast, Safe Systems Development. O'Reilly, 2nd edn. (2021)
3. Bouhoula, A., Jouannaud, J., Meseguer, J.: Specification and proof in membership equational logic. Theor. Comput. Sci. **236**(1-2), 35–132 (2000). https://doi.org/10.1016/S0304-3975(99)00206-6

4. van den Brand, M., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: the ASF+SDF compiler. ACM Trans. Program. Lang. Syst. **24**(4), 334–368 (2002). https://doi.org/10.1145/567097.567099
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.L.: Maude Manual v3.5.1 (July 2025), https://maude.cs.illinois.edu/manual
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, LNCS, vol. 4350. Springer (2007). https://doi.org/10.1007/978-3-540-71999-1
7. Denker, G., Talcott, C.L., Rosu, G., van den Brand, M., Eker, S., Serbanuta, T.: Rewriting logic systems. In: WRLA 2006. ENTCS, vol. 176(4), pp. 233–247. Elsevier (2006). https://doi.org/10.1016/j.entcs.2007.06.018
8. Eker, S.: Fast matching in combinations of regular equational theories. In: Meseguer, J. (ed.) RWLW 1996. ENTCS, vol. 4, pp. 90–109. Elsevier (1996). https://doi.org/10.1016/S1571-0661(04)00035-0
9. Garavel, H., Tabikh, M., Arrada, I.: Benchmarking implementations of term rewriting and pattern matching in algebraic, functional, and object-oriented languages - the 4th rewrite engines competition. In: Rusu, V. (ed.) WRLA 2018. LNCS, vol. 11152, pp. 1–25. Springer (2018). https://doi.org/10.1007/978-3-319-99840-4_1
10. Goto, E.: Monocopy and associative algorithms in extended Lisp. Tech. rep., University of Tokyo (1974)
11. Hafner, C.D., Godden, K.: Portability of syntax and semantics in DATALOG. ACM Transactions on Information Systems **3**(2), 141–164 (Apr 1985). https://doi.org/10.1145/3914.3982
12. Jordan, H., Scholz, B., Subotic, P.: Soufflé: On synthesis of program analyzers. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016, Part II. LNCS, vol. 9780, pp. 422–430. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_23
13. Kirchner, H., Moreau, P.: Promoting rewriting to a programming language: a compiler for non-deterministic rewrite programs in associative-commutative theories. J. Funct. Program. **11**(2), 207–251 (2001). https://doi.org/10.1017/S0956796800003907
14. Klabnik, S., Nichols, C., Krycho, C., et al.: The Rust Programming Language. No Starch Press (2026)
15. Lincoln, P., Martí-Oliet, N., Meseguer, J., Ricciulli, L.: Compiling rewriting onto SIMD and MIMD/SIMD machines. In: Halatsis, C., Maritsas, D.G., Philokyprou, G., Theodoridis, S. (eds.) PARLE'94. LNCS, vol. 817, pp. 37–48. Springer (1994). https://doi.org/10.1007/3-540-58184-7_88
16. Lincoln, P., Meseguer, J., Ricciulli, L.: The rewrite rule machine node architecture and its performance. In: Buchberger, B., Volkert, J. (eds.) CONPAR 94 - VAPP VI. LNCS, vol. 854, pp. 509–520. Springer (1994). https://doi.org/10.1007/3-540-58430-7_45
17. Lutz, M.: Programming Python. O'Reilly, 4th edn. (2011)
18. Martin-Martin, E., Montenegro, M., Riesco, A., Rodríguez-Hortalá, J., Rubio, R.: Verification of the ROS NavFn planner using executable specification languages. J. Log. Algebraic Methods Program. **132**, 100860 (2023). https://doi.org/10.1016/j.jlamp.2023.100860
19. McKeeman, W.M.: Differential testing for software. Digit. Tech. J. **10**(1), 100–107 (1998)
20. Meseguer, J.: Twenty years of rewriting logic. J. Log. Algebr. Program. **81**(7-8), 721–781 (2012). https://doi.org/10.1016/j.jlap.2012.06.003

21. Moreau, P.: REM (reduce elan machine): Core of the new ELAN compiler. In: Bachmair, L. (ed.) RTA 2000. LNCS, vol. 1833, pp. 265–269. Springer (2000). https://doi.org/10.1007/10721975_19
22. Mossakowski, T., Maeder, C., Lüttich, K.: The heterogeneous tool set, Hets. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 519–522. Springer (2007). https://doi.org/10.1007/978-3-540-71209-1_40
23. de Moura, L.M., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean theorem prover. In: CADE 2015. LNCS, vol. 9195, pp. 378–388. Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_26
24. Ogata, K., Ohhara, K., Futatsugi, K.: TRAM: an abstract machine for order-sorted conditioned term rewriting systems. In: Comon, H. (ed.) RTA-97. LNCS, vol. 1232, pp. 335–338. Springer (1997). https://doi.org/10.1007/3-540-62950-5_84
25. Project, D.: Dafny, https://dafny.org/
26. Riesco, A.: MUnit: A unit framework for Maude. In: Rusu, V. (ed.) WRLA 2018. LNCS, vol. 11152, pp. 45–58. Springer (2018). https://doi.org/10.1007/978-3-319-99840-4_3
27. Rubio, R.: Maude as a library: an efficient all-purpose programming interface. In: Bae, K. (ed.) WRLA 2022. LNCS, vol. 13252, p. 14. Springer (2022). https://doi.org/10.1007/978-3-031-12441-9_14
28. Rubio, R., García, B.A., Riesco, A.: An experimental Maude compiler to modern imperative languages (2026), https://github.com/fadoss/maudec
29. Rubio, R., Riesco, A.: Maude to Lean translator (2024), https://github.com/fadoss/maude2lean
30. Rubio, R., Riesco, A.: Maude2Lean: Theorem proving for Maude specifications using Lean. J. Log. Algebraic Methods Program. **142**, 101005 (2025). https://doi.org/10.1016/j.jlamp.2024.101005
31. Stroustrup, B.: The C++ Programming Language. Addison Wesley, 4th edn. (2014)

Generating Invariants by Deductive Model Checking

Kyungmin Bae[1], Santiago Escobar[2], Raúl López-Rueda[2], José Meseguer[3], and Julia Sapiña[2]

[1] Pohang University of Science and Technology, Pohang, South Korea
[2] VRAIN, Universitat Politècnica de València, Valencia, Spain
[3] University of Illinois at Urbana-Champaign, Urbana, IL, USA.

Abstract. We present several significant new advances on the *deductive model checking* verification methodology for infinite-state systems supported by the DM-Check tool, which can prove a conjectured inductive invariant represented as a *disjunction of constrained patterns* by showing that the conjectured set is transition-closed. The new advances include: (i) several new inference rules to reason about semantic equivalence and generalization between constrained patterns; (ii) an *invariant generation* algorithm and methodology to compute the set of *all reachable states* from a disjunction of constrained patterns as a *fixpoint* representable as another disjunction of constrained patterns; (iii) a new version of the DM-Check tool supporting these new advances; and (iv) two case studies illustrating the practical usefulness of these advances.

1 Introduction

We present several significant advances in the *deductive model checking* (in the sense of [1,2]) of infinite-state concurrent systems specified as rewrite theories $\mathcal{R} = (\Sigma, E \cup B, R)$ [16]. The states of $\mathcal{R}$ are elements of the initial algebra $\mathbb{T}_{\Sigma/E\cup B}$, and its transitions are specified by the rewrite rules R. Infinite sets of states in $\mathcal{R}$ are represented as disjunctions of *constrained patterns* of the form: $u_1 \mid \varphi_1 \vee \ldots \vee u_n \mid \varphi_n$ where each $u_i \mid \varphi_i$ consists of a constructor term u_i denoting the (usually infinite) set of system states that are (ground) substitution instances of u_i, and a conjunction φ_i of Σ-equations that *constrain* the allowed ground substitutions ρ. That is, we only allow those ρ such that $\varphi\rho$ is inductively valid in $\mathbb{T}_{\Sigma/E\cup B}$. Such constraints (and therefore the sets of states they define) are very expressive, since the equational theory $(\Sigma, E\cup B)$ may contain arbitrarily complex recursive function definitions, which can be extended with other *auxiliary function definitions* added just to make constrained patterns even more expressive (see examples in Sections 4.1 and 5.2). This expressive power is really needed to specify important system properties such as invariants, but it comes at a cost. For example, the simple question of whether a constrained pattern $u \mid \varphi$ is *more general* (contains more states) than another one $v \mid \psi$ may require inductive theorem proving to be answered: we need to (i) find a B-matching substitution α such that $v =_B (u\alpha)$ (using a B-matching algorithm for axioms B such as

associativity and/or commutativity and/or unit element axioms); and then (ii) prove that $\psi \Rightarrow (\phi\alpha)$ is *valid* in $\mathbb{T}_{\Sigma/E \cup B}$ using an inductive theorem prover.

The key idea is to compute the set of states *reachable in one step* by transitions in R from the set of states denoted by $u_1 \mid \varphi_1 \vee \ldots \vee u_n \mid \varphi_n$ by performing *constrained narrowing* [14], which provides a *finite representation* of that set of states *reachable in one step* as another pattern disjunction $v_1 \mid \psi_1 \vee \ldots \vee v_m \mid \psi_m$. This provides a method to prove *inductive invariants* of $\mathcal{R}$, that is, sets of states that are *transition closed*. Indeed, $u_1 \mid \varphi_1 \vee \ldots \vee u_n \mid \varphi_n$ will be an inductive invariant iff the one-transition-step states denoted by $v_1 \mid \psi_1 \vee \ldots \vee v_m \mid \psi_m$ are a *subset* of the set of states denoted by $u_1 \mid \varphi_1 \vee \ldots \vee u_n \mid \varphi_n$. The problem is that, although the "model checking" reachability part is automatic by constrained narrowing, as we have already seen, just for two constrained patterns proving containment is nontrivial, since it requires inductive theorem proving. Proving containment between *disjunctions* of such patterns is even harder and *requires further deductive reasoning*, thus the "deductive" in "deductive model checking".

In [1,2] we presented a methodology for *proving inductive invariants* by deductive model checking using the DM-Check tool, which itself uses Maude's **NuITP** inductive theorem prover [8] as a backend. In this paper we report on the following significant new advances after [1,2]:

1. *Four new inference rules* that support deductive reasoning about the *semantic equivalence* between disjunctions of patterns. They complement other inference rules in [1,2] and are useful for proving containment between sets of states.
2. A *fifth new rule* to over-approximate disjunctions of constrained patterns by *generalization* (the inverse of containment).
3. A substantial *extension* of our deductive model checking methodology from the prior case of proving inductive invariants (which only requires a single step of constrained narrowing), to the more general case of *generating* inductive invariants by *several steps* of constrained narrowing. That is, we begin with a pattern disjunction $u_1 \mid \varphi_1 \vee \ldots \vee u_n \mid \varphi_n$ and try to obtain a *finite representation* of *all states* reachable from $u_1 \mid \varphi_1 \vee \ldots \vee u_n \mid \varphi_n$ (i.e., a *fixpoint* of the transition relation R) as a pattern disjunction $v_1 \mid \psi_1 \vee \ldots \vee v_m \mid \psi_m$.
4. A *new version* of the DM-Check tool that supports all of the new capabilities described in (1)–(3) above.
5. Two *case studies* demonstrating the practical usefulness of all these advances.

Related Work. We briefly discuss work closely related to DM-Check and refer to [2] for further details. The work closest to ours is [21] and the Maude Invariant Analyzer (InvA), an early approach that demonstrates the benefits of combining automated and interactive deduction; its functionality is subsumed by DM-Check. Invariant verification using proof scores in CafeOBJ [12,19,18,22] has also been applied to many case studies, but typically requires more manual effort than DM-Check. Our work is also related to narrowing-based reachability analysis and logical model checking [10,3,4,5,6,9], which usually rely on the finite variant property, thus restricting the class of systems that can be handled. Ivy [20,13] combines model checking with theorem proving for infinite-state systems, but emphasizes automatically discharging verification conditions using SMT.

2 Preliminaries

We assume familiarity with rewriting logic and refer the reader to [16,2].

We assume an order-sorted signature Σ and the term Σ-algebras $\mathbb{T}_\Sigma$ and $\mathbb{T}_\Sigma(X)$ for X an S-sorted set of variables. The subterm of term t at a term position p is denoted $t|_p$ and the term replacement of $t|_p$ by w at position p is denoted $t[w]_p$. Given an order-sorted substitution θ, its domain is $dom(\theta)$ and its range $ran(\theta)$. The application of a substitution θ to a term t is $t\theta$. The order-sorted equational deduction relation is denoted $E \vdash u = v$ and its associated E-equality relation $=_E$. The term Σ-algebras modulo E are $\mathbb{T}_{\Sigma/E}$ and $\mathbb{T}_{\Sigma/E}(X)$ and we denote the equivalence class of a term t by $[t]_E$ (or just $[t]$). The *E-unifiers* of an equation $u = v$ are substitutions θ such that $u\theta =_E v\theta$.

We assume an equational theory $(\Sigma, E \cup B)$ having a ground convergent decomposition $(\Sigma, B, \vec{E})$, a constructor subtheory $(\Omega, B_\Omega, \varnothing) \subseteq (\Sigma, B, \vec{E})$, and a sort *State* at the top of one of its connected components whose data elements denote states of a rewrite theory having $(\Sigma, B, \vec{E})$ as its equational subtheory. A constrained constructor pattern is of the form $u \mid \varphi$, where $u \in T_{\Omega,State}(X)$, with X_s countable for each sort s in Σ, and φ is a conjunction of Σ-equalities. We denote its semantics as $[\![u \mid \varphi]\!] = \{[v] \in T_{\Omega/B_\Omega,State} \mid \exists \rho \in [X \to T_\Omega] \; s.t. \; [v] = [u\rho] \wedge E \cup B \models \varphi\rho\}$. Pattern subsumption is denoted by $u \mid \varphi \sqsubseteq_{B_\Omega} v \mid \psi$, which by definition holds iff there exists a B_Ω-matching substitution α such that: (i) $u =_{B_\Omega} (v\alpha)$, and (ii) $\mathbb{T}_{\Sigma/E\cup B} \models \varphi \Rightarrow (\psi\alpha)$.

A rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$ is *topmost* iff it has a sort *State* such that: (i) if $f(t_1, \ldots, t_n)$ has sort *State*, none of the t_i, $1 \leqslant i \leqslant n$ nor any subterm of any such t_i has sort *State*, and (ii) for any rewrite rule $l \to r$ *if* φ in $\mathcal{R}$, l and r have sort *State*. Constrained narrowing between constrained constructor terms is defined as $u \mid \varphi \overset{\alpha}{\leadsto}_{R,B_\Omega} v \mid \psi$ that holds iff there exists a rule $l \to r$ *if* ϕ in R and a disjoint[4] B_Ω-unifier α of the equation $u = l$ such that $v \mid \psi = (r \mid \varphi \wedge \phi)\alpha$.

3 DM-Check Methodology and Earlier Versions

DM-Check[5] [1,2] provides a novel deductive model checking methodology in which narrowing-based logical model checking of symbolic states—specified as disjunctions of constrained patterns—is combined with inductive theorem proving to discharge inductive verification conditions that ensure useful symbolic state space reductions. In fact, constrained narrowing [14] is a form of *deductive model checking* by itself, where symbolic model checking and inductive theorem proving are combined. In particular:

1. The existence of a ground substitution ρ such that $E \cup B \vdash (\psi \wedge \psi_j)\beta\rho$ is equivalent to saying that $\mathbb{T}_{\Sigma/E\cup B} \models \exists(\psi \wedge \psi_j)\beta$, where $\exists(\psi \wedge \psi_j)\beta$ denotes the existential closure of $(\psi \wedge \psi_j)\beta$; an *inductive* satisfiability property.

[4] θ is a disjoint B_Ω-unifier of $u = l$ iff $vars(u) \cap vars(l) = \varnothing$. Unifier disjointness can always be ensured by renaming variables.

[5] DM-Check is publicly available at `https://safe-tools.dsic.upv.es/dmc`.

2. Constrained narrowing search from $\bigvee_{i\in I} u_i \mid \varphi_i$ to find an intersection with $\bigvee_{j\in J} v_j \mid \psi_j$ will usually generate an infinite *narrowing forest* that symbolically describes the set of all states reachable from $[\![\bigvee_{i\in I} u_i \mid \varphi_i]\!]$.
3. However, if using the subsumption relation $\sqsubseteq_{B_\Omega}$ we can *fold* such an infinite narrowing tree into a *finite* narrowing graph, then only a finite number of constrained patterns need to be examined in finite time.
4. Folding means that any constrained pattern $u \mid \varphi$ found at depth $k+1$ is subsumed by some other constrained pattern $v \mid \psi$ found at depth $j \leqslant k$, i.e., $u \mid \varphi \sqsubseteq_{B_\Omega} v \mid \psi$. But then inductive theorem proving is essential, since, as explained in Section 2, for some matching substitution α we need to prove $\mathbb{T}_{\Sigma/E\cup B} \models \varphi \Rightarrow (\psi\alpha)$.

A simple integration of DM-Check with an inductive theorem prover is to use the inductive theorem prover in a terminating, automated mode as an oracle. But more powerful combinations are possible: DM-Check also allows that inductive verification conditions not discharged automatically by the oracle are dealt with by commands that refine some constrained patterns by useful semantic equivalences.

NuITP [8] is a next-generation inductive theorem prover written in Maude that combines advanced symbolic techniques such as narrowing, equality predicates, variant unification, variant satisfiability, order-sorted congruence closure, ordered rewriting, and strategy-based rewriting. NuITP implements the inductive inference system of [15,7] to verify properties of equational theories $(\Sigma, E \cup B)$ specified in Maude, where Σ is an order-sorted signature, B is any combination of the associativity (A), commutativity (C) and identity (U) axioms, and E is a set of ground convergent, possibly conditional equations. DM-Check uses NuITP as an oracle for two purposes: (i) to check subsumptions $u \mid \varphi \sqsubseteq_{B_\Omega} v \mid \psi$ for the constructor subtheory (Ω, B_Ω), and (ii) to check whether $u \mid \varphi$ denotes an empty set of states because φ is inductively unsatisfiable.

Invariants in DM-Check can be proved in several ways. Sections 4 and 5 below provide *new* commands. Commands supported in earlier versions [1,2] include:

Checking inductive invariant. The command "`check ind-invariant` $\bigvee_{i\in I} u_i \mid \varphi_i$" performs one step of constrained narrowing to check that $\bigcup_{i\in I}[\![u_i \mid \varphi_i]\!]$ is *transition-closed*. Any resulting constrained patterns not subsumed by $\bigvee_{i\in I} u_i \mid \varphi_i$ are returned as proof obligations. A default[6] NuITP proof strategy σ is used as an oracle to prove subsumptions and discard patterns with unsatisfiable constraints.
Verifying other invariants positively. The command "`check` $\bigvee_{i\in I} u_i \mid \varphi_i$ `subsumed by` $\bigvee_{j\in J} v_j \mid \psi_j$" tries to prove $\bigcup_{i\in I}[\![u_i \mid \varphi_i]\!] \subseteq \bigcup_{j\in J}[\![v_j \mid \psi_j]\!]$ using NuITP as an oracle. If DM-Check can prove this containment automatically only for a subset $I_0 \subseteq I$, it returns the remaining patterns $u_i \mid \varphi_i$ (for $i \in I\backslash I_0$) as proof obligations to be discharged using NuITP.
Verifying other invariants negatively. The command "`intersect` $\bigvee_{i\in I} u_i \mid \varphi_i$ `with` $\bigvee_{j\in J} v_j \mid \psi_j$" tries to prove $\bigcup_{i\in I}[\![u_i \mid \varphi_i]\!] \cap \bigcup_{j\in J}[\![v_j \mid \psi_j]\!] = \varnothing$. It symbolically computes the intersection using disjoint B_Ω-unification, invoking

[6] At present, a fixed default strategy is provided. In a future version the strategy will be user-definable.

NuITP as an oracle to discard any constrained patterns in the resulting disjunction whose constraints are shown unsatisfiable.

Adding lemmas. The command "`add lemma` φ" lets the user add a supporting lemma, which is sent to NuITP to help automatically discharge verification conditions. Instead of explicitly adding lemmas, the user may delegate intermediate results to NuITP by turning pending proof goals into lemmas.

- The command "`empty` <*id*> `of` <*parent-id*> `.`" indicates that a constrained pattern has no possible *valid* instantiation.
- The command "`lemma` <*id*> `of` <*parent-id*> `.`" marks the constrained pattern as proved, and generates a lemma from its condition.
- The command "`contained` <*id*> `of` <*parent-id*> `.`" indicates that a constrained pattern is folded into the original invariant, and a lemma is automatically generated from the associated subsumption.

All three commands assume *id* identifies one of the remaining constrained patterns, which is a child generated by one narrowing step from the constrained pattern identified by *parent-id* belonging to the original inductive invariant.

Case splitting. The command "`case` <*id*> `of` <*parent-id*> `on` <*variable*> `with` <*generator-set*> `.`" instantiates a variable in the constrained pattern *id*, which is a child generated by one narrowing step from the constrained pattern *parent-id* in the original inductive invariant, where the *generator-set* is a constructor-based generator set for the sort of that variable; see [2] for further details.

4 Pattern Equivalence and Generalization Commands

We propose five new commands that extend the reasoning capabilities of DM-Check in two different ways: (i) four new commands provide new ways of *preserving semantic equivalence* between constrained pattern disjunctions; they complement similar commands described in Section 3; and (ii) a *constrained pattern generalization* command, that allows a user to generalize constrained patterns into more general ones. The new semantic-equivalence-preserving commands are:

Variant Unify. If a constrained pattern has the form $u \mid \varphi \wedge t = t'$ ($\wedge$ is associative-commutative with identity $\top$), where t, t' are terms in an equational subtheory that enjoys the finite variant property, then the equation $t = t'$ has a finite number of variant unifiers, say, $\theta_1, \ldots, \theta_k$, $k \geqslant 0$, and the pattern $u \mid \varphi \wedge t = t'$ is semantically equivalent to the pattern disjunction

$$u\theta_1 \mid \varphi\theta_1 \vee \ldots \vee u\theta_k \mid \varphi\theta_k$$

Narrowing. If a constrained pattern has the form $u \mid \varphi \wedge t[f(u_1, \ldots, u_n)]_p = t'$ ($=$ is commutative), where p is a term position in t, the subterm $f(u_1, \ldots, u_n)$ at p is such that the $u_1, \ldots, u_n$ are constructor terms, f is a free function defined by equations (the case for conditional equations is similar) $E_f = \{f(v_{i.1}, \ldots, v_{i.n}) = r_i\}_{i \in I}$ (with the $v_{i.1}, \ldots, v_{i.n}$ constructor terms), and $f(u_1, \ldots, u_n)$ can be narrowed with the oriented equations E_f (modulo the axioms B of the equational theory) to, say, $\{f(u_1, \ldots, u_n) \overset{\alpha_j}{\leadsto}_{E_f, B_\Omega} w_j\}_{1 \leqslant j \leqslant k}$, then the pattern $u \mid$

$\varphi \wedge t[f(u_1, \ldots, u_n)]_p = t'$ is semantically equivalent to the pattern disjunction:

$$(u \mid \varphi \wedge t[w_1]_p = t')\alpha_1 \vee \ldots \vee (u \mid \varphi \wedge t[w_k]_p = t')\alpha_k$$

Variable Abstraction. If a constrained pattern has the form $u \mid \varphi \wedge t[v]_p = t'$, with v a non-variable subterm, then, it is semantically equivalent to the following constrained pattern, where x is a *fresh* variable whose sort is that of v:

$$u \mid \varphi \wedge t[x]_p = t' \wedge x = v$$

Substitution. If a constrained pattern has the form $u \mid \varphi \wedge x = v$, where the variable x does not occur in v and the smallest sort of v is smaller or equal to the sort of x, then it is semantically equivalent to the constrained pattern:

$$(u \mid \varphi)\{x \mapsto v\}$$

Generalization. This command generalizes a constrained pattern $u \mid \varphi$ to a *more general one* $v \mid \psi$, that is, to a pattern that (modulo the axioms B_Ω) *subsumes* $u \mid \varphi$, i.e., $u \mid \varphi \sqsubseteq_{B_\Omega} v \mid \psi$. The use of the **generalize** command to *over-approximate* states will be illustrated only in Section 5.2.

4.1 A QLOCK Example

The use of the new commands preserving pattern equivalence can be illustrated by an example from [17], where a Maude specification of the `QLOCK` mutual exclusion protocol originally proposed by K. Futatsugi was used to motivate the need for these new commands in DM-Check before they were implemented. The following rewrite rules define the behavior of QLOCK:

```
rl [r2w] : < U i | W | C | Q > => < U | W i | C | Q ; i > .
rl [w2c] : < U | W i | C | i ; Q > => < U | W | C i | i ; Q > .
rl [c2r] : < U | W | C i | i ; Q > => < U i | W | C | Q > .
```

Processes in `QLOCK` are abstractly modeled as natural numbers. The states of `QLOCK` have the general form `< U | W | C | Q >`, where `U` are the processes in the multiset *reception* area, `W` the processes in the multiset *waiting* area, and `C` the processes in the multiset *critical section*, and where `Q` is a list of process names awaiting to enter (or in) the critical section. The meaning of the rules `[r2w]` (reception to waiting), `[w2c]` (waiting to critical) and `[c2r]` (critical to reception) is self-explanatory and formalizes the protocol's semantics.

For a full specification of `QLOCK` and the proof of its properties see `https://safe-tools.dsic.upv.es/dmc/examples/qlock.html`. `QLOCK`'s specification also defines two predicates of sort `MSet -> Boolean` (no duplicates and nonempty) and a function of sort `List -> MSet` (mapping a list to its associated multiset):

```
eq set(mt) = tt .        eq set(i) = tt .        eq set(S S U) = ff .
ceq set(i U) = tt if i in U = ff /\ set(U) = tt .
eq non-mt(mt) = ff [variant] .                   eq l2ms(nil) = mt .
eq non-mt(S) = tt [variant] .                    eq l2ms(n ; L) = n l2ms(L) .
```

Note that only function `non-mt` has the `variant` equational attribute that specifies that these equations satisfy the *finite variant property* [11].

The new **DM-Check** commands are useful here to prove several properties of the `QLOCK` example. Note that `QLOCK` is *parametric* on the *infinite* set of initial states specified by the following constrained pattern, with `S` of sort `NeMSet`:

$$\texttt{< S | mt | mt | nil > | set(S) = tt} \tag{1}$$

The *inductive invariant* associated to `QLOCK` (i.e., transition-closed invariant) from the initial state pattern (1) is the *mutual exclusion property* defined as follows:

$$\begin{array}{l} \texttt{< U | W | mt | Q >} \mid \texttt{non-mt(U W) = tt} \wedge \texttt{set(U W) = tt} \wedge \texttt{l2ms(Q) = W} \quad \vee \\ \texttt{< V | T | i | i ; Q' >} \mid \texttt{non-mt(V T i) = tt} \wedge \texttt{set(V T i) = tt} \wedge \\ \qquad\qquad\qquad\qquad \texttt{l2ms(i ; Q') = T i} \end{array} \tag{2}$$

The conjectured inductive invariant (2) can be verified by **DM-Check** by checking that it folds in one step using its `check ind-invariant` command:

```
DM-Check> check ind-invariant \
    ((< U:MSet | W:MSet | mt | Q:List >) | (non-xmt(U:MSet W:MSet) = tt) \
       /\ (set(U:MSet W:MSet) = tt) /\ (l2ms(Q:List) = W:MSet)) \
 \/ ((< V:MSet | T:MSet | i:Nat | i:Nat ; Q':List >) | \
      (non-mt(V:MSet T:MSet i:Nat) = tt) /\ (set(V:MSet T:MSet i:Nat) = tt) \
       /\ (l2ms(i:Nat ; Q':List) = T:MSet i:Nat)) .

    Invariant satisfied.
```

The inductive invariant satisfies that all its states in their critical section component are either `mt` or a single process `i`.

However, we still need to check that our inductive invariant (2) is actually an invariant *from* the initial state pattern (1). This can be checked by using the `check subsumed by` command:

```
DM-Check> check ((< S:NeMSet | mt | mt | nil >) | (set(S:NeMSet) = tt)) \
    subsumed by ((< U:MSet | W:MSet | mt | Q:List >) | \
                  (non-mt(U:MSet W:MSet) = tt) \
                   /\ (set(U:MSet W:MSet) = tt) /\ (l2ms(Q:List) = W:MSet)) \
           \/  ((< V:MSet | T:MSet | i:Nat | i:Nat ; Q':List >) | \
                  (non-mt(V:MSet T:MSet i:Nat) = tt) \
                   /\ (set(V:MSet T:MSet i:Nat) = tt) \
                   /\ (l2ms(i:Nat ; Q':List) = T:MSet i:Nat)) .

    Subsumption satisfied.
```

Any mutex algorithm worth its salt should be deadlock-free. Let us show that our inductive invariant from (1) is *deadlock free*, i.e., that all its states are *enabled* to perform some *transition* in `QLOCK`. The set of all *transition-enabled states* has a very simple expression as the disjunction of lefthand side patterns:

```
          < U1:MSet n1:Nat | W1:MSet | C1:MSet | Q1:List > | true \/
  < U2:MSet | W2:MSet n2:Nat | C2:MSet | n2:Nat ; Q2:List >| true \/
   < U3:MSet | W3:MSet | C3:MSet n3:Nat | n3:Nat ; Q3:List > | true
```

Deadlock freedom can then be proved with the following subsumption command; however, the subsumption check fails:

```
DM-Check> check ((< U:MSet | W:MSet | mt | Q:List >) | \
                (non-mt(U:MSet W:MSet) = tt) \
                 /\ (set(U:MSet W:MSet) = tt) /\ (l2ms(Q:List) = W:MSet)) \
            \/  ((< V:MSet | T:MSet | i:Nat | i:Nat ; Q':List >) | \
                (non-mt(V:MSet T:MSet i:Nat) = tt) \
                 /\ (set(V:MSet T:MSet i:Nat) = tt) \
                 /\ (l2ms(i:Nat ; Q':List) = T:MSet i:Nat)) \
 subsumed by \
  ((< U1:MSet n1:Nat | W1:MSet | C1:MSet | Q1:List >) | true) \
  \/ ((< U2:MSet | W2:MSet n2:Nat | C2:MSet | n2:Nat ; Q2:List >) | true) \
  \/ ((< U3:MSet | W3:MSet | C3:MSet n3:Nat | n3:Nat ; Q3:List >) | true) .

    Constrained terms on the left that could not be subsumed:

Term 4:  < $17:MSet | $18:MSet | mt | $16:List >
Matching: no matching found
Constraint 4:  (non-mt($17:MSet $18:MSet) = tt) /\ (set($17:MSet $18:MSet)
               = tt)  /\ l2ms($16:List) = $18:MSet
```

for the first pattern of the two from (2) because it is *too general* to show that all its ground instances are *transition-enabled*.

Since `non-mt` enjoys the finite variant property, we can use the new ***variant unify*** command to unify away the equation `non-mt(U W) = tt` using Maude's `filtered variant unify` command:

```
DM-Check> vunify to 4 with non-mt($17:MSet $18:MSet) =? tt .

    Constrained terms on the left that could not be subsumed:

Term 5:  < $19:NeMSet | mt | mt | $20:List >
Matching: no matching found
Constraint 5:  (set($19:NeMSet) = tt) /\ l2ms($20:List) = mt

Term 6:  < mt | $21:NeMSet | mt | $22:List >
Matching: no matching found
Constraint 6:  (set($21:NeMSet) = tt) /\ l2ms($22:List) = $21:NeMSet

Term 7:  < $23:NeMSet | $24:NeMSet | mt | $25:List >
Matching: no matching found
Constraint 7: (set($23:NeMSet $24:NeMSet) = tt) /\ l2ms($25:List) = $24:NeMSet
```

The previous `Term 4` becomes now equivalent to those `Term 5`, `Term 6` and `Term 7`.

We can now use the ***case*** command (which was already available in previous versions of DM-Check) with $\{m{:}Nat, (k{:}Nat\ S{:}NeMSet)\}$ as generator set for sort *NeMSet* to make `Term 5` semantically equivalent to two instances that can then be folded. Indeed, **DM-Check** returns the previous remaining terms after performing the case analysis and checking whether they are folded.

```
DM-Check> case to 5 on $19:NeMSet with m:Nat ;; k:Nat S:NeMSet .

    Constrained terms on the left that could not be subsumed:

Term 6:  < mt | $21:NeMSet | mt | $22:List >
Matching: no matching found
Constraint 6:  (set($21:NeMSet) = tt) /\ l2ms($22:List) = $21:NeMSet

Term 7:  < $23:NeMSet | $24:NeMSet | mt | $25:List >
Matching: no matching found
Constraint 7: (set($23:NeMSet $24:NeMSet) = tt) /\ l2ms($25:List) = $24:NeMSet
```

We can also apply the ***case*** command to `Term 7`, leaving only `Term 6`.

```
DM-Check> case to 7 on $23:NeMSet with m:Nat ;; k:Nat S:NeMSet .

    Constrained terms on the left that could not be subsumed:

Term 6:  < mt | $21:NeMSet | mt | $22:List >
Matching: no matching found
Constraint 6:  (set($21:NeMSet) = tt) /\ l2ms($22:List) = $21:NeMSet
```

Now, we can apply the new ***narrowing*** command to the term `l2ms($22:List)` in the constraint, which in this way becomes equivalent to a pattern disjunction:

```
DM-Check> narrow to 6 on l2ms($22:List) .

    Constrained terms on the left that could not be subsumed:

Term 7:  < mt | $23:NeMSet | mt | nil >
Matching: no matching found
Constraint 7:  (set($23:NeMSet) = tt) /\ mt = $23:NeMSet

Term 8:  < mt | $24:NeMSet | mt | $25:Nat ; $26:List >
Matching: no matching found
Constraint 8: (set($24:NeMSet) = tt) /\ ($25:Nat l2ms($26:List)) = $24:NeMSet
```

We are now only left with two patterns that can easily be discarded. First, by using the previous ***variant unify*** command:

```
DM-Check> vunify to 7 with $23:NeMSet =? mt .

    Constrained terms on the left that could not be subsumed:

Term 8:  < mt | $24:NeMSet | mt | $25:Nat ; $26:List >
```

```
Matching: no matching found
Constraint 8: (set($24:NeMSet) = tt) /\ ($25:Nat l2ms($26:List)) = $24:NeMSet
```

`Term 8` cannot be folded but using the new ***variable abstraction*** command on `l2ms(L:List)` with fresh variable `U:MSet` becomes semantically equivalent:

```
DM-Check> va to 8 on l2ms($26:List) with $27:MSet .

    Constrained terms on the left that could not be subsumed:

Term 9:  < mt | $27:NeMSet | mt | $28:Nat ; $29:List >
Matching: no matching found
Constraint 9:  (tt = set($27:NeMSet)) /\ ($30:MSet = l2ms($29:List)) /\
               $27:NeMSet = $28:Nat $30:MSet
```

If the term to be abstracted has several occurrences, this command will display the different available alternatives, allowing the user to select one by using the ***select va*** command. In the present case, however, only one abstraction is possible, and therefore ***va*** is applied automatically.

Finally, the last remaining pattern is discarded using the newly introduced ***substitution*** command, which finishes the proof of deadlock freedom for `QLOCK`:

```
DM-Check> substitution to 9 on $27:NeMSet with $28:Nat $30:MSet .

    Subsumption satisfied.
```

5 Generating Invariants by Deductive Model Checking

The new features of DM-Check, including generalization, make it possible to interactively generate an inductive invariant by deductive model checking. We present our methodology for generating invariants using DM-Check and illustrate it through a case study of Lamport's bakery protocol.

5.1 Fixpoint Generation

Our methodology is based on the folding narrowing search algorithm [17], which is adapted from narrowing-based model checking [3,10] and extended with constrained narrowing [14]. Given an initial pattern disjunction $u_1 \mid \varphi_1 \vee \cdots \vee u_n \mid \varphi_n$, the algorithm iteratively computes pattern disjunctions P_d and F_d—where F_d is called the *frontier* of P_d—for each depth $d \in \mathbb{N}$:

(1) $P_0 = F_0 = u_1 \mid \varphi_1 \vee \cdots \vee u_n \mid \varphi_n$.
(2) $P_{d+1} = P_d \vee F_{d+1}$, where F_{d+1} is the disjunction of the patterns obtained from F_d by one step of constrained narrowing and not subsumed by P_d.[7]

[7] Formally, $F_{d+1} = \{(w \mid \phi) \mid (\exists i)\ 1 \leqslant i \leqslant k, (u'_i \mid \varphi'_i \rightsquigarrow_{R,B_\Omega} w \mid \phi) \wedge (w \mid \phi) \not\sqsubseteq^{\sigma}_{B_\Omega} P_d\}$, where $F_d = \bigvee_{i=1,\ldots,k} u'_i \mid \varphi'_i$ and σ is a terminating subsumption proof strategy [17].

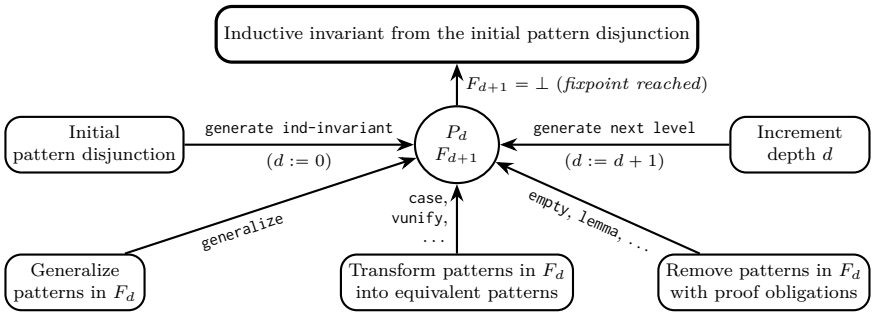


Fig. 1. Invariant generation by deductive model checking in DM-Check.

The algorithm finds an inductive invariant when it reaches a *fixpoint*. By construction, F_{d+1} *excludes* all patterns subsumed by P_d. Hence, if for some depth d every pattern obtained from F_d by constrained narrowing is subsumed by P_d, then $F_{d+1} = \varnothing$ and thus the algorithm reaches a fixpoint (i.e., $P_{d+1} = P_d$). In this case, P_d is an inductive invariant from the initial pattern [17], and captures all states reachable from the initial pattern $u_1 \mid \varphi_1 \vee \cdots \vee u_n \mid \varphi_n$.

In practice, such a fully automatic algorithm may not work for several reasons: (i) the initial pattern disjunction *may not be general enough* to reach a fixpoint, for example by starting with a disjunction of *ground* patterns; (ii) the algorithm may fail to reach a fixpoint at finite depth (looping) because some increasingly more complex patterns are generated that cannot be subsumed by other patterns; (iii) such looping may take place even though a fixpoint could have been reached at finite depth if actual set-theoretic containment could have been checked; this can happen because subsumption checking is only a *sufficient* condition for set-theoretic containment, and is further limited by the use of a fixed, terminating theorem proving strategy σ for checking implications between constraints.

These challenges for reaching a fixpoint can be addressed by providing not only an *automated* mode (which may certainly succeed in some cases), but also an *interactive* mode of deductive model checking depicted in Figure 1, where automated reachability analysis can be combined with deductive methods to: (i) *generalizing* patterns for over-approximation, (ii) proving *semantic equivalence* between pattern disjunctions; and (iii) proving *inductive lemmas* about implications between constraints and about constraint unsatisfiability.

To support methods (i)–(iii) in an interactive mode of deductive model checking we integrate the folding narrowing search mechanism with interactive commands in DM-Check for reasoning about pattern disjunctions. The commands `generate ind-invariant` and `generate next level` control the overall search, while the commands explained in Sections 3 and 4 support pattern generalization, pattern transformation, and proof obligation generation. Finally, NuITP can be used to discharge any remaining proof obligations.

5.2 Example: Lamport's Bakery Protocol

This section illustrates our methodology for generating invariants by deductive model checking, using a Maude specification of Lamport's bakery protocol [10,3]. In our previous work [2], DM-Check is used to verify the mutual exclusion of the bakery protocol for an unbounded number of processes; however, the full invariant is provided manually. In contrast, this section shows how an inductive invariant can be obtained interactively using the new features of DM-Check.

In the bakery protocol, multiple processes (or customers) compete to enter the critical section. Each process receives a ticket number to indicate that it is waiting, and the process with the smallest ticket number is served. Note that there are two sources of infiniteness: (i) due to the unbounded number of processes, the number of initial states is infinite; and (ii) due to unbounded ticket numbers, for each initial state, the number of reachable states is infinite.

Rewriting Logic Specification. A process is represented as a term of the form $[md]$, where its mode md is *idle* (not yet picked a number), $wait(j)$ (waiting with number j), or $crit(j)$ (being served with number j). A top-level state has the form $n \mid m \mid [md_1]\,;\cdots;\,[md_k]$, where n is the current number in the bakery's number dispenser, m is the number currently being served, and $[md_1]\,;\cdots;\,[md_k]$ is a semicolon-separated set of customer processes.

A set of processes is represented as a term of sort `ProcSet`—with two subsorts, `ProcIdleSet`, representing a set of idle processes, and `ProcWaitSet`, representing a set of idle or waiting processes—using an associative-commutative constructor `_;_` with identity element `none`. Similarly, natural numbers are represented as terms of sort `iNat`—with subsort `iNzNat` for non-zero natural numbers—using an associative-commutative constructor `_+_` with identity element 0.

We also define additional constructors and functions for specifying invariant properties. A set of ticket numbers is represented as a term of sort `MSet`, where an element with number n has the form `e(`n`)` and the empty set is `null`. The function `ticket(`*procs*`)` returns the set of ticket numbers occurring in a given set of processes *procs*. The function $i(n, m)$ returns the set of natural numbers $\{\, k \mid n \leqslant k < m \,\}$. For full details about the specification of `Bakery` see `https://safe-tools.dsic.upv.es/dmc/examples/bakery.html`.

The following rewrite rules define the behavior of the bakery protocol, where `PS` is a variable of sort `ProcSet`, and `N` and `M` are variables of sort `iNat`:

```
rl [wake] : N | M | [idle]    ; PS  =>  1 + N |     M | [wait(N)] ; PS .
rl [crit] : N | M | [wait(M)] ; PS  =>      N |     M | [crit(M)] ; PS .
rl [exit] : N | M | [crit(M)] ; PS  =>      N | 1 + M | [idle]    ; PS .
```

Initially, the two counters are equal and all processes are idle, as captured by the constrained pattern `(N | N | IS) | true`, where `IS` is a variable of sort `ProcIdleSet`. The error states violating the mutual exclusion property are described by the following constrained pattern, where `K` and `L` are variables of sort `iNat`: `(M | N | [crit(K)] ; [crit(L)] ; PS) | true`.

Interactive Deductive Model Checking. We begin by symbolically exploring the state space from the initial pattern using the `generate ind-invariant` command. As expected, the initial pattern is not inductive and DM-Check reports a successor pattern (some parts of the output are omitted and replaced with ...):

```
DM-Check> generate ind-invariant (N:iNat | N:iNat | IS:ProcIdleSet) | true .

    Invariant could not be proved.

Parent 1
Term:  $2:iNat | $2:iNat | $1:ProcIdleSet              Constraint:  true

Child 2 (Parent 1)
Term:  (1 + $4:iNat) | $4:iNat | $3:ProcIdleSet ; [wait($4:iNat)]
...
Matching: no matching found                            Constraint:  true
```

We continue the symbolic exploration by running the `generate next level` command. The child nodes then become parent nodes, and new child nodes are generated by constrained narrowing:

```
DM-Check> generate next level .

    Invariant could not be proved.

Parent 1
Term:  $2:iNat | $2:iNat | $1:ProcIdleSet              Constraint:  true

Parent 2
Term:  (1 + $4:iNat) | $4:iNat | $3:ProcIdleSet ; [wait($4:iNat)]
Constraint:  true

Child 3 (Parent 2)
Term:  (1 + 1 + $5:iNat) | $5:iNat | $6:ProcIdleSet ;
       [wait($5:iNat)] ; [wait(1 + $5:iNat)]           ...
Matching: no matching found                            Constraint:  true

Child 4 (Parent 2)
Term:  (1 + $7:iNat) | $7:iNat | $8:ProcIdleSet ; [crit($7:iNat)]
...
Matching: no matching found                            Constraint:  true
```

Further executions of this command do not lead to a fixpoint, but produce many patterns similar to Child 3, containing waiting processes with ticket numbers greater than or equal to the second counter but less than the first counter.[8] Based on this observation, we generalize Child 3 using the `generalize` command as follows (the output is the same as above except for Child 3):

[8] Typically, this command is executed multiple times to generate a variety of constrained patterns, allowing the user to identify generic patterns. Due to space limitations, we only show the result of a single execution of `generate next level`.

```
DM-Check> generalize to 3 using \
      (M:iNat + N:iNat | N:iNat | WS:ProcWaitSet) | \
             (tickets(WS:ProcWaitSet) = i(N:iNat, M:iNat + N:iNat)) .

    Invariant could not be proved.
...

Child 3 (Parent 2)
Term:  ($9:iNat + $10:iNat) | $10:iNat | $11:ProcWaitSet
...
Matching: no matching found
Constraint:  tickets($11:ProcWaitSet) = i($10:iNat, $9:iNat + $10:iNat)

...
```

A couple of executions of the `generate next level` command yield the following output. Only two parent nodes remain: all other parent nodes are subsumed by these nodes and therefore omitted from the output.

```
DM-Check> generate next level .
...
DM-Check> generate next level .

    Invariant could not be proved.

Parent 1
Term:  ($3:iNat + $1:iNat) | $1:iNat | $2:ProcWaitSet
Constraint:  tickets($2:ProcWaitSet) = i($1:iNat, $3:iNat + $1:iNat)

Parent 2
Term:  ($4:iNat + $5:iNat) | $4:iNat | $6:ProcWaitSet ; [crit($4:iNat)]
Constraint:  tickets($6 ; [wait($4)]) = i($4, $5 + $4)

Child 4 (Parent 2)
Term:  ($10:iNat + $11:iNat) | 1 + $10:iNat | $12:ProcWaitSet ; [idle]
...
Matching: no matching found
Constraint:  tickets($12 ; [wait($10)]) = i($10, $11 + $10)

Child 5 (Parent 2)
Term:  ($13:iNat + $14:iNat) | $13:iNat | $15:ProcWaitSet ;
       [crit($13:iNat)] ; [crit($13:iNat)]              ...
Matching: no matching found
Constraint:  tickets(($15 ; [wait($13)]) ; [wait($13)]) = i($13, $14 + $13)
```

Two child nodes are not subsumed by the parent nodes. The constraint of Child 5 is unsatisfiable, since $\mathtt{i}(\$19, \$20 + \$19)$ cannot contain duplicate numbers. The pattern of Child 4 is not sufficiently instantiated; so we split on whether `$11:iNat` is zero or non-zero. We therefore run the following commands:

```
DM-Check> empty 5 .
...
DM-Check> case to 4 on $11:iNat with 0 ;; 1 ;; (1 + #N:iNzNat) .

    Invariant could not be proved.
...

Child 5 (Parent 2)
Term:  $13:iNat | 1 + $13:iNat | $14:ProcWaitSet ; [idle]
...             Matching: no matching found
Constraint:  (e($13:iNat), tickets($14:ProcWaitSet)) = null

Child 6 (Parent 2)
Term:  (1 + $15:iNat) | 1 + $15:iNat | $16:ProcWaitSet ; [idle]
...             Matching parent #1: 1           ...
NuITP implication #1 (open):    ...

Child 7 (Parent 2)
Term:  (1 + $18:iNat + $17:iNzNat) | 1 + $18:iNat | $19:ProcWaitSet ; [idle]
...             Matching parent #1: 1           ...
NuITP implication #1 (open):    ...
```

The `case` command generates three new child nodes. We can easily see that Child 5 has a false constraint, and Children 6 and 7 have valid open implications. Thus, we run the `empty` and `lemma` commands as follows.

```
DM-Check> empty 5 .
...
DM-Check> lemma 6 .
...
DM-Check> lemma 7 .

    The following Invariant is satisfied

  ((($3:iNat + $1:iNat) | $1:iNat | $2:ProcWaitSet) |
    (tickets($2) = i($1, $3 + $1))) \/
  ((($4:iNat + $5:iNat) | $4:iNat | $6:ProcWaitSet ; [crit($4:iNat)]) |
    (tickets($6 ; [wait($4)]) = i($4, $5 + $4)))

    with the following proof obligations:.

  (e($10:iNat), tickets($12:ProcWaitSet)) = e($10:iNat + #N:iNzNat),
    i($10:iNat, $10:iNat + #N:iNzNat) -> tickets($12:ProcWaitSet ; [idle]) =
    i(1 + $10:iNat, #N:iNzNat + 1 + $10:iNat)
  (e($10:iNat), tickets($12:ProcWaitSet)) = e($10:iNat) ->
    tickets($12:ProcWaitSet ; [idle]) = i(1 + $10:iNat, 0 + 1 + $10:iNat)
  (e($19:iNat), tickets($20:ProcWaitSet)) = null -> false
  tickets(($14:ProcWaitSet ; [wait($15:iNat)]) ; [wait($13:iNat)]) =
    i($13:iNat, $14:iNat + $15:iNat) -> false
```

The output of the last command indicates that an inductive invariant is successfully generated, with four proof obligations remaining. All of the proof obligations can be discharged using NuITP (see `https://safe-tools.dsic.upv.es/dmc/examples/bakery.html`).

Proving Other Invariants. We can use the generated invariant to prove other invariants from the initial pattern. For example, the following command shows that, in each reachable state, the ticket numbers range from the second counter to the first counter (including the second but excluding the first):

```
DM-Check> check ($3:iNat + $1:iNat | $1:iNat | $2:ProcWaitSet) | \
        tickets($2:ProcWaitSet) = i($1:iNat, $3:iNat + $1:iNat) \/ \
    ($4:iNat + $5:iNat | $4:iNat | $6:ProcWaitSet ; [crit($4:iNat)]) | \
        tickets($6:ProcWaitSet ; [wait($4:iNat)]) = \
        i($4:iNat, $5:iNat + $4:iNat) \
  subsumed by (M:iNat + N:iNat | N:iNat | PS:ProcSet) | \
        tickets(PS:ProcSet) = i(N:iNat, N:iNat + M:iNat) .

    Subsumption satisfied.
```

The following command verifies that the generated invariant does not intersect with the error states that violate the mutual exclusion property. This proves that all states reachable from the initial pattern satisfy the mutual exclusion property, since the generated invariant includes all such reachable states.

```
DM-Check> intersect ($3:iNat + $1:iNat | $1:iNat | $2:ProcWaitSet) | \
        tickets($2:ProcWaitSet) = i($1:iNat, $3:iNat + $1:iNat) \/ \
    ($4:iNat + $5:iNat | $4:iNat | $6:ProcWaitSet ; [crit($4:iNat)]) | \
        tickets($6:ProcWaitSet ; [wait($4:iNat)]) = \
        i($4:iNat, $5:iNat + $4:iNat) \
  with \
    (M:iNat | N:iNat | [crit(K:iNat)] ; [crit(L:iNat)] ; PS:ProcSet) | true .

    No intersection.
```

6 Conclusions and Future Work

Up to now the deductive model checking methodology in [1,2] and its associated DM-Check tool were only applied to proving inductive invariants. In model checking terms, this amounts to the special case of *bounded* model checking with depth bound 1. Even for this subcase, the inference rules available for proving semantic equivalences between disjunctions of constrained patterns were limited, so that examples such as QLOCK could not be verified. In this work, this deductive model checking methodology has been significantly extended by: (i) new inference rules for proving such semantic equivalences; (ii) a generalization command for over-approximating sets of states; (iii) an *unbounded* deductive model checking

methodology to generate inductive invariants; and (iv) a new version of DM-Check supporting (i)—(iii). The usefulness of this extended methodology has been illustrated by means of the QLOCK and BAKERY examples.

Much work remains ahead. Firstly, further experimentation with other case studies and improvements to DM-Check should be pursued to make it more scalable. Secondly, *further automation* of this deductive model checking methodology should be investigated; for example by: (i) using more powerful terminating strategies in the **NuITP** to significantly increase the number of constrained patterns that can be automatically folded into previous ones; and (ii) applying some inference rules of DM-Check *automatically* by means of a *strategy*, e.g., a strategy could apply the `generalize` rule when some "looping" of constrained patterns is detected.

Acknowledgements S. Escobar, R. López-Rueda and J. Sapiña have been supported by the grant CIPROM/2022/6 funded by Generalitat Valenciana, and by the grant PID2024-162030OB-100 funded by MCIN/AEI/10.13039/501100011033 and ERDF A way of making Europe. K. Bae, S. Escobar, J. Sapiña, and R. López-Rueda have been supported by the NATO Science for Peace and Security Programme project SymSafe (grant number G6133). K. Bae was supported by the National Research Foundation of Korea (NRF) grant (No. RS-2021-NR060080) and the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant (No. RS-2024-00439856), both funded by the Korea government (MSIT).

References

1. Bae, K., Escobar, S., López-Rueda, R., Meseguer, J., Sapiña, J.: Verifying invariants by deductive model checking. In: Proceedings of the 15th International Workshop on Rewriting Logic and its Applications (WRLA 2024). Lecture Notes in Computer Science, vol. 14953, pp. 3–21. Springer (2024). https://doi.org/10.1007/978-3-031-65941-6_1
2. Bae, K., Escobar, S., López-Rueda, R., Meseguer, J., Sapiña, J.: DM-Check: Verifying invariants of concurrent systems by deductive model checking. Journal of Logical and Algebraic Methods in Programming **149**, 101107 (2026). https://doi.org/10.1016/j.jlamp.2025.101107
3. Bae, K., Escobar, S., Meseguer, J.: Abstract logical model checking of infinite-state systems using narrowing. In: Proceedings of the 24th International Conference on Rewriting Techniques and Applications (RTA 2013). Leibniz International Proceedings in Informatics (LIPIcs), vol. 21, pp. 81–96. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2013). https://doi.org/10.4230/LIPIcs.RTA.2013.81
4. Bae, K., Meseguer, J.: Infinite-state model checking of LTLR formulas using narrowing. In: Proceedings of the 10th International Workshop on Rewriting Logic and its Applications (WRLA 2014). Lecture Notes in Computer Science, vol. 8663, pp. 113–129. Springer (2014). https://doi.org/10.1007/978-3-319-12904-4_6
5. Bae, K., Meseguer, J.: Predicate abstraction of rewrite theories. In: RTA-TLCA. Lecture Notes in Computer Science, vol. 8560, pp. 61–76. Springer (2014)

6. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.: Equational unification and matching, and symbolic reachability analysis in Maude 3.2 (system description). In: Proceedings of the 11th International Joint Conference on Automated Reasoning (IJCAR 2022). Lecture Notes in Computer Science, vol. 13385, pp. 529–540. Springer (2022). https://doi.org/10.1007/978-3-031-10769-6_31
7. Durán, F., Escobar, S., Meseguer, J., Sapiña, J.: NuITP alpha 30 – an inductive theorem prover for Maude equational theories (May 2024), available at https://nuitp.webs.upv.es/
8. Durán, F.J., Escobar, S., Meseguer, J., Sapiña, J.: NuITP: An inductive theorem prover for equational program verification. In: Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming, PPDP 2024, Milano, Italy, September 9-11, 2024. pp. 6:1–6:11. Association for Computing Machinery (2024). https://doi.org/10.1145/3678232.3678236
9. Escobar, S., López-Rueda, R., Sapiña, J.: Symbolic analysis by using folding narrowing with irreducibility and SMT constraints. In: Proceedings of the 9th International Workshop for Safety-Critical Systems (FTSCS 2023). pp. 14–25. Association for Computing Machinery (2023). https://doi.org/10.1145/3623503.3623537
10. Escobar, S., Meseguer, J.: Symbolic model checking of infinite-state systems using narrowing. In: Proceedings of the 18th International Conference on Term Rewriting and Applications (RTA 2007). Lecture Notes in Computer Science, vol. 4533, pp. 153–168. Springer (2007). https://doi.org/10.1007/978-3-540-73449-9_13
11. Escobar, S., Sasse, R., Meseguer, J.: Folding variant narrowing and optimal variant termination. The Journal of Logic and Algebraic Programming **81**(7–8), 898–928 (2012). https://doi.org/10.1016/j.jlap.2012.01.002
12. Futatsugi, K.: Advances of proof scores in CafeOBJ. Science of Computer Programming **224**, 102893 (2022). https://doi.org/10.1016/j.scico.2022.102893
13. McMillan, K.L., Padon, O.: Ivy: A multi-modal verification tool for distributed algorithms. In: Computer Aided Verification. pp. 190–202. Springer LNCS 12225 (2020)
14. Meseguer, J.: Generalized rewrite theories, coherence completion, and symbolic methods. Journal of Logical and Algebraic Methods in Programming **110** (2020). https://doi.org/10.1016/j.jlamp.2019.100483
15. Meseguer, J.: Inductive reasoning with equality predicates, contextual rewriting and variant-based simplification. Journal of Logical and Algebraic Methods in Programming **144**, 101036 (2025). https://doi.org/10.1016/J.JLAMP.2025.101036
16. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)
17. Meseguer, J.: Symbolic computation and verification methods in maude. In: Escobar, S., Titolo, L. (eds.) Logic-Based Program Synthesis and Transformation - 35th International Symposium, LOPSTR 2025, Rende, Italy, September 9-10, 2025, Proceedings. Lecture Notes in Computer Science, vol. 16117, pp. 1–21. Springer (2025). https://doi.org/10.1007/978-3-032-04848-6_1, https://doi.org/10.1007/978-3-032-04848-6_1
18. Ogata, K., Futatsugi, K.: Proof scores in the OTS/CafeOBJ method. In: Formal Methods for Open Object-Based Distributed Systems, 6th IFIP WG 6.1 International Conference, FMOODS 2003, Paris, France, November 19.21, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2884, pp. 170–184. Springer (2003). https://doi.org/10.1007/978-3-540-39958-2_12

19. Ogata, K., Futatsugi, K.: Theorem proving based on proof scores for rewrite theory specifications of OTSs. In: Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi. Lecture Notes in Computer Science, vol. 8373, pp. 630–656. Springer (2014)
20. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 614–630. PLDI '16, ACM (2016)
21. Rocha, C., Meseguer, J.: Mechanical analysis of reliable communication in the alternating bit protocol using the Maude invariant analyzer tool. In: Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi. Lecture Notes in Computer Science, vol. 8373, pp. 603–629. Springer (2014)
22. Tran, D.D., Ogata, K.: IPSG: Invariant proof score generator. In: 46th IEEE Annual Computers, Software, and Applications Conferenc, COMPSAC 2022, Los Alamitos, CA, USA, June 27 - July 1, 2022. pp. 1050–1055. IEEE (2022). https://doi.org/10.1109/COMPSAC54236.2022.00164

Equational and Inductive Reasoning for Maude in Athena

Mateo Sanabria[1][0000−0003−3407−9792], Carlos Varela[2][0000−0003−4708−3109], Camilo Rocha[3][0000−0003−4356−7704], and Nicolás Cardozo[1][0000−0002−1094−9952]

[1] Universidad de los Andes, Bogotá , Colombia
[2] Rensselaer Polytechnic Institute, Troy, USA
[3] Pontificia Universidad Javeriana, Cali, Colombia

Abstract. In the rewriting logic framework, equational-based specifications are used to define deterministic functional behavior, abstract data types, and canonical representations of data. These specifications include a (possibly order-sorted) signature and equations interpreted modulo structural axioms, such as associativity, commutativity, and identity. While equational rewriting provides a powerful basis for execution and symbolic reasoning, it does not by itself offer native support for inductive or deductive reasoning. This paper presents maude2athena, a framework that systematically translates Maude's equational theories into Athena, a theorem proving language designed to support natural deduction proofs over many-sorted first-order logic specifications, including inductive reasoning, equational chaining, case-based reasoning, and proofs by contradiction. The translation supports induction-based reasoning modulo structural axioms with parametric induction rules; it faithfully encodes membership equational logic in a many-sorted setting without exponential blowup under reasonable conditions. This approach preserves the semantics of the original specification, while ensuring that the translation remains compact and amenable to deductive reasoning. This work helps bridge the gap between model checking and theorem proving, enabling formal verification efforts that can benefit from both of these approaches.

Keywords: Deductive reasoning · Inductive reasoning · Membership equational logic · Reasoning modulo axioms · Model checking and theorem proving · Maude · Athena

1 Introduction

Rewriting Logic [1] has established itself as a powerful semantic and specification framework for modeling computational systems, supporting both executable specifications and rigorous equational reasoning. Within this framework, *equational-based specifications* play a central role in defining deterministic functional behavior, abstract data types, and canonical representations of data. Languages such as Maude [2] provide strong support for this style of specification through membership equational logic, offering subsort relations, operator overloading, and rewriting modulo structural axioms such as associativity, commutativity, and identity. While rewriting logic is a logical framework that can be used for symbolic reasoning, including reasoning over its own specifications [3], Maude currently offers limited support for interactive theorem proving, especially for proofs that require explicit inductive reasoning or fine-grained proof control. In contrast, interactive theorem provers such as Athena [4] provide robust environments for natural-deduction reasoning in many-sorted first-order logic, together with strong support for induction and proof structuring. However, Athena lacks native support for the order-sorted features that are essential to many Maude specifications. As a consequence, fully-fledged Maude specifications cannot be directly reused within Athena without a careful reconciliation of their underlying logical foundations.

This paper presents maude2athena, a framework that systematically translates Maude functional modules into Athena modules, enabling equational and inductive reasoning over Maude specifications within Athena's interactive proof environment. The translation is grounded in a semantics-preserving mapping from membership equational logic to many-sorted first-order logic, in which subsort relations are made explicit through cast operators and coherence axioms [5]. To ensure scalability and avoid exponential blowup, the approach relies on the notion of strictly sensible order-sorted algebras [5, 6], which guarantees a linear and unambiguous translation in the presence of operator overloading.

A central challenge in this translation is the loss of native structural induction when order-sorted datatypes are flattened into many-sorted domains. Although this transformation preserves equational reasoning, it removes the inductive structure that is essential for proving many correctness properties. To address this issue, Athena is extended with *parametric structural induction primitives* that reconstruct the induction principles implicit in the original Maude specification. This design allows different induction schemes to be instantiated as needed, enabling inductive proofs over translated domains without resorting to encodings of operational semantics or meta-level reasoning. In this paper, this parametric framework is instantiated with structural induction for sufficiently complete equational specifications [7]; other induction principles can be accommodated as well, such as structural induction over the oriented equations of a terminating equational theory.

By bridging executable equational specifications in Maude with interactive inductive proofs in Athena, maude2athena enables a compositional workflow in which specification, execution, and proof coexist within a single formal development. The proposed translation preserves the equational theory of the original specification while remaining compact and amenable to both interactive and automated reasoning. The approach is validated through theoretical results establishing the correctness of the translation, as well as through a non-trivial case study involving the verification of a compiler specification that relies heavily on subsorting and inductive reasoning.[4]

2 Preliminaries

This section presents an overview on rewriting logic in Maude, and theorem proving in Athena.

2.1 Rewriting Logic and Maude

Rewriting logic [1] is built on top of membership equational logic [8] as the underlying equational formalism. Maude [2] is a high-performance logical framework and specification language based on rewriting logic. Membership equational logic supports sorts, equations, and membership axioms that characterize the elements of a sort. While the full language extends these characteristics with rewrite rules to represent transitions in concurrent systems, this paper focuses on the translation of the functional (equational) sub-language.

Membership equational logic is defined over a many-kinded signature (K, F), where K is a set of kinds and $F = \{F_{w,k}\}_{(w,k)\in K^* \times K}$ is a family of function symbols typed over those kinds. In addition to kinds, the logic considers a K-indexed family of sorts $S = \{S_k\}_{k\in K}$, which allows for ascribing sorts to terms using the notation $t : s$ (term t has sort s). Thus, a complete signature in membership equational logic is a triple $\Sigma = (K, F, S)$. The sorts S are equipped with a partial order $\leq$ (subsorting), and operators in F are defined over these sorts. This view associates the

[4] The maude2athena framework is available at `https://github.com/FLAGlab/Maude2Athena`.

kinds K as the connected components of the subsort relation $\leq$. Since an order-sorted algebra can be canonically mapped to a membership algebra [1, 9], the order-sorted setting is used throughout this paper. A membership equational theory is a pair (Σ, E), combining a signature Σ with a set of sentences E (conditional equations and membership axioms). Equality modulo E, denoted $=_E$, is the finest congruence on the term algebra $T_\Sigma(X)$ that satisfies the Horn clauses in E. That is, for any terms $t, u \in T_\Sigma(X)$, $t =_E u$ iff $E \vdash t = u$. Semantic structures of equational theories are called *algebras*. The expression T_Σ is the *sorted ground term algebra* of Σ and $T_\Sigma(X)$ the *sorted term algebra* of Σ with variables in X. Likewise, the quotient structure $T_{\Sigma/E}$ is the *initial algebra* of (Σ, E) meaning that two terms $t, u \in T_\Sigma$ are in the same equivalence class iff $t =_E u$. This is naturally extended to the algebra $T_{\Sigma/E}(X)$ of terms with variables in X modulo the equations E.

Maude partitions the specification into a set of structural axioms A (such as associativity, commutativity, and identity) and a set of functional equations E'. Consequently, an equational theory is understood as $(\Sigma, E' \cup A)$, where rewriting is performed modulo the structural axioms A by orienting the equations from left to right. In Maude, functional modules correspond to membership equational theories and define data types and their operations. Equations are treated as simplification rules applied left to right. By assuming *sort-decreasingness*, *ground confluence*, and *operational termination*, repeated application of the oriented equations reduces a term to its canonical form [10, 11]. Furthermore, it is assumed that an equational theory $(\Sigma, E' \cup A)$ is *sufficiently complete* with respect to a subsignature $\Omega \subseteq \Sigma$ meaning that, for any sort $s \in S$ and term $t \in T_{\Sigma,s}$, there is a term $u \in T_{\Omega,s}$ such that $t =_E u$.

Listing 1.1 presents a functional module specification of Peano numbers. The `PEANO` module is delimited by the Maude keywords `fmod` and `endfm`. Sorts are declared with the `sort` keyword, and subsort relations use `subsorts` together with the symbol `<` (Lines 2–3). Operators are declared with `op`; each declaration specifies the operator name, its sort signature, and an optional set of attributes that declare the structural set of axioms for the equational theory. Particularly, the attribute `ctor` marks constructors for the sort. The elements of the sort `Even` are defined recursively by a conditional equation (Line 8), and the Peano axioms are introduced as equations (Lines 9–10) defining the set E for the equational theory.

```
1  fmod PEANO is
2    sort NzNat Even Nat .
3    subsorts NzNat Even < Nat .
4    op zero : -> Even [ctor] .
5    op s_ : Nat -> NzNat [ctor] .
6    op _+_ : Nat Nat -> Nat .
7    vars N M : Nat .
8    cmb s s N : Even if N : Even .
9    eq N + zero = N .
10   eq N + s M = s (N + M) .
11 endfm
```

Listing 1.1. Definition of Peano numbers in Maude.

2.2 Theorem Proving in Athena

Athena [4] is a dual deduction and computation language: users can define *procedures*, which abstract over computations (i.e., lambda calculus abstractions in functional programming); and *methods*, which abstract over Fitch-style natural deductions [12, 13]. When evaluating procedures, if

successful, Athena can return values (of different types), or else, it can diverge or result in an error. When evaluating methods, if successful, Athena produces logically sound theorems—modulo its assumption base—as sentences in many-sorted first order logic [14]. Athena has been effectively used to reason about concurrent programming—particularly, the actor model [15, 16], and about safety-critical cyber-physical systems [17, 18].

Athena's proofs have a natural deduction style. Logical sentences can either be asserted into the assumption base or proven as theorems using Athena's deduction tools, with a soundness guarantee that any proven theorem is a logical consequence of sentences in the assumption base. Athena performs automatic sort-checking to prevent ill-sorted expressions in specifications and allows theorems to be introduced at an abstract level by encapsulating proofs in parameterized methods, which can be instantiated to prove different specializations of the abstract theorems. It defines two syntactic categories: an *expression*, which represents a computation, and a *deduction*, a logical argument such as a proof. A valid deduction always concludes with a *sentence*, *i.e.,* the conclusion of the proof.

Sorts form the foundation of Athena's deductive language, classifying all terms in its sentences. Sorts can be defined as *domains*, *structures*, and *datatypes*, each providing different mechanisms for constructing elements and each assuming different primitive axioms.

Domains are sorts that describe the sets of objects to be modeled. A *datatype* is a special kind of domain that is *inductively generated*, meaning that every element of the domain can be built up in a finite number of steps by applying *constructors* of the datatype. Beyond providing syntactic convenience over domains, datatypes can be assumed to follow *free-generation axioms* for their elements, ensuring that different constructor applications create different elements (*no-confusion* axioms), and that every element is represented by some constructor application (*no-junk* axioms).

Structures, just like regular datatypes, are inductively generated by their constructors. The only difference is that the constructors might not be injective, so that the same constructor applied to two distinct sequences of arguments might result in the same value.

The keyword `domain` introduces domains in Athena. Listing 1.2 presents a definition for Peano numbers based on three domains: `Nat, Even, NzNat`, with function symbols `zero,S,plus`, defined using the `declare` instruction. Note that, alternatively, Peano numbers can be defined using datatypes as: `datatype Nat := zero | (s Nat)`.

```
1 domains Nat, Even, NzNat
2 declare zero:[]         -> Even
3 declare s:   [Nat]      -> NzNat
4 declare plus:[Nat Nat] -> Nat [+]
5 define  [n m]            := [?n:Nat ?m:Nat]
6 assert  Plus-zero-axiom := (forall n .  (zero + n) = n)
7 assert* Plus-s-axiom    := ((n + (s m)) = (s (n + m)))
```

Listing 1.2. Peano example definition using domains.

The datatype definition provides free-generation axioms, ensuring that different constructor applications yield distinct elements. In contrast, the domain based approach offers more expressiveness by allowing sort refinements, like `Even` and `NzNat`, which enable more precise sort specifications. The version using domains is inspired by languages with subsort capabilities, such as Maude, and leverages Athena's many-sorted foundation to provide fine-grained sort control.

The semantics of Athena defines how phrases F (expressions or deductions) are evaluated within an *evaluation context*, a four-tuple $\langle \rho, \beta, \sigma, \gamma \rangle$ representing the logical and computational environment in which expressions are interpreted, where:

- ρ is the execution environment, a computable function that maps any given identifier I either to a value V or to a special *unbound* token.
- β is the assumption base, a finite set of sentences.
- σ is the store, a computable function that maps any natural number (representing a memory location) to a value (the location's contents) or to a special *unassigned* token.
- γ is a set of symbols with their associated signatures, and a collection of sort constructors (with arities). γ also includes information on whether a given sort constructor is a datatype or structure, and if so, which function symbols are its constructors.

The result of evaluating a phrase F in an evaluation context $\langle \rho, \beta, \sigma, \gamma \rangle$ is one of the following:

1. A pair (V, σ') consisting of a value V and a store σ', where V is the output of the evaluation and σ' reflects any side effects accumulated during the evaluation.
2. A pair consisting of an error message and a store σ', indicating the occurrence of an error during the computation.
3. Nontermination.

Sentences are added into the assumption base using `assert` or `assert*`, where `assert*` uses implicit universal quantification for all free variables, before inserting the sentence into the assumption base. Lines 6–7 in Listing 1.2 add the properties of Peano numbers into the assumption base.

Once sentences are introduced into the assumption base, they can be used in deductive reasoning. Further, the datatypes and structures definitions allow deductions of theorems that are not derivable from the free-generation axioms alone, by using the `by-induction` method. For example, assuming that `Nat` follows a datatype definition, Listing 1.3 displays the induction proof for the associativity property for the plus operator, Line 2.

In general, the method `by-induction` requires the proof of the property for all base cases (*zero* in the case of Peano, Lines 5–8) and the proof for all inductive definitions (S for `Nat` datatype, Lines 9–16). Additionally, this induction proof showcases one of the most useful Athena methods: `chain` (Lines 6–8 and Lines 12–16). `chain` allows equational chaining based proofs, where each step is labeled with its justification.

```
1  define [p q r] := [?p:Nat ?q:Nat ?r:Nat]
2  define plus_associative := (forall p q r . ((q + r) + p) = (q + (r + p)))
3  conclude plus_associative
4  by-induction plus_associative {
5     zero =>  pick-any q r
6              (!chain [((q + r) + zero)
7                       --> (q + r)          [Plus-zero-axiom]
8                       <-- (q + (r + zero)) [Plus-zero-axiom])
9     | (s p) => let {induction-hypothesis :=
10                   (forall ?q ?r . (?q + ?r) + p = ?q + (?r + p))}
11             pick-any q r
12             (!chain [ ((q + r) + (s p))
13                       --> (s ((q + r) + p))   [Plus-s-axiom]
14                       --> (s ((q + (r + p)))) [induction-hypothesis]
15                       <-- (q + (s (r + p)))   [Plus-s-axiom]
16                       <-- (q + (r + (s p)))  ])
17 }
```

Listing 1.3. Deduction of associativity property using datatype induction.

3 From Maude to Athena

At the heart of the proposed approach, lies the need to translate the order-sorted structure of terms of an equational theory in Maude to the many-sorted setting of equational theories in Athena. This section presents such a translation as a function from Maude modules to Athena theories. Translation of membership (of terms in sorts) is accomplished by using predicates in Athena.

3.1 Strictly Sensible Order-Sorted Signatures

The translation provides a linear translation of *strictly sensible* algebras to many-sorted algebras [5]. The key idea of the translation is to add an equivalence relation called core equality to the translated many-sorted structures. By defining this relation, the complexity of translating a strictly sensible order-sorted algebra to a many-sorted one is reduced and the translated many-sorted algebra equations only increase by a very small amount of new equations. Algebras definitions are lifted to signatures to keep the presentation of their results concise.

Fix a membership signature $\Sigma = (K, F, S)$, with kinds K, function symbols F, and sorts S, and a poset $(S, \leq)$ over the sorts. For a function symbol $g : s_1 \times \cdots \times s_n \to s$, $\text{target}(g) = s$ denotes its result sort. In what follows, assume $f, f', f'' \in F$ and, for a fixed $n \in \mathbb{N}$ and $s_i, s_i', s_i'' \in S$ for $0 \leq i < n$, where $f : s_0 \times \cdots \times s_n \to s$; $f' : s_0' \times \cdots \times s_n' \to s'$; $f'' : s_0'' \times \cdots \times s_n'' \to s''$.

Argument compatibility captures the idea of overloaded function symbols that can be treated uniformly as their pair-wise argument positions share a common supersort. Strongly sensible signatures have argument compatible function symbols in which the target sorts are the same.

Definition 1. *Two overloaded function symbols f and f' are called argument compatible, denoted $ac(f, f')$, iff $(\forall\, i \mid 0 \leq i \leq n \ : s_i \equiv_{\leq} s_i')$, where $s_i \equiv_{\leq} s_i'$ is a predicate expressing that s_i and s_i' are in the same connected component of $(\Sigma, \leq)$. The signature Σ is strongly sensible iff $(\forall\, f, f' \mid ac(f, f') \ : s = s')$.*

On the other hand, maximal argument-bounding signatures ensure that every function symbol has a representative among its compatible function symbols, thereby defining one representative for each class of argument compatible function symbols.

Definition 2. *The signature Σ is maximal argument-bounding iff:*

$$(\forall\, f \mid (\exists\, f'' \mid (\forall\, f' \mid ac(f, f') \ : (\forall\, i \mid 0 \leq i \leq n \ : s_i' \leq s_i'')))).$$

A *strictly sensible* signature is a signature that is both strongly sensible and maximal argument-bounding. Strictly sensible signatures play a central role in the translation from order-sorted equational theories in Maude to many-sorted theories in Athena. It rules out ambiguous cases of function symbol overloading and ensures that the translation remains well-defined and systematic. In contrast to more general translations (e.g., [19]), which can produce non-linear or ambiguous mappings in the presence of unrestricted overloading, requiring strict sensibility guarantees that every function symbol has a single, consistent representative in the translated signature. In particular, strict sensibility ensures that each function symbol has a unique representative in the translated signature, allowing the order-sorted algebra to be flattened into a many-sorted representation without loss of soundness or consistency.

In the rest of the paper, it is assumed that any signature Σ is strictly sensible.

3.2 The Translation Function

Let $\mathcal{E} = (\Sigma, E \cup A)$ be a membership equational theory, with signature (K, F, S) and poset of sorts $(S, \leq)$, and $\Omega \subseteq \Sigma$ be a constructor subsignature for $\mathcal{E}$. This section defines the mapping $\mathcal{E} \mapsto (\beta, \gamma)$ as the function *tr*, where β is an Athena assumption base and γ is its symbol set. The ρ execution environment and σ store are omitted from the mapping because they are orthogonal to the generated Athena representation. The behavior of *tr* is driven primarily by the subsort relation $\leq$ and by the constructors Ω.

Definition 3 (Translation Function tr**).** *Let $\mathcal{E} = (\Sigma, E \cup A)$ be a membership equational theory with signature $\Sigma = (K, F, S)$, sort poset $(S, \leq)$, and constructor subsignature $\Omega \subseteq \Sigma$. The function $\mathrm{tr} : \mathcal{E} \to (\beta, \gamma)$ is defined by the following five components:*

(i) Sort translation (**tr_S**). *For each sort $s \in S$:*

$$\mathrm{tr}_S(s) = \begin{cases} \texttt{datatype}(s,\, \{c \in \Omega \mid \mathrm{target}(c) = s\}) & \text{if } s \notin \mathrm{dom}(\leq) \cup \mathrm{ran}(\leq) \text{ and } \exists\, c \in \Omega. \\ \texttt{domain}(s) & \text{otherwise} \end{cases}$$

(ii) Function symbol translation (**tr_F**). *Let $[f]_{\mathrm{ac}}$ denote the argument compatibility class of $f \in F$. For each class $[f]_{\mathrm{ac}}$, select the maximal representative f^* (guaranteed to exist by strict sensibility), where $f^* \colon s_1 \times \cdots \times s_n \to s$, and set*

$$\gamma \;\cup\; \left\{\, f^* : [\mathrm{tr}_S(s_1) \cdots \mathrm{tr}_S(s_n)] \to \mathrm{tr}_S(s) \,\right\} \;\mapsto\; \gamma.$$

For each $s \leq s'$ in $(S, \leq)$, set

$$\gamma \;\cup\; \left\{\, \mathrm{Cast}_{s \to s'} : [\mathrm{tr}_S(s)] \to \mathrm{tr}_S(s') \,\right\} \;\mapsto\; \gamma.$$

Additionally, for each $f^* : s \times s \to s$ in γ with structural attributes in A, let x, y, z be fresh variables of sort $\mathrm{tr}_S(s)$ and set

$$\beta \;\cup\; \left\{ \begin{array}{ll} f^*(f^*(x,y),z) = f^*(x, f^*(y,z)) & \text{if } \texttt{assoc} \in \mathrm{attr}(f), \\ f^*(x,y) = f^*(y,x) & \text{if } \texttt{comm} \in \mathrm{attr}(f), \\ f^*(\mathrm{tr}_T(e), x) = x,\; f^*(x, \mathrm{tr}_T(e)) = x & \text{if } \texttt{id:}e \in \mathrm{attr}(f) \end{array} \right\} \;\mapsto\; \beta.$$

(iii) Term translation (**tr_T**). *For each $t \in T_\Sigma(X)$:*

$$\mathrm{tr}_T(t) = \begin{cases} x & \text{if } t = x \in X \\ f^*(\mathrm{tr}_T(t_1), \ldots, \mathrm{tr}_T(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

where f^* is the maximal representative of $[f]_{\mathrm{ac}}$ and, at each argument position i where the subterm has sort s_i' with $s_i' \leq s_i$ and $s_i' \neq s_i$, $\mathrm{tr}_T(t_i)$ is wrapped in the appropriate cast chain $\mathrm{Cast}_{s_j \to s_{j+1}}(\cdots(\mathrm{tr}_T(t_i))\cdots)$ so that the result has sort $\mathrm{tr}_S(s_i)$.

(iv) Equation translation (**tr_E**). *For each equation $l = r$ in $E \cup A$, set*

$$\beta \;\cup\; \left\{\, \mathrm{tr}_T(l) = \mathrm{tr}_T(r) \,\right\} \;\mapsto\; \beta.$$

For each composable pair $s \leq s' \leq s''$, add the core equality*:*

$$\beta \;\cup\; \left\{\, \mathrm{Cast}_{s' \to s''}(\mathrm{Cast}_{s \to s'}(x)) = \mathrm{Cast}_{s \to s''}(x) \,\right\} \;\mapsto\; \beta,$$

ensuring all casting paths between the same source and target sorts are identified.

(v) Membership translation (tr_M)***.*** *For each sort s appearing in a membership axiom, set*

$$\gamma \;\cup\; \{\, \mathrm{is\_s} : [\mathrm{tr}_S(\mathrm{kind}(s))] \to \texttt{Boolean} \,\} \;\mapsto\; \gamma.$$

Then, for each unconditional membership $t : s$ and each conditional membership $t : s$ if P, set $\beta \cup \{\mathrm{tr}_M(\cdot)\} \mapsto \beta$, where:

$$\mathrm{tr}_M(t : s) = \mathrm{is\_s}(\mathrm{tr}_T(t)) \qquad \mathrm{tr}_M(t : s \text{ if } P) = (\mathrm{tr}_T(P) \Rightarrow \mathrm{is\_s}(\mathrm{tr}_T(t)))$$

The five components of *tr* interact as a pipeline: tr_S determines how each sort is represented in Athena; tr_F uses these representations to declare function and cast symbols and to assert the structural axioms of A; tr_T relies on both to translate terms, inserting casts where implicit subsorting must become explicit; tr_E and tr_M then lift the term translation to equations and membership axioms, respectively. To make each step concrete, the rest of this section walks through *tr* component by component on two variants of the Peano numbers specification: the simple module in Listing 1.4 (no subsort relation) and the richer module in Listing 1.1 (with subsorts `NzNat` and `Even`), showing the corresponding Athena output in Listing 1.5 and Listing 1.6.

```
1 fmod PEANO is
2     sort Nat .
3     op zero : -> Nat [ctor] .
4     op s_ : Nat -> Nat [ctor] .
5     op _+_ : Nat Nat -> Nat .
6     vars N M : Nat .
7     eq N + zero = N .
8     eq N + s M = s (N + M) .
9 endfm
```

Listing 1.4. Simple Peano module in Maude.

```
1 module PEANO {
2     datatype Nat := zero  | (s Nat)
3     declare + : [Nat Nat] -> Nat
4     define [N M] := [?N:Nat ?M:Nat]
5     assert* (N + zero = N)
6     assert* (N + s M = s (N + M))
7 }
```

Listing 1.5. Simple Peano module in Athena.

Sorts. Through tr_S, Maude sorts are translated into Athena domains or datatypes. Whenever a sort $s \in S$ satisfies that it does not appear in the subsort relation $\leq$ and that there are constructors with target sort s, *tr* translates it to a datatype. In any other case, sorts are translated to domains. For instance, `zero` and `s_` in Listing 1.4 are function symbols (with target sort `Nat`) defined with the `ctor` attribute. Therefore, tr_S uses their sort signatures to define the corresponding Athena datatype, as shown in Line 2 of Listing 1.5. The remaining defined function symbols with target sort s are translated with the `declare` keyword. This is the case for the `_+_` function symbol, which is translated into Line 3 of Listing 1.5.

The treatment is different when the sort s is part of the subsort relation, as is the case in Listing 1.1. This module introduces `NzNat` and `Even` as subsorts of `Nat`. In this case, the previous translation case no longer applies. For sorts related through the subsort relation (`NzNat`, `Even`, and `Nat`) the translation function tr_S instantiates each sort as a plain Athena domain (Line 2 in Listing 1.6). The process then proceeds through the four remaining translation components (tr_F, tr_T, tr_E, tr_M): (1) function symbols and introducing explicit cast function symbols, (2) terms using casting function symbols, (3) equations into Athena assertions using the transformed terms, and (4) conditional membership axioms using term translation and predicate introduction.

Function Symbols. In the simple module (Listing 1.4), no subsort relation exists, so tr_F declares each function symbol directly: `_+_` becomes the Athena declaration in Line 3 of Listing 1.5. In the richer module (Listing 1.1), the subsorts introduce overloading. The three operators `zero`, `s_`, and `_+_` are grouped by tr_F into argument compatibility classes; for each class the unique maximal representative is selected and declared in γ (Lines 3–5 of Listing 1.6). Additionally, the two subsort pairs `Even` $\leq$ `Nat` and `NzNat` $\leq$ `Nat` produce two cast symbols (Lines 6–7 of Listing 1.6).

Terms and Equations. In the simple module, no subsort relation exists, so tr_T leaves terms unchanged and tr_E translates each equation directly into an Athena assertion. In the richer module, implicit coercions must be made explicit. Consider the first equation in Listing 1.1 (Line 9): because the argument has sort `Even` while the operator expects `Nat`, tr_T inserts a cast (Line 9 of Listing 1.6). The second equation (Line 10 of Listing 1.1) requires casts on both the argument and the result (Line 11 of Listing 1.6). Because the subsort poset of this module has no composable triple $s \leq s' \leq s''$, no core equalities are generated; in specifications with longer subsort chains, tr_E would add the corresponding transitivity assertions to β.

Memberships. Membership judgements have no native counterpart in Athena's many-sorted logic. The component tr_M bridges this gap by introducing a unary predicate for each sort that appears in a membership axiom; the predicate's domain is the kind of that sort, so it can be applied to any term of the right kind. In the richer Peano module, the conditional membership in Line 8 of Listing 1.1 is translated into an implication guarded by a membership predicate (Lines 12–13 of Listing 1.6). Here, tr_M chooses the kind of the target sort as the predicate's domain (*i.e.,* the topmost supersort in the connected component). The simple module contains no membership axioms, so tr_M produces no output for it. The complete translation of Listing 1.1 is shown in Listing 1.6.

```
1  module PEANO{
2    domains NzNat Even Nat
3    declare s : [Nat] -> NzNat
4    declare zero : [] -> Even
5    declare + : [Nat Nat] -> Nat
6    declare Cast_Even_to_Nat : [Even] -> Nat
7    declare Cast_NzNat_to_Nat : [NzNat] -> Nat
8    define [N M] := [?N:Nat ?M:Nat]
9    assert* ((+ N (Cast_Even_to_Nat zero)) = N)
10   assert* ((+ N (Cast_NzNat_to_Nat (s M)))
11         = (Cast_NzNat_to_Nat (s (+ N M))))
12   declare is_even : [Nat] -> Boolean
13   assert* ((is_even N) ==> (is_even s s N))  }
```

Listing 1.6. Peano Athena Module

Finally, the correctness of the translation is stated with respect to equational provability.

Theorem 1 (Equational Provability). *Let $\mathcal{E} = (\Sigma, E \cup A)$ be a membership equational theory. If Σ is strictly sensible and $\mathcal{E}$ is sufficiently complete w.r.t. $\Omega \subseteq \Sigma$, then for any $t, u \in T_{\Sigma,k}(X)$ of the same kind k, the following two sentences are equivalent:*

1. $t =_{\mathcal{E}} u$
2. $\mathrm{tr}_T(t) =_{\beta} \mathrm{tr}_T(u)$, i.e., $\mathrm{tr}_T(t)$ *and* $\mathrm{tr}_T(u)$ *are provably equal in Athena under the theory* $\mathrm{tr}(\mathcal{E}) = (\beta, \gamma)$ *with the inference rules of many-sorted equational logic.*

Proof. The equational fragment of the translation (components tr_S, tr_F, tr_T, and tr_E) instantiates the strictly sensible order-sorted–to–many-sorted mapping [5]. The membership component tr_M introduces predicate symbols and assertions of the form $\mathrm{is\_s}(\mathrm{tr}_T(t))$ and $\mathrm{tr}_T(P) \Rightarrow \mathrm{is\_s}(\mathrm{tr}_T(t))$; because is_s is a predicate symbol, no assertion produced by tr_M can have the form $t' = u'$ for terms t', u' of a non-Boolean sort, so tr_M does not introduce new equalities between translated terms. Hence, the equational content of β is precisely the image of E under tr_E, the structural axioms of A under tr_F, together with the core equalities.

- $\diamond$ $t =_{\mathcal{E}} u \Rightarrow \mathrm{tr}_T(t) =_\beta \mathrm{tr}_T(u)$
  The proof proceeds by structural induction on the derivation of $t =_{\mathcal{E}} u$, case-splitting on the last inference rule applied.
  - **Axiom instantiation.** Suppose $t =_{\mathcal{E}} u$ is obtained by applying a substitution θ to an equation $l = r$ in $E \cup A$, so that $t = \theta(l)$ and $u = \theta(r)$. By tr_E, the translated equation $\mathrm{tr}_T(l) = \mathrm{tr}_T(r)$ is in β, universally quantified over all free variables. Because tr_T is defined compositionally and the cast insertion at each argument position depends only on the sort of the subterm and the expected sort of the position, tr_T commutes with substitution: for each variable x, define $\hat{\theta}(x) = \mathrm{tr}_T(\theta(x))$; then $\mathrm{tr}_T(\theta(t')) = \hat{\theta}(\mathrm{tr}_T(t'))$ for any term t', possibly up to core equalities in β that identify different casting paths to the same target sort. Hence, $\hat{\theta}(\mathrm{tr}_T(l)) =_\beta \hat{\theta}(\mathrm{tr}_T(r))$ follows by instantiation of the universally quantified axiom in β.
  - **Reflexivity, symmetry, transitivity.** These are inference rules of many-sorted equational logic and thus hold natively in Athena's deductive system.
  - **Replacement.** Suppose $t = C[t']$ and $u = C[u']$ for a context C and $t' =_{\mathcal{E}} u'$, and assume by the induction hypothesis that $\mathrm{tr}_T(t') =_\beta \mathrm{tr}_T(u')$. The translation tr_T maps C to a context C' in the many-sorted signature γ: each function symbol in C is replaced by its maximal representative f^*, and explicit casts are inserted at every argument position where a subsort coercion is needed. Because each cast symbol $\mathrm{Cast}_{s \to s'}$ is a function symbol in γ, the standard congruence rule of many-sorted equational logic guarantees that if $a =_\beta b$ then $\mathrm{Cast}_{s \to s'}(a) =_\beta \mathrm{Cast}_{s \to s'}(b)$. Moreover, if t' and u' require different casting chains to reach the expected sort of the hole in C', the core equalities (added by tr_E for every composable pair $s \leq s' \leq s''$) ensure that both chains yield equal results in β. Hence, $\mathrm{tr}_T(C[t']) = C'[\mathrm{tr}_T(t')] =_\beta C'[\mathrm{tr}_T(u')] = \mathrm{tr}_T(C[u'])$ by congruence in many-sorted equational logic.

  Since every inference rule used in the derivation of $t =_{\mathcal{E}} u$ is preserved, $\mathrm{tr}_T(t) =_\beta \mathrm{tr}_T(u)$.
- $\diamond$ $\mathrm{tr}_T(t) =_\beta \mathrm{tr}_T(u) \Rightarrow t =_{\mathcal{E}} u$
  It suffices to show that β is a conservative extension of the image of $E \cup A$ with respect to equalities between translated terms. The axioms of β fall into three classes:
  (a) translated equations $\mathrm{tr}_T(l) = \mathrm{tr}_T(r)$ for each $l = r$ in $E \cup A$;
  (b) core equalities $\mathrm{Cast}_{s' \to s''}(\mathrm{Cast}_{s \to s'}(x)) = \mathrm{Cast}_{s \to s''}(x)$ for every composable pair $s \leq s' \leq s''$;
  (c) membership assertions produced by tr_M.

  As argued above, class (c) consists entirely of Boolean-sorted sentences and cannot produce new equalities between non-Boolean terms.

  For classes (a) and (b), the argument proceeds as follows:
  - **Unambiguous operator mapping.** Strict sensibility guarantees that each argument compatibility class $[f]_{\mathrm{ac}}$ has a unique maximal representative f^*. The translation maps every f in the same class to the *same* f^* and *omits* all others. Because the selection is unique, no two originally distinct function symbols are merged by the translation, therefore no invalid equalities between terms arise from the operator mapping.

- **Conservative cast equalities.** Each cast symbol $\mathrm{Cast}_{s \to s'}$ is a fresh uninterpreted function symbol in γ, distinct for each pair $s < s'$. The only axioms governing cast symbols are the core equalities in class (b), which encode the transitivity and path independence of the subsort relation. Because the subsort relation $\leq$ is already part of $\mathcal{E}$, the core equalities do not identify terms that are not already related by subsorting.
- **Isomorphism of initial algebras.** According to Li *et al.* [5], the strictly sensible mapping combined with the core casting equalities induces a bijection between the equivalence classes of the initial order-sorted algebra $T_{\Sigma/(E \cup A)}$ and those of the initial many-sorted algebra $T_{\gamma/\beta_{\mathrm{eq}}}$, where β_{eq} denotes the equational part of β (classes (a) and (b)). The sufficient completeness assumption ensures that every ground term reduces to a constructor term, which is needed for the bijection to be an isomorphism of algebras.

Therefore, if $\mathrm{tr}_T(t) =_\beta \mathrm{tr}_T(u)$, then $\mathrm{tr}_T(t)$ and $\mathrm{tr}_T(u)$ lie in the same equivalence class of $T_{\gamma/\beta_{\mathrm{eq}}}$. By the isomorphism, t and u belong to the same class in $T_{\Sigma/(E \cup A)}$, *i.e.,* $t =_{\mathcal{E}} u$.

4 Inductive Reasoning

The translation function tr from $(\Sigma, E \cup A)$ to (β, γ) presented in Section 3 enables equational reasoning over Maude specifications within Athena by preserving the underlying equational theory in a many-sorted first-order logic setting. However, equational reasoning alone is often insufficient to establish properties of interest, which typically require some form of inductive reasoning. This section explains how equational reasoning with $tr(\mathcal{E})$ in Athena can be extended with inductive reasoning based on parametric induction principles. The key idea is that for the induction principle to be applied, it must be instantiated using the structure of sorts in Σ associated with each connected component in the poset $(S, \leq)$. To illustrate this approach, this section presents a concrete instantiation of such a parametric induction principle, showing how structural induction can be recovered and applied in practice.

In Athena, there is native support for structural, constructor-based induction over datatypes, which allows properties to be proved by reasoning directly on the inductive structure generated by datatype constructors. However, the translation function tr systematically represents many Maude sorts as domains, rather than datatypes, in order to preserve the structure induced by subsorting in the many-sorted setting of Athena. As a consequence, Athena's built-in induction mechanisms are no longer directly applicable, and native datatype induction is insufficient for reasoning about the translated specifications. To address this limitation, the approach adopted here is to equip Athena with induction principles defined over domains, rather than relying solely on datatype induction. These principles are formulated in a parametric way, allowing different induction schemes to be instantiated depending on the structure of the sorts under consideration. In particular, this includes natural extensions of Athena's native structural induction to domains that arise from translated equational specifications.

More generally, an induction principle in this setting is realized by declaring a primitive method in Athena that explicitly encodes both the basis cases and the inductive cases associated with a given sort structure $(S, \leq)$. For a selected induction principle η, the primitive method is defined separately for each connected component of the sort hierarchy, reflecting the constructors and subsort relations that characterize that component. The basis cases correspond to the minimal elements or constructors of the component, while the inductive cases capture how the property is preserved by the relevant constructor applications. Once declared, such a primitive method can

be applied uniformly to predicates over the corresponding domain, thereby extending equational reasoning with inductive reasoning in a principled and modular way.

The following definition extends the translation function (Definition 3) with a sixth component, tr_η, that formalizes the generation of structural induction principles. It operates on the execution environment ρ—previously noted as orthogonal to the generated representation (β, γ), but now required to host the primitive methods that realize induction.

Definition 4 (Induction Method Translation tr_η). *Let $\mathcal{E}$, Σ, Ω, and $(S, \leq)$ be as in Definition 3. The translation function* tr *is extended with:*

(vi) Induction method translation (tr_η). *For each kind k in $(S, \leq)$, let $C = \{\, s \in S \mid \exists\, c \in \Omega.\ \mathrm{target}(c) = s \,\} \cap [k]_{\leq}$ be the set of sorts in the connected component of k that are directly generated by Ω-constructors. Define the* effective constructor set

$$\Omega_C^+ \;=\; \Omega_C \;\cup\; \{\, \mathrm{Cast}_{s \to s'} \mid s \notin C,\ s' \in C,\ s \leq s' \,\},$$

where $\Omega_C = \{c \in \Omega \mid \mathrm{target}(c) \in C\}$ and the second component adds subsort injection casts from sorts outside C into C. By construction, Ω_C^+ is jointly exhaustive: every constructor normal form of kind k is headed by some element of Ω_C^+. For each $c : s_1 \times \cdots \times s_n \to s$ in Ω_C^+, let $I_c = \{\, i \mid 1 \leq i \leq n,\ s_i \in C \,\}$ be the set of recursive argument positions*. The component tr_η generates a primitive method that takes a predicate $P : \mathrm{tr}_S(k) \to \texttt{Boolean}$ and encodes the obligations of $\eta_C(P)$ as follows.*

(a) Base-case sentences.

(1) For each constant $c :\to s$ in Ω_C^+ (0-arity, so $I_c = \varnothing$), define

$$b_c \;:=\; P\big(\mathrm{Cast}_{s \to k}(c)\big).$$

(2) For each constructor $c : s_1 \times \cdots \times s_n \to s$ in Ω_C^+ with $n \geq 1$ and $I_c = \varnothing$, define

$$b_c \;:=\; \forall\, x_1 \cdots x_n\ .\ P\big(\mathrm{Cast}_{s \to k}(c(x_1, \ldots, x_n))\big),$$

where each $x_i : \mathrm{tr}_S(s_i)$.

(b) Inductive-case sentences. *For each constructor $c : s_1 \times \cdots \times s_n \to s$ in Ω_C^+ with $I_c \neq \varnothing$, define the sentence*

$$h_c \;:=\; \forall\, x_1 \cdots x_n\ .\ \Big(\bigwedge_{i \in I_c} P\big(\mathrm{Cast}_{s_i \to k}(x_i)\big)\Big) \;\Rightarrow\; P\big(\mathrm{Cast}_{s \to k}(c(x_1, \ldots, x_n))\big),$$

where each $x_i : \mathrm{tr}_S(s_i)$. When $|I_c| = 1$, the conjunction reduces to a single hypothesis.

(c) Primitive method assembly. *Let $B = \{b_c\}$ and $H = \{h_c\}$ be the collected base-case and inductive-case sentences, and let $(o_1, \ldots, o_m)$ be an enumeration of all sentences in $B \cup H$. The primitive method for component C is added to the execution environment ρ as*

$$\rho \;\cup\; \big\{\, \texttt{prim-method}_C(P) \;:\; \texttt{check}_m \;\Rightarrow\; \forall\, x : \mathrm{tr}_S(k).\ P(x) \,\big\} \;\mapsto\; \rho,$$

where each obligation is verified sequentially through nested `check` *expressions:* $\texttt{check}_1$ *tests* $\texttt{holds}?(o_1)$ *and, upon success, enters* $\texttt{check}_2$*, which tests* $\texttt{holds}?(o_2)$*, and so on. In general, for $i = 1, \ldots, m$,* $\texttt{check}_i$ *tests* $\texttt{holds}?(o_i)$ *and proceeds to* $\texttt{check}_{i+1}$*; when $i = m$, the innermost check concludes $\forall\, x : \mathrm{tr}_S(k).\ P(x)$. If any* $\texttt{holds}?(o_i)$ *fails, the method signals an error.*

The full translation thus maps $\mathcal{E} \mapsto (\beta, \gamma, \rho)$, extending the original pair with the execution environment populated by the generated primitive methods.

As an example, consider the primitive method in Listing 1.7. It defines `nat-induction`, a structural induction principle for the `Nat` domain in Listing 1.6. It explicitly captures the two components of a standard induction proof. The first definition, labeled `basis`, corresponds to the sentence of the basis case over the constant constructor `Cast_Even_to_Nat zero`. The second definition, labeled `ic`, encodes the inductive case: it states that if the property holds for an arbitrary `Nat` element `x` (inductive hypothesis), then it needs to hold for its successor, `Cast_NzNat_to_Nat (s x)`. The `check` expression enforces both conditions. It first verifies that the basis case holds, and if so, proceeds to check the inductive case. This mechanism effectively restores structural induction for domains translated from Maude datatypes, allowing the proof of properties such as associativity or commutativity complementing the built-in `by-induction` method.

```
primitive-method (nat-induction property) :=
  let {
    basis := (property (Cast_Even_to_Nat zero));
    ic := (forall x (if (property x) (property (Cast_NzNat_to_Nat (s x)))))
    }
    check { (holds? basis) =>
              check {(holds? ic) => (forall x (property x))
                    | else => (error "Inductive step does not hold.")}
              | else => (error "Basis step does not hold.")}
```

Listing 1.7. Primitive induction method for Nat.

The following theorem establishes the soundness of inductive proofs carried out using the primitive methods generated by tr_η.

Theorem 2 (Soundness of Inductive Proofs). *Let $\mathcal{E} = (\Sigma, E \cup A)$ be a membership equational theory that is sufficiently complete a connected component of $(S, \leq)$ with kind k, and $P : \mathrm{tr}_S(k) \to$ `Boolean` a predicate. If the primitive method $\texttt{prim-method}_C(P)$ generated by tr_η (Definition 4) checks all base cases and inductive steps and concludes $\forall\, x : \mathrm{tr}_S(k).\ P(x)$, then for every ground term t of sort $s \in C$ in $T_{\Sigma,k}$, the property $P(\mathrm{tr}_T(t))$ holds in Athena under the theory $\mathrm{tr}(\mathcal{E}) = (\beta, \gamma, \rho)$.*

Proof. Assume, for contradiction, that there exists a ground term t of sort $s \in C$ in $T_{\Sigma,k}$ such that $P(\mathrm{tr}_T(t))$ does not hold, even though all obligations of $\texttt{prim-method}_C(P)$ have been checked. Because $\mathcal{E}$ is sufficiently complete, t reduces to a constructor normal form headed by some $c \in \Omega$ with $\mathrm{target}(c)$ in the connected component of k. If $\mathrm{target}(c) \in C$, then $c \in \Omega_C \subseteq \Omega_C^+$; otherwise c is an injection from an external sort $s \notin C$ with $s \leq s'$ for some $s' \in C$, so $c = \mathrm{Cast}_{s \to s'}$ belongs to the second component of Ω_C^+. In either case $c \in \Omega_C^+$. By Theorem 1, $P(\mathrm{tr}_T(t))$ holds if and only if P holds on the translated normal form, so it may be assumed without loss of generality that $t = c(t_1, \ldots, t_n)$ is already in constructor normal form. It proceeds by induction on the number of constructor applications in t.

Base case: $(I_c = \varnothing)$. The constructor c has no argument positions of sorts in C, so t contains exactly one constructor application. The primitive method verified the base-case sentence b_c (Definition 4). Whether $c :\to s$ is a constant (in which case b_c directly asserts $P(\mathrm{Cast}_{s \to k}(c))$) or $c : s_1 \times \cdots \times s_n \to s$ with $n \geq 1$ and $I_c = \varnothing$ (in which case b_c is universally quantified and instantiates at $\mathrm{tr}_T(t_1), \ldots, \mathrm{tr}_T(t_n)$) the verified sentence b_c yields $P(\mathrm{tr}_T(t))$.

Inductive step: ($I_c \neq \varnothing$). For each $i \in I_c$, the subterm t_i is a ground term of sort $s_i \in C$ with strictly fewer constructor applications than t. By the induction hypothesis, $P(\mathrm{tr}_T(t_i))$ holds for every $i \in I_c$. The primitive method verified the inductive-case sentence h_c (Definition 4), which asserts that the conjunction of $P(\mathrm{Cast}_{s_i \to k}(x_i))$ for $i \in I_c$ implies $P(\mathrm{Cast}_{s \to k}(c(x_1, \ldots, x_n)))$. Instantiating at $\mathrm{tr}_T(t_1), \ldots, \mathrm{tr}_T(t_n)$ and applying modus ponens with the induction hypotheses yields $P(\mathrm{tr}_T(t))$.

In both cases $P(\mathrm{tr}_T(t))$ holds, contradicting the assumption. Since t was arbitrary, $P(\mathrm{tr}_T(t))$ holds for every ground term of kind k.

5 Case Study

The case study verifies inductive properties of a compiler for numeric expressions on a Stack Machine [4]. The case study relies heavily on order-sorted features of data types and includes structural axioms.[5] Listing 1.8 shows the Maude specification, defining three sorts: a source language of arithmetic expressions (`Exp`), a target instruction set (`Instr`/`Program`), and a stack machine (`Stack`).

```
fmod TOY-COMPILER is
  protecting INT .
  sort Exp .
  subsort Int < Exp .
  op _plus_ : Exp Exp -> Exp [ctor] .
  op _minus_ : Exp Exp -> Exp [ctor] .
  op _mult_ : Exp Exp -> Exp [ctor] .
  sorts Instr Program Stack .
  subsort Instr < Program .
  op push : Int -> Instr [ctor] .
  ops add sub mult : -> Instr [ctor] .
  op nil : -> Program [ctor] .
  op _++_ : Program Program -> Program [ctor assoc id: nil] .
  op empty : -> Stack [ctor] .
  op _::_ : Int Stack -> Stack [ctor] .
  op I : Exp -> Int .          --- interpreter
  op compile : Exp -> Program .
  op exec : Program Stack -> Stack .
  vars N N1 N2 : Int .  vars E1 E2 : Exp .
  var P : Program .     var S : Stack .
  eq I(N) = N .
  eq I(E1 plus E2) = I(E1) + I(E2) .
  --- I equations for minus, mult analogous
  eq compile(N) = push(N) .
  eq compile(E1 plus E2) =
     compile(E2) ++ compile(E1) ++ add .
  --- compile equations for minus, mult analogous
  eq exec(nil, S) = S .
  eq exec(push(N) ++ P, S) = exec(P, N :: S) .
  eq exec(add ++ P, N1 :: N2 :: S) =
```

[5] The complete Maude specification of the compiler with the translated equationally equivalent Athena program (with proofs) is available at: `https://github.com/FLAGlab/Maude2Athena`

```
31      exec(P, (N1 + N2) :: S) .
32    --- exec equations for sub, mult analogous
33    ...
34  endfm
```

Listing 1.8. Toy compiler in Maude (shorten).

The compiler specification relies on two order-sorted features. First, `Int < Exp` lets integers appear directly as expressions without an explicit injection constructor. Second, `Instr < Program` lets single instructions be used where programs are expected. The operator `_++_` carries the structural axioms `assoc` and `id: nil`.

To preserve the semantics of `Int < Exp`, the translation introduces explicit cast functions for every subsort relation in the sort poset, as explained in Section 3. maude2athena automatically inserts these casts. For example, the equations in lines Line 24 and Line 29 are translated to:

```
declare push : [Int] -> Instr
declare compile : [Exp] -> Program
declare Cast_Int_to_Exp : [Int] -> Exp
declare Cast_Instr_to_Program : [Instr] -> Program
assert* eq_4 := ((compile (Cast_Int_to_Exp N))
                    = (Cast_Instr_to_Program (push N)))
assert* eq_9 := ((exec (++ (Cast_Instr_to_Program (push N)) P) S)
                    = (exec P (:: N S)))
```

Since Athena requires unique function symbols and does not support Maude's mixfix notations natively, the tool flattens operators and resolves overloading. Specifically, the mixfix Maude operator `_plus_` is translated to a standard prefix function in Athena. If overloading collisions are detected, unique identifiers are generated based on sort signatures:

```
declare plus : [Exp Exp] -> Exp
declare ++ : [Program Program] -> Program
assert* ((compile (plus E1 E2)) =
          (++ (compile E2)
              (++ (compile E1) (Cast_Instr_to_Program add))))
define [_v1 _v2 _v3 _v4] :=
       [?_v1:Program ?_v2:Program ?_v3:Program ?_v4:Program]
assert* assoc_++ := ((++ (++ _v1 _v2) _v3) = (++ _v1 (++ _v2 _v3)))
assert* left_id_++ := ((++ nil _v4) = _v4)
assert* right_id_++ := ((++ _v4 nil) = _v4)
```

This listing also illustrates how the structural axioms A are handled. The operator `_++_` is declared with the attributes `assoc, id: nil`, meaning that rewriting is performed modulo associativity and identity. Because Athena has no built-in support for such axioms, tr_F translates each structural axiom into an explicit equational assertion: `assoc_++` encodes associativity, while `left_id_++` and `right_id_++` encode the left and right identity axioms for `nil`. In general, any combination of associativity, commutativity, and identity (ACU) attributes attached to operators in A is translated by tr_F into the corresponding set of universally quantified assertions in β.

The tool analyzes the signature to determine which sorts can be modeled as `datatypes` (allowing native induction) and which must be `domains`. In this example, the Maude `Stack` is translated as an Athena datatype:

```
datatype Stack := empty | (:: Int Stack)
```

However, `Exp` and `Program` are translated as domains to accommodate the subsorting relations that are otherwise incompatible with standard datatype constructors.

The result of this translation is a fully typed Athena module where the correctness of the compiler (`forall e . (exec (compile e)) = (I e)`) can be rigorously proven. The core verification goal is the following theorem, which asserts that compiling an expression `e` followed by a program `p` is equivalent to executing `p` on a stack with the evaluated result of `e`:[6]

```
(forall ?p ?s .   (exec (++ (compile ?e) ?p) ?s) =
                  (exec ?p (:: (I ?e) ?s)))
```

Note that this sentence cannot be asserted as a logical sentence because `Exp` is a domain, not a datatype. Instead, it is defined as a predicate over expressions. This enables the theorem to be passed as an argument to the induction method of choice. In this case, the predicate `correctness` is defined as:

```
declare correctness: [Exp] -> Boolean
assert* correctness_def := (iff (correctness ?e)
    (forall ?p ?s .
        (exec (++ (compile ?e) ?p) ?s) = (exec ?p (:: (I ?e) ?s))))
```

Proving that `(forall ?e . correctness ?e)` holds, brings to attention the challenges of reasoning over non-inductive domains and handling explicit subsorting.

Note that in Maude, `Exp` is defined via subsorting ($Int < Exp$), which prevents it from being translated into a standard Athena `datatype`. Consequently, Athena's native `by-induction` method cannot be applied. To overcome this, maude2athena automatically generates a custom **primitive method** that reconstructs the inductive principle of the original Maude sort. The generated method, `exp-induction`, takes the `correctness` predicate as an argument and strictly enforces that the user proves the inductive step for every constructor (e.g., `plus`, `mult`) before discharging the goal:

```
primitive-method (exp-induction property) :=
    let {
        basis_n  := (forall ?n (property (Cast_Int_to_Exp ?n)))
        ic_plus  := (forall ?e1 (forall ?e2
                        (if (and (property ?e1) (property ?e2))
                            (property (plus ?e1 ?e2)))));
        ic_minus := ...
        ic_mult  := ...
    }
    check { (holds? basis_n) =>   ...  }
```

The translation replaces implicit subsorting with explicit cast operators. This requires the proof script to handle these casts in the basis cases. For instance, the basis case for integers is not merely `forall n`, but rather `forall n` applied to the cast `Cast_Int_to_Exp`. The proof proceeds by unwrapping these casts using the axioms generated by the translation (*e.g.,* `eq_4`):

[6] The correctness property follows trivially as a corollary of this theorem by instantiating it to the empty program on an empty stack.

```
define basis_step := (forall n . correctness (Cast_Int_to_Exp n))
conclude basis_step
  pick-any n:Int p:Program s:Stack
    (!chain [ (exec (++ (compile (Cast_Int_to_Exp n)) p) s)
            = (exec (++ (Cast_Instr_to_Program (push n)) p) s)   [eq_4]
            = (exec p (:: n s))                                  [eq_9]
            ... ])
```

This explicit handling ensures that the proof is rigorous and mathematically consistent with the original order-sorted semantics. Finally, the proof is concluded by invoking the custom induction method, which aggregates the base case and inductive steps to discharge the main theorem: `(!exp-induction correctness)`.

6 Related Work

The formal verification of rewriting logic specifications typically involves a trade off between the high performance execution provided by environments like Maude and the interactive deductive capabilities of theorem provers like Athena [4], Isabelle [20], or Lean [21]. There are three main methodologies designed to address this challenge: (1) the development of native domain-specific theorem provers, (2) the application of automated symbolic reasoning techniques, and (3) the integration with general purpose proof assistants via translation.

The native approach, pioneered by the original Maude Inductive Theorem Prover (ITP) [2] and modernized by NuITP [22], offers powerful automation using variant-based reasoning directly at Maude's metalevel. While highly effective for discharging goals automatically without translation, tools like NuITP inherently generate proofs as metalevel search traces. In contrast, maude2athena translates specifications into Athena to leverage its natural-deduction framework, which closely mirrors standard mathematical reasoning [13]. Rather than competing on pure automation, the proposed approach emphasizes proof transparency and structure. Athena's built-in methods (equality chaining, structural induction, case analysis, and many others) allow proofs to be developed and read as structured mathematical arguments rather than execution traces or low-level tactic scripts, following human-like proof strategies [23]. Consequently, maude2athena occupies a complementary niche to native provers like NuITP: it targets verification scenarios where transparent, independently checkable, and human-readable proofs are the primary requirement.

A highly effective alternative to interactive deduction relies on automated symbolic methods. Maude integrates with SMT solvers to discharge proof obligations generated from specifications. Furthermore, techniques such as narrowing and its combination with SMT solving allow for the analysis of infinite state systems [24]. Reachability Logic and its associated tools also provide mechanisms to deductively reason about Maude programs [25, 26]. These automated methodologies are essential for verifying safety properties and analyzing state spaces, though they may not cover properties requiring complex inductive arguments or higher order reasoning.

The third strategy is to translate specifications to an external formalism to leverage its proof infrastructure. The Heterogeneous Tool Set (Hets) [27] treats Maude as an institution, enabling translations to Isabelle/HOL via CASL [28]. While theoretically robust, this path relies on complex encodings to simulate subsorting. Similarly, approaches to translate order-sorted algebras into many-sorted algebras have been explored to leverage standard theorem provers [5, 6]. The Maude2Lean framework [29] presents a direct translation from rewriting logic to the calculus of inductive constructions. Unlike the institutional approach of Hets, Maude2Lean translates the syntax of Maude

terms and the semantics of rewriting logic into inductive datatypes in Lean. This preserves the inductive structure of the logic, allowing native structural induction on terms and relations. However, such translations shift the first-order theorem proving problem to a different domain (e.g., type theory or higher order logic), sometimes incurring in a verbosity trade-off, as proofs become arguments about the translated semantics rather than direct reasoning about the domain model.

7 Concluding Remarks

This work presents maude2athena, a tool for translating order-sorted Maude specifications into the many-sorted first-order logic of Athena. The main contribution of this work is the reconciliation of symbolic reasoning and equational rewriting with the power of inductive reasoning, enabling specifications that leverage the best of both worlds. To the best of the authors' knowledge, this is the first implementation grounded in the theoretical translation proposed by Li *et al.* [5]. The proposed approach bridges the semantic gap between the two formalisms by making subsorting explicit via cast operators, modeling membership as predicates, and recovering structural induction through custom primitive methods. The maude2athena framework was validated using several examples. It includes the case study of a compiler specification, demonstrating the approach's ability to handle deep inductive structures and operator overloading, while preserving the equational consistency of the original system.

The future work focuses on two key areas. First, to enhance modularity by mapping Maude's built-in modules directly to Athena's native domains. Such integration would allow developers to leverage Athena's existing decision procedures and automated reasoning capabilities for standard data types. Second, extending maude2athena to handle rewrite rules to enable the verification of concurrent and distributed systems. The current version is restricted to functional modules, while rewrite rules are the primary mechanism for defining concurrent state transitions. Athena, being grounded in standard first-order logic, lacks native primitives for modeling the non-determinism and asynchrony inherent in such rules. Rather than constructing a new concurrency framework within Athena from scratch, the idea is to directly work on top of Talcott's formalization of Actor theories in rewriting logic [30]. By translating these pre-validated semantic structures directly into Athena, Maude's system modules can be mapped to logical transition relations. On the one hand, this would significantly reduce the formalization burden. On the other hand, it will effectively equip Athena with the capability to reason about temporal properties and distributed protocols without needing to manually reconstruct the underlying theory of concurrency.

Acknowledgements. The authors would like to thank the anonymous reviewers for their very insightful comments on an earlier version of this paper. They would like to also thank Konstantine Arkoudas for his insight on encoding structural induction over plain domains as a primitive method in Athena to preserve the inductive reasoning capabilities. Rocha acknowledges support from the SGR project PROMUEVA (BPIN 2021000100160) under the supervision of Minciencias (Ministerio de Ciencia Tecnología e Innovación, Colombia).

References

1. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical computer science (1992)

2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C.: All about Maude-a high-performance logical framework: how to specify, program, and verify systems in rewriting logic. Springer (2007)
3. Clavel, M., Meseguer, J., and Palomino, M.: Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. Theoretical Computer Science (2007)
4. Arkoudas, K. and Musser, D.: Fundamental proof methods in computer science: a computer-based approach. MIT Press (2017)
5. Li, L. and Gunter, E.L.: A Method to Translate Order-Sorted Algebras to Many-Sorted Algebras. In: Fourth International Workshop on Rewriting Techniques for Program Transformations and Evaluation. EPTCS, pp. 20–34 (2017)
6. Goguen, J.A. and Meseguer, J.: Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theoretical Computer Science (1992)
7. Rocha, C. and Meseguer, J.: Constructors, Sufficient Completeness, and Deadlock Freedom of Rewrite Theories. In: Logic for Programming, Artificial Intelligence, and Reasoning, pp. 594–609. Springer (2010)
8. Bouhoula, A., Jouannaud, J.-P., and Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science (2000)
9. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Recent Trends in Algebraic Development Techniques, pp. 18–61. Springer (1998)
10. Durán, F. and Meseguer, J.: On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. The Journal of Logic and Algebraic Programming (2012)
11. Lucas, S. and Meseguer, J.: Operational termination of membership equational programs: the order-sorted way. Electronic Notes in Theoretical Computer Science (2009)
12. Arkoudas, K.: Denotational proof languages. PhD thesis, Massachusetts Institute of Technology (2000).
13. Arkoudas, K.: Simplifying Proofs in Fitch-Style Natural Deduction Systems. Journal of Automated Reasoning (2005)
14. Manzano, M.: Extensions of first-order logic. Cambridge University Press (1996)
15. Musser, D.R. and Varela, C.A.: Structured Reasoning About Actor Systems. In: Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control. AGERE! 2013, pp. 37–48. ACM, Indianapolis, Indiana, USA (2013)
16. Varela, C.A.: "The AMST Language: Formal Verification and Execution of Actor Systems". In: Concurrent Programming, Open Systems and Formal Methods: Essays Dedicated to Gul Agha to Celebrate His Scientific Career. Cham: Springer Nature Switzerland, 2026, pp. 36–59.
17. Paul, S., McCarthy, C., Patterson, S., and Varela, C.: Formal verification of timely knowledge propagation in airborne networks. Science of Computer Programming (2025)
18. Paul, S., Cruz, E., Dutta, A., Bhaumik, A., Blasch, E., Agha, G., Patterson, S., Kopsaftopoulos, F., and Varela, C.: Formal Verification of Safety-Critical Aerospace Systems. IEEE Aerospace and Electronic Systems Magazine (2023)
19. Meseguer, J. and Skeirik, S.: Equational formulas and pattern operations in initial order-sorted algebras. Form. Asp. Comput. (2017)
20. Wenzel, M., Paulson, L.C., and Nipkow, T.: The Isabelle framework. In: International Conference on Theorem Proving in Higher Order Logics, pp. 33–38 (2008)
21. Moura, L.d. and Ullrich, S.: The Lean 4 theorem prover and programming language. In: International Conference on Automated Deduction, pp. 625–635 (2021)

22. Durán, F., Escobar, S., Meseguer, J., and Sapiña, J.: NuITP: An inductive theorem prover for equational program verification. In: International Symposium on Principles and Practice of Declarative Programming, pp. 1–11 (2024)
23. Bringsjord, S. and Govindarajulu, N.S.: Fundamental Proof Methods in Computer Science: A Computer-Based Approach, by Arkoudas and Musser, The MIT Press, Cambridge, USA, ISBN 978-0-262-03553-8. Theory and Practice of Logic Programming (2021)
24. Rocha, C., Meseguer, J., and Muñoz, C.: Rewriting modulo SMT and open system analysis. In: International Workshop on Rewriting Logic and its Applications, pp. 247–262 (2014)
25. Romero, M. and Rocha, C.: Symbolic Execution and Reachability Analysis Using Rewriting Modulo SMT for Spatial Concurrent Constraint Systems with Extrusion. In: NASA Formal Methods, pp. 435–451. Springer (2018)
26. Arias, J., Bae, K., Olarte, C., Ölveczky, P.C., Petrucci, L., and Rømming, F.: Symbolic analysis and parameter synthesis for time Petri nets using Maude and SMT solving. In: International Conference on Applications and Theory of Petri Nets and Concurrency, pp. 369–392 (2023)
27. Codescu, M., Mossakowski, T., Riesco, A., and Maeder, C.: Integrating Maude into HETS. In: International Conference on Algebraic Methodology and Software Technology, pp. 60–75 (2010)
28. Astesiano, E., Bidoit, M., Kirchner, H., Krieg-Brückner, B., Mosses, P.D., Sannella, D., and Tarlecki, A.: CASL: the common algebraic specification language. Theoretical Computer Science (2002)
29. Rubio, R. and Riesco, A.: Maude2Lean: Theorem proving for Maude specifications using Lean. Journal of Logical and Algebraic Methods in Programming (2022)
30. Talcott, C.: Actor theories in rewriting logic. Theoretical Computer Science (2002)

A Theory of Composable Lingos for Protocol Dialects

Víctor García[1], Santiago Escobar[1], Catherine Meadows[2], and Jose Meseguer[3]

[1] VRAIN, Universitat Politècnica de València, Valencia, Spain
{vicgarval,sescobar}@upv.es
[2] cameadows3306@gmail.com
[3] University of Illinois at Urbana-Champaign, Illinois, USA
meseguer@illinois.edu

Abstract. Formal patterns are formally specified solutions to frequently occurring distributed system problems that are generic, executable, and come with strong qualitative and/or quantitative formal guarantees. A formal pattern is a *generic system transformation* which transforms a usually infinite class of systems in need of the pattern's solution into enhanced versions of such systems that solve the problem in question. In this paper we demonstrate the application of formal patterns to *protocol dialects*. Dialects are methods for hardening protocols so as to endow them with light-weight security, especially against easy attacks that can lead to more serious ones. A *lingo* is a dialect's key security component, because attackers are unable to "speak" the lingo. A lingo's "talk" changes all the time, becoming a moving target for attackers. In this paper we present several formal patterns for both lingos and dialects. Lingo formal patterns can make lingos stronger by both *transforming* them and by *composing* several lingos into a stronger lingo. Dialects themselves can be obtained by the application of a single *dialect formal pattern*, generic on both the chosen lingo and the chosen protocol.

Keywords: Formal Patterns, Lingos, Compositions, Security, Maude

1 Introduction

Formal patterns [18] are methods for formally specifying generic solutions to distributed system problems that are executable and provide formal guarantees. Such patterns allow one to retain what has been learned about solving such problems without tying that knowledge to a particular concrete solution. They also allow for the composition of solutions and the composition of guarantees.

Formal patterns have been successfully applied to a number of different distributed systems problems [6, 19, 25]. They have also been applied to security protocols, and have been helpful in developing methods for hardening protocols against malicious attacks [4,7,21]. In this paper we apply this methodology to an emerging protocol hardening technique, the design of *protocol dialects*. Protocol dialects are methods for modifying protocols in order to provide a light-weight

layer of security, especially against relatively easy attacks that could potentially be leveraged into more serious ones. The scenario is usually that of a network of mutually trusting principals, e.g., an enterprise network, that needs to defend itself from outside attackers. The most effective approach is to prevent outside parties from even initiating a communication with group members, e.g., by requiring messages to be modified in some way unpredictable by the attacker.[4]

Protocol dialects are intended to be used as a first line of defense, not as a replacement for traditional authentication protocols. Indeed, they are primarily concerned with protecting communication between members of an enclave in which all parties trust each other against weak attackers that are trying to leverage off protocol vulnerabilities. We can make use of this restricted scope to use an attacker model based on the on-path attacker model [8,11]: First, the attacker can read messages, but cannot destroy or replace them. Second, principals trust each other and share a common secret that they can use, for example, as the seed for a pseudo-random number generator to generate and share common secret parameters. Because of the simplified threat model and trust assumption, dialects can be designed to be simpler than full-scale authentication protocols. Thus they can be expected to be prone to fewer implementation and configuration errors. With this in mind, dialects can be used as a separate but very thin layer in the protocol stack, useful as a defense against attacks in case the main authentication protocol is misconfigured or implemented incorrectly. Considering the dialect as a separate layer makes it easier to keep it simple, because it makes it possible to avoid dependencies on the layers it is communicating with. In our work, we consider dialects that sit right above the transport layer in the TCP/IP model, while the protocol it is modifying sits just above the dialect in the application layer. However, other locations are possible for the protocol being modified. Variations in the threat model are also possible, e.g., deciding whether or not to include replay or session hijacking attacks, which can both be implemented by an on-path attacker.

Protocol dialects as formal patterns were first introduced in [8]. A dialect is generic on two parameters: the given *lingo*, which describes the actual *message transformation*, and the underlying *protocol* whose security it improves. The *dialect* pattern is a *protocol transformation* yielding a new protocol responsible for applying the lingo and interfacing with the underlying protocol. Lingos can be transformed and composed into stronger ones in a compositional way. Using lingo compositions a given protocol can be made more secure in a wide variety of ways by means of the different dialects used in such lingo compositions.

We substantially extend the work in [8] by concentrating on the design of lingos, instead of the design of dialects as a whole, and developing *new formal patterns* to create and transform new lingos from simpler ones. This is a more flexible and computationally more efficient approach that increases the security of transformed lingos (and therefore of dialects) and provides a simpler methodology for designing new lingos and dialects out of previous ones and increasing

[4] By "message" we mean a unit of information transferred by the protocol, and can mean anything from a packet to a message in the traditional sense.

their security. In particular, we are able to *replace* two important formal patterns for dialects in [8] by much simpler similar formal patterns for lingos. Lingos rely on constantly changing secret *parameters*, which are generated by a pseudorandom number generator shared throughout, itself parametric on a single shared secret, an approach to lingo design introduced by Gogeneni et al. in [11].

From the software engineering point of view, the importance of formal patterns —in particular that of lingos and dialects— is that they make possible the design and verification of distributed systems of much higher quality than those developed in a traditional way, and they endow such systems with desired new features —for dialects with new lightweight security features— in a fully modular way. As explained in [20], formal patterns provide a modular, generic way of developing formal executable specifications of system *designs* that can be subjected to systematic formal verification *before* they are implemented and support the correct by construction development of high quality systems with a notable economy of effort and a high degree of reusability. Mathematically, the *dialect formal pattern* is a *theory transformation* definable as a partial function:

$$\mathcal{D} : \mathbf{PLingos} \times \mathbf{Protocols} \ni (\Lambda_p, \mathcal{P}) \mapsto \mathcal{D}_{\Lambda_p}(\mathcal{P}) \in \mathbf{Protocols}$$

where **PLingos** and **Protocols** denote classes of executable specifications, i.e., of executable *theories*, respectively specifying the classes of parametric lingos and communication protocols. Likewise, as we explain in the paper, *lingo formal patterns* are similar theory transformations mapping one or more lingos to a new lingo enjoying specific formal properties.

1.1 Related Work

Probably, the first work on dialects is that of Sjoholmsierchio et al. in [24]. In this work, the dialect is proposed as a variation of an open-source protocol to introduce new security measures or remove unused code, and is applied to the OpenFlow protocol, introducing, among other things a defense against ciphersuite downgrade attacks on TLS 1.2. This variation was independent of TLS, and was achieved by adding proxies, thus allowing for modification without touching the rest of the system and foreseeing the use of dialects as thin layers. In later work by Lukaszewski and Xie [14] a layer 4.5 in the TCP/IP model was proposed for dialects. In our own model we also make use of proxies for both sending and receiving parties, but stop short of proposing an additional official layer.

The threat model we use in this paper, and the one used in [8], is similar to that used by Goginieni et al. in [11] and Ren et. al [22]: an on-path attacker who is not able to corrupt any members of the enclave, as discussed earlier. We have also followed the suggestion in [22] to view dialects as composable protocol transformations. In this paper, we simplify this approach by reducing dialect transformations to more basic lingo transformations, which are generic, compositional methods for producing new lingos (and thus new dialects) from old ones.

In [10,11,15] Mei, Gogineni, et al. introduce an approach to dialects in which the message transformation is updated each time using a shared pseudo-random

number generator. This means that the security of a transformation depends as much, and possibly more, on its unpredictability as it does on the inability of the attacker to reproduce a particular instance of a transformation. We have adopted and extended this approach. In particular, we model a transformation as a parametrized function, or *lingo*, that produces a new message using the original message and the parameter as input. Both the lingo and the parameter can be chosen pseudo-randomly, as described in [8].

None of the work cited above (except [8]) applies formal design and evaluation techniques. However, in [26] Talcott applies formal techniques to the study of dialects running over unreliable transport, such as UDP. [26] is complementary to our work in a number of ways. First, we have been looking at dialects running over reliable transport, such as TCP. Secondly, we concentrate on a particular attacker model, the on-path attacker with no ability to compromise keys or corrupt participants, and then explore ways of generating dialects and lingos that can be used to defend against this type of attacker. In [26], however, the dialects are relatively simple, but the attacker models are explored in more detail, giving the attacker a set of specific actions that it can perform. Like our model, the work in [26] is formalized in the Maude formal specification language, providing a clear potential synergy between the two models in the future.

1.2 Contributions of this paper

1. We present a theory of lingos that adds important new *properties*, *transformations* and *composition operations* to the notion of lingo introduced in [8]. The main goal of this theory is to improve the security of protocol dialects.
2. We show that these new lingo transformations and composition operations are *formal patterns* [19] that guarantee desired properties and replace and greatly simplify two earlier formal patterns for dialects in [8]. These lingo transformations provide new compositional techniques for creating new lingos and dialects.
3. We provide formal executable specifications in Maude [5] for various lingos, their transformations and compositions, and dialects. For more information on them, see [9].

Section 2 substantially extends lingos beyond [8] and provides a wealth of new results about lingos and lingo transformations. Section 3 provides new lingo composition operations and proves their properties. Section 4 recalls the dialect notion and illustrates it with various examples. Section 5 concludes the paper.

2 Lingos

A lingo is a data transformation f between two data types D_1 and D_2 with a one-sided inverse transformation g. The transformation f is *parametric* on a secret parameter value a belonging to a parameter set A. For each parameter a, data from D_1 is transformed into data of D_2, which, using the same parameter

a, can be transformed back into the original data from D_1 using the one-sided inverse g. In all our applications, lingos will be used to transform the payload of a message of sort D_1 appearing in some protocol $\mathcal{P}$ whereas D_2 will then be the data type of transformed payloads. Such transformed payloads can then be sent, either as a single message or as a sequence of messages, to make it hard for malicious attackers to interfere with the communication of honest participants, who are the only ones knowing the current parameter $a \in A$. The point of a lingo is that when such transformed messages are received by an honest participant they can be transformed back using g (if they were broken into several packets they should first be reassembled) to get the original payload in D_1.

Briefly, the set D_1 can be thought of as analogous to the plaintext space, the set D_2 as analogous to the ciphertext space in cryptographic systems, and the parameter set A (also known as the secret parameter set) as analogous to the key space. The elements of A are generally required to be pseudorandomly generated from a secret shared by enclave members, so it can't be guessed in advance, and it is also generally updated with each use, so information gleaned from seeing one dialected message should not provide any help in breaking another. However, it is not necessarily required that an attacker is not be able to guess a after seeing $f(d_1, a)$. The only requirement is that it is not feasible for the attacker to compute $f(d_1, a)$ without having been told that a is the current parameter for $f(d_1, a)$. This is for two reasons. First, the lingo is intended to provide authentication, not secrecy. Secondly, the lingo is only intended to be secure against an on-path attacker [8] that can eavesdrop on traffic but not interfere with it. Thus, if an enclave member sends a message $f(d_1, a)$, the attacker, even if it learns a, can't remove $f(d_1, a)$ from the channel and replace it with $f(d_1', a)$ where d_1' is a message of the attacker's own choosing. It can send $f(d_1', a)$ after $f(d_1, a)$ is sent, but $f(d_1', a)$ may not be accepted if the secret parameter is changed each time the message is sent. The enforcement of the policy on the sharing and updating of parameters is the job of the dialect, which will be discussed in Section 4.

Definition 1 (Lingo). *A* lingo Λ *is a 5-tuple* $\Lambda = (D_1, D_2, A, f, g)$*, where* D_1*,* D_2 *and* A *are sets called, respectively, the* input*,* output *and* parameter *sets,* f*,* g *are functions* $f : D_1 \times A \to D_2$*,* $g : D_2 \times A \to D_1$ *such that*[5] $\forall d_1 \in D_1$*,* $\forall a \in A$*,* $g(f(d_1, a), a) = d_1$*. We call a lingo* non-trivial *iff* D_1*,* D_2 *and* A *are non-empty sets. In what follows, all lingos considered will be non-trivial.*

Note that a lingo $\Lambda = (D_1, D_2, A, f, g)$ *is just a three-sorted* (Σ, E)*-algebra, with sorts*[6] D_1*,* D_2 *and* A*, function symbols* $f : D_1 \times A \to D_2$ *and* $g : D_2 \times A \to D_1$*, and* E *the single* Σ*-equation* $g(f(d_1, a), a) = d_1$*.*

[5] The equality $g(f(d_1, a), a) = d_1$ can be generalized to an equivalence $g(f(d, a), a) \equiv d$. The generalization of lingos to allow for such message equivalence is left for future work.

[6] Note the slight abuse of notation: in Σ, D_1, D_2 and A are uninterpreted sort *names*, whereas in a given lingo $\Lambda' = (D_1', D_2', A', f', g')$, such sorts are respectively interpreted as *sets* D_1', D_2' and A' and, likewise, the uninterpreted function *symbols* f and g in Σ are interpreted as actual *functions* f' and g' in a given lingo.

Note the asymmetry between f and g, in the lingo definition, since we do not have an equation of the form $\forall d_2 \in D_2$, $\forall a \in A$, $f(g(d_2, a), a) = d_2$. The reason for this asymmetry is that, given $a \in A$, the set $\{f(d_1, a) \mid d_1 \in D_1\}$ may be a *proper* subset of D_2. However, we show in [9] that the equation $f(g(d_2, a), a) = d_2$ does hold for any $d_2 \in \{f(d_1, a) \mid d_1 \in D_1\}$. Other results and proofs are included in [9]. Of course, for the special case of a lingo such that $\forall a \in A$, $D_2 = \{f(d_1, a) \mid d_1 \in D_1\}$, the equation $\forall d_2 \in D_2$, $\forall a \in A$, $f(g(d_2, a), a) = d_2$ will indeed hold. In particular, the set equality $D_2 = \{f(d_1, a) \mid d_1 \in D_1\}$ holds for all $a \in A$ in the following example.

Example 1 (The XOR{n} Lingo). Let $\Lambda_{xor}\{n\} = (\{0,1\}^n, \{0,1\}^n, \{0,1\}^n, \oplus, \oplus)$, with $\_ \oplus \_ : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ the bitwise exclusive or operation.

Note that $\Lambda_{xor}\{n\}$ is *parametric* on $n \in \mathbb{N} \setminus \{0\}$. That is, for each choice of $n \geq 1$ we get a corresponding lingo. This is a common phenomenon: for many lingos, D_1, D_2 and A are not fixed sets, but *parametrized data types*, so that for each choice of the parameter we get a corresponding instance lingo. For a more detailed explanation of *parametrized data types*, see [9].

Example 2 (XOR-BSEQ Lingo for Bit-sequences). Yet another variation on the same theme is to define the $\Lambda_{xor.BSeq}$ lingo, whose elements are *bit sequences* of arbitrary length.

Note that $\{0,1\}^n$ is not the only possible parametrized data type on which a lingo based on the *xor* operation could be based. A different parametrized data type can have as elements the *finite subsets* of a parameter set D, and where the *xor* operation is interpreted as the *symmetric difference* of two finite subsets of D. A finite subset, say, $\{a, b, c\} \subseteq D$, with $a, b, c \in D$ different elements, can then be represented as the expression a *xor* b *xor* c. For D a finite set, there is an isomorphism between its power set $\mathcal{P}(D)$ with symmetric difference, and the function set $[D \to \{0,1\}]$ with pointwise *xor* of predicates $p, q \in [D \to \{0,1\}]$, i.e., p *xor* $q = \lambda d \in D.\ p(d)$ *xor* $q(d)$, which when $|D| = n$ is isomorphic to $\{0,1\}^n$ with pointwise *xor*.

Example 3 (Divide and Check (D&C) Lingo). Let $\Lambda_{D\&C} = (\mathbb{N}, \mathbb{N} \times \mathbb{N}, \mathbb{N}, f, g)$, given $\forall n, a, x, y \in \mathbb{N}$ then:

- $f(n, a) = (quot(n + (a + 2), a + 2), rem(n + (a + 2), a + 2))$
- $g((x, y), a) = (x \cdot (a + 2)) + y - (a + 2)$

where *quot* and *rem* denote the quotient and remainder functions on naturals.

The idea of the $\Lambda_{D\&C}$ lingo is quite simple. Given a parameter $a \in \mathbb{N}$, an input number n is transformed into a pair of numbers (x, y): the quotient x of $n+a+2$ by $a+2$ (this makes sure that $a+2 \geq 2$), and the remainder y of $n+a+2$ by $a+2$. The meaning of "divide" is obvious. The meaning of "check" reflects the fact that a receiver of a pair (x, y) who knows a can check $x = quot((x \cdot (a+2)) + y, a+2))$ and $y = rem((x \cdot (a+2)) + y, a+2))$, giving some assurance that the pair (x, y) was obtained from $n = (x \cdot (a+2)) + y - (a+2)$ and has not been tampered with.

This is an example of an *f-checkable* lingo (see Section 2.1 below), whereas no such check is possible for any of the isomorphic versions of the Λ_{xor} lingo (see again Section 2.1 below).

For a detailed explanation with Maude formal executable specifications of Examples 1, 2 and 3, we refer the reader to [9].

2.1 f-Checkable Lingos and the $\Lambda \mapsto \Lambda^\sharp$ Lingo Transformation

In this section we discuss lingos where the validity of the generated payloads can be checked by the dialect. We define a lingo transformation that substantially reduces the probability of an attacker successfully forging a message.

Definition 2 (f-Checkable Lingo). *A Lingo* $\Lambda = (D_1, D_2, A, f, g)$ *is called* f-checkable *iff* $\forall a \in A,\ \exists d_2 \in D_2$ *s.t.* $\nexists\, d_1 \in D_1$ *s.t.* $d_2 = f(d_1, a)$.

Above, Λ is understood, not as a fixed lingo, but as an equational theory specifying lingos. Therefore, the "f" in "f-Checkable Lingo" is an uninterpreted function symbol. However, once an actual interpretation for the theory Λ is given, and therefore for D_1, D_2, A, f and g, f becomes interpreted as the concrete function specified by that interpretation.

Note that the Λ_{xor} lingo of Example 1 is not f-checkable. We call a lingo $\Lambda = (D_1, D_2, A, f, g)$ *symmetric* iff (D_2, D_1, A, g, f) is also a lingo. Obviously, Λ_{xor} is symmetric. Any symmetric lingo is *not* f-checkable, since any $d_2 \in D_2$ satisfies the equation $f(g(d_2, a), a) = d_2$.

Example 4. The $\Lambda_{D\&C}$ lingo of Example 3 is f-checkable. Indeed, for each $a \in \mathbb{N}$, any (x, y) with $y \geq a + 2$ cannot be of the form $(x, y) = f(n, a)$ for any $n \in \mathbb{N}$. This is a cheap, easy check. The actual check that a d_1 exists, namely, the check $(x, y) = f(g((x, y), a), a)$ provided by [9] (where *proofs of all theorems are given*), was already explained when making explicit the meaning of "C" in $\Lambda_{D\&C}$.

The following theorem explains how to build an f-checkable lingo. The proof is in [9].

Theorem 1. *Let* $\Lambda = (D_1, D_2, A, f, g)$ *be a lingo, where we assume that* D_1, D_2 *and* A *are computable data-types and* $|D_1| \geq 2$. *Then,* $\Lambda^\sharp = (D_1, D_2 \times D_2, A \otimes A, f^\sharp, g^\sharp)$ *is an* f*-checkable lingo, where:*

- $A \otimes A = A \times A \setminus id_A$, *with* $id_A = \{(a, a) \in A^2 | a \in A\}$
- $f^\sharp(d_1, (a, a')) = (f(d_1, a), f(d_1, a'))$
- $g^\sharp((d_2, d_2'), (a, a')) = g(d_2, a)$.

Example 5 (f-checkable transformed version of $\Lambda_{xor}\{n\}$*).* The lingo transformation $\Lambda \mapsto \Lambda^\sharp$ maps $\Lambda_{xor}\{n\}$ in Example 1 to the f-checkable lingo $\Lambda^\sharp_{xor}\{n\} = (\{0,1\}^n, \{0,1\}^n \times \{0,1\}^n, \{0,1\}^n \otimes \{0,1\}^n, \oplus^\sharp, \oplus^\sharp)$.

2.2 Malleable Lingos

A cryptographic function is called *malleable* if an intruder, *without knowing the secret key*, can use one or more existing encrypted messages to generate another, related, message also encrypted with the same secret key. In a similar way, if a lingo Λ is what below we call *malleable*, an intruder can disrupt the communication between an honest sender *Alice* and an honest receiver *Bob* in a protocol $\mathcal{P}$ whose messages are modified by means of a lingo[7] Λ by producing a message *compliant* with a secret parameter $a \in A$ of Λ supposedly sent from *Alice* to *Bob* (who share the secret parameter a), but actually sent by the intruder.

Definition 3 (Malleable Lingo). *Let $\Lambda = (D_1, D_2, A, f, g)$ be a lingo, which has been formally specified as the initial algebra of an equational theory $(\Sigma_\Lambda, E_\Lambda)$. Λ is called* malleable *iff there exists a* recipe*, i.e., a Σ_Λ-term $t(x, y)$ of sort D_2 with free variables x, y of respective sorts D_2 and A, as well as a subset $A_0 \subseteq A$ such that $\forall d_1 \in D_1,\ \forall a \in A,\ \forall a' \in A_0$,*

1. *$f(d_1, a) \neq t(f(d_1, a), a')$, where, by convention, $t(f(d_1, a), a')$ abbreviates the substitution instance $t\{x \mapsto f(d_1, a), y \mapsto a'\}$*
2. *$\exists d_1' \in D_1$ such that $t(f(d_1, a), a') = f(d_1', a)$.*

Example 6. The lingo $\Lambda_{xor}\{n\}$ from Example 1 is *malleable*. The recipe $t(x, y)$ is $x \oplus y$, and $A_0 = \{0,1\}^n \setminus \{\overrightarrow{0}\}$. Then, for any $a' \in A_0$ we have, $t(f(d_1, a), a') = d_1 \oplus a \oplus a' \neq d_1 \oplus a$, which meets condition (1), since $a' \neq \overrightarrow{0}$ and $\oplus$ is a group addition; and it meets condition (2), since $t(f(d_1, a), a') = f(d_1 \oplus a', a)$. An attacker needs not know $d_1 \oplus a'$.

The notion of an f-checkable lingo of Section 2.1 provides a first line of defense against an intruder trying to generate a payload compliant with a secret parameter a. However, the example below shows that an f-checkable lingo may be malleable.

Example 7. The lingo $\Lambda^\sharp_{xor}\{n\}$ from Example 5 with $n \geq 2$ is malleable with recipe $t((x_1, x_2), (y_1, y_2)) = (x_1 \oplus y_1, x_2 \oplus y_1)$ and $A_0 = \{(y_1, y_2) \in \{0,1\}^n \otimes \{0,1\}^n \mid y_1 \neq \overrightarrow{0}\}$. Indeed, for any $d_1 \in \{0,1\}^n$, $(a_1, a_2) \in \{0,1\}^n \otimes \{0,1\}^n$, and $(a_1', a_2') \in A_0$, we have:

1. $t(f^\sharp(d_1, (a_1, a_2)), (a_1', a_2'))) = (d_1 \oplus a_1 \oplus a_1', d_1 \oplus a_2 \oplus a_1')$ so that, by $a_1' \neq \overrightarrow{0}$, we have $f^\sharp(d_1, (a_1, a_2)) \neq t(f^\sharp(d_1, (a_1, a_2)), (a_1', a_2')))$.
2. From $t(f^\sharp(d_1, (a_1, a_2)), (a_1', a_2'))) = (d_1 \oplus a_1 \oplus a_1', d_1 \oplus a_2 \oplus a_1')$ it trivially follows that there is a d_1', namely, $d_1' = d_1 \oplus a_1'$, such that $f^\sharp(d_1', (a_1, a_2)) = (d_1 \oplus a_1' \oplus a_1, d_1 \oplus a_1' \oplus a_2) = t(f^\sharp(d_1, (a_1, a_2)), (a_1', a_2')))$.

Therefore, $\Lambda^\sharp_{xor}\{n\}$ with $n \geq 2$ is malleable. An attacker needs not know d_1', only needs to use $f^\sharp(d_1, (a_1, a_2))$, choose $(a, a') \in A_0$, and then use t.

[7] I.e., honest participants that use a *dialect* $\mathcal{D}_\Lambda(\mathcal{P})$ based on $\mathcal{P}$ and Λ to modify their messages (see §4).

Examples 6–7 suggest the following generalization of Theorem 1 above. The proof is in [9].

Theorem 2. *Let $\Lambda = (D_1, D_2, A, f, g)$ be a lingo which has been formally specified as the initial algebra of an equational theory $(\Sigma_\Lambda, E_\Lambda)$ and is such that: (i) $D_2 \subseteq D_1$, (ii) there is a subset $A_0 \subseteq A$ with at least two different elements a_0, a_1 and such that $\forall d_1 \in D_1$, $\forall a' \in A_0$, $f(d_1, a') \neq d_1$, and (iii) $\forall\ d_1 \in D_1$, $\forall a \in A$, $\forall a' \in A_0$, $f(f(d_1, a), a') = f(f(d_1, a'), a)$. Then both Λ and $\Lambda^\sharp$ are malleable.*

Remark 1. Note that we don't require a specification of the relation between the two plaintexts d_1 and d_1' (as is done in the informal definition given at the beginning of this section) except for both belonging to D_1. This still captures the way in which an attacker would take advantage of a malleable function but is more straightforward to reason about.

Remark 2. Note that the recipes are used with two arguments: the ciphertext $f(d_1, a)$ and a parameter a' in A_0. It is also possible to ignore $f(d_1, a)$, i.e., for some lingos the attacker may generate a random cyphertext that, due to malleablity, can be mistaken for a genuine message by the receiver. More generally, other notions of lingo malleability besides that in Def. 3 may be studied.

Remark 3. Malleability is in general undesirable, but some threats introduced by it are mitigated by the on-path attacker model. For example, the on-path attacker cannot substitute a message $f(d_1', a)$ for the message $f(d_1, a)$ from which it was derived. But there are also other cases, for example if we allow repeated parameters, or encounter a situation in which an attacker can send a random message which a receiver could mistake for some $f(d_1, a)$, as in Remark 2.

Finally, one would like to transform a possibly malleable lingo Λ into a non-malleable lingo Λ'. Developing general methods for proving that a lingo is non-malleable is a topic for future research. Note, however, that in cryptography non-malleability of a cryptosystem has been shown to be equivalent to authenticated encryption [2, 3]. This may or may not be the case for protocol dialects, which use a weaker attacker model. In [9], we discuss *authenticating lingos* and a transformation mapping a lingo to an authenticating one which we conjecture can be used to build non-malleable lingos out of malleable ones.

3 Lingo Compositions

Lingos can be composed in various ways to obtain new lingos. Lingo compositions provide modular, automated methods of obtaining new lingos from existing ones. Such compositions are often stronger, i.e., harder to compromise by an attacker, than the lingos so composed. We define here two such lingo composition operations, namely, *horizontal* and *functional* composition of lingos.

3.1 Horizontal Composition of Lingos

Given a finite family of lingos $\overrightarrow{\Lambda} = \{\Lambda_i\}_{1 \leq i \leq k}$, $k \geq 2$, all sharing the same input data type D_1, their *horizontal composition*, denoted $\bigoplus \overrightarrow{\Lambda}$ is intuitively their union, i.e., in $\bigoplus \overrightarrow{\Lambda}$ the input data type D_1 remains the same, D_2 is the union of the $D_{2.i}$, $1 \leq i \leq k$, and A is the disjoint union of the A_i, $1 \leq i \leq k$. $\bigoplus \overrightarrow{\Lambda}$ is a more complex lingo than each of its component lingos Λ_i, making it harder to compromise by an attacker, because it becomes a hydra with k heads: quite unpredictably, sometimes behaves like some Λ_i and sometimes like another Λ_j. Furthermore, it has a bigger parameter space, since $|A| = \Sigma_{1 \leq i \leq k} |A_i|$.

Definition 4 (Horizontal Composition). *Let $\overrightarrow{\Lambda}$ be a finite family of lingos $\overrightarrow{\Lambda} = \{\Lambda_i\}_{1 \leq i \leq k}$, $k \geq 2$, all having the same input data type D_1, i.e., $\Lambda_i = (D_1, D_{2.i}, A_i, f_i, g_i)$, $1 \leq i \leq k$. Let $\overrightarrow{d}_0 = (d_{0.1}, \ldots, d_{0.k})$ be a choice of default $D_{2.i}$-values, $d_{0.i} \in D_{2.i}$, $1 \leq i \leq k$. Then, the* horizontal composition *of the lingos $\overrightarrow{\Lambda}$ with default $D_{2.i}$-values $\overrightarrow{d}_0$ is the lingo:*

$$\bigoplus_{\overrightarrow{d_0}} \overrightarrow{\Lambda} = (D_1, \bigcup_{1 \leq i \leq k} D_{2.i}, \bigcup_{1 \leq i \leq k} A_i \times \{i\}, \oplus \overrightarrow{f}, \oplus \overrightarrow{g})$$

where, for each $d_1 \in D_1$, $d_2 \in \bigcup_{1 \leq i \leq k} D_{2.i}$, and $a_i \in A_i$, $1 \leq i \leq k$,

1. $\oplus \overrightarrow{f} = \lambda(d_1, (a_i, i)).f_i(d_1, a_i)$,
2. $\oplus \overrightarrow{g} = \lambda(d_2, (a_i, i))$. **if** $d_2 \in D_{2.i}$ **then** $g_i(d_i, a_i)$ **else** $g_i(d_{0.i}, a_i)$.

$\bigoplus_{\overrightarrow{d_0}} \overrightarrow{\Lambda}$ is indeed a lingo: $\oplus \overrightarrow{g}(\oplus \overrightarrow{f}(d_1, (a_i, i)), (a_i, i)) = g_i(f_i(d_1, a_i), a_i) = d_1$.

We have already pointed out that, in practice, two protocol participants using a lingo to first modify and then decode a payload d_1 that, say, Alice sends to Bob, agree on a secret parameter $a \in A$ by agreeing on a secret random number n, because both use a function $param : \mathbb{N} \to A$ to get $a = param(n)$. This then poses the practical problem of synthesizing a function $\oplus \overrightarrow{param} : \mathbb{N} \to \bigcup_{1 \leq i \leq k} A_i \times \{i\}$ for $\bigoplus_{\overrightarrow{d_0}} \overrightarrow{\Lambda}$ out of the family a functions $\{param_i : \mathbb{N} \to A_i\}_{1 \leq i \leq k}$ used in each of the lingos Λ_i. Furthermore, different lingos in the family $\overrightarrow{\Lambda}$ may have different degrees of strength against an adversary. This suggests favoring the choice of stronger lingos over that of weaker lingos in the family $\overrightarrow{\Lambda}$ to achieve a function $\oplus \overrightarrow{param}$ that improves the overall strength of $\bigoplus_{\overrightarrow{d_0}} \overrightarrow{\Lambda}$. This can be achieved by choosing a *bias vector* $\overrightarrow{\beta} = (\beta_1, \ldots, \beta_k) \in \mathbb{N}^k_{>0}$, so that, say, if lingo Λ_i is deemed to be stronger than lingo Λ_j, then the user chooses $\beta_i > \beta_j$. That is, $\overrightarrow{\beta}$ specifies a *biased* die with $k \geq 2$ faces, so that the die will show face j with probability $\frac{\beta_j}{\Sigma_{1 \leq i \leq k} \beta_i}$. Therefore, we can use a pseudo-random function $throw_{\overrightarrow{\beta}} : \mathbb{N} \to \{1, \ldots, k\}$ simulating a sequence of throws of a k-face die with bias $\overrightarrow{\beta}$ to get our desired function $\oplus \overrightarrow{param}$ as the function:

$$\oplus \overrightarrow{param}(n) = (param_{throw_{\overrightarrow{\beta}}(n)}(n), throw_{\overrightarrow{\beta}}(n)).$$

Example 8. To see how horizontal composition can strengthen lingos, consider the D&C lingo described in Example 3, in which $f(n,a) = (x,y)$ where $x = quot(n+(a+2), a+2)$ and $y = rem(n+(a+2), a+2)$. We note that any choice of $0 \leq y < a+2$ will pass the f-check, so the choice of $y = 0$ or 1 will always pass the f-check. To counter this, let *reverse*-D&C be the lingo with $f(n,a) = (y,x)$, where y and x are computed as in D&C. The attacker's best strategy in reverse-D&C is the opposite of that in D&C. Thus composing D&C and reverse-D&C horizontally with a bias vector $(.5, .5)$ means that the attacker strategy of choosing the first (respectively, second) element of the output to be 0 succeeds with probability 0.5 in any particular instance, as opposed to probability 1 for D&C by itself.

The following result follows easily from the definition of horizontal composition.

Lemma 1. *If each Λ_i in $\overrightarrow{\Lambda} = \{\Lambda_i\}_{1 \leq i \leq k}$, $k \geq 2$, is f-checkable, then $\bigoplus_{\overrightarrow{d_0}} \overrightarrow{\Lambda}$ is also f-checkable.*

Example 9 (Horizontal composition of XOR-BSeq and D&C Lingos). The lingos $\Lambda_{xor.BSeq}$ of Example 2 and $\Lambda_{D\&C}$ of Example 3 share the same D_1, namely, $\mathbb{N}$. Therefore, they do have a horizontal composition $\Lambda_{xor.BSeq} \oplus_{\overrightarrow{d}_0} \Lambda_{D\&C}$ for any choice of $\overrightarrow{d}_0$.

We refer the reader to [9] for details on the executable specification of horizontal composition in Maude.

3.2 Functional Composition of Lingos

Given two lingos Λ and Λ' such that the output data type D_2 of Λ coincides with the input data type of Λ', it is possible to define a new lingo $\Lambda \odot \Lambda'$ whose f and g functions are naturally the compositions of f and f' (resp. g' and g) in a suitable way. Moreover, Λ and Λ' can be chosen so that $\Lambda \odot \Lambda'$ is harder to compromise than either Λ or Λ'.

Definition 5 (Functional Composition). *Given lingos $\Lambda = (D_1, D_2, A, f, g)$ and $\Lambda' = (D_2, D_3, A', f', g')$, their* functional composition *is the lingo $\Lambda \odot \Lambda' = (D_1, D_3, A \times A', f \cdot f', g * g')$, where for each $d_1 \in D_1$, $d_3 \in D_3$, and $(a, a') \in A \times A'$,*

- $f \cdot f'(d_1, (a, a')) =_{def} f'(f(d_1, a), a')$,
- $g * g'(d_3, (a, a')) =_{def} g(g'(d_3, a'), a)$.

$(D_1, D_3, A \times A', f \cdot f', g * g')$ is indeed a lingo, since we have:

$$\begin{aligned} g * g'(f \cdot f'(d_1, (a, a')), (a, a')) &= g(g'(f'(f(d_1, a), a'), a'), a) \\ &= g(f(d_1, a), a) = d_1. \end{aligned}$$

Lemma 2. *Given lingos $\Lambda = (D_1, D_2, A, f, g)$ and $\Lambda' = (D_2, D_3, A', f', g')$, if Λ' is f-checkable, then $\Lambda \odot \Lambda'$ is also f-checkable.*

Example 10 (Functional composition of XOR-BSeq and D&C Lingos). The lingos $\Lambda_{xor.BSeq}$ of Example 2 and $\Lambda_{D\&C}$ of Example 3 are functionally composable as $\Lambda_{xor.BSeq} \odot \Lambda_{D\&C}$, because $\mathbb{N}$ is both the output data type of $\Lambda_{xor.BSeq}$ and the input data type of $\Lambda_{D\&C}$. Note that, by Lemma 2 above, and Theorem 1 above, $\Lambda_{xor.BSeq} \odot \Lambda_{D\&C}$ is f-checkable, in spite of $\Lambda_{xor.BSeq}$ not being so.

We refer the reader to [9] for details on the executable specification of functional composition in Maude.

4 Lingo Transformations and Dialects as Formal Patterns

We make the notion of a *formal pattern* as a transformation of rewrite theories explicit and explain how it applies to both lingos and dialects. Since dialects are parametric on the chosen lingo and the chosen protocol, we begin by explaining how protocols are formalized as generalized actor rewrite theories. With all these notions in hand we can explain in detail the *lingo formal pattern* and its rewriting logic semantics and give examples of different dialects for a given protocol based on the choice of various lingos, sometimes themselves compositions of simpler ones. Finally, we explain how both the lingo formal pattern, lingos and dialect formal patterns are specified as *formal executable specifications* in the Maude rewriting logic language [5], a theme further developed in [9].

4.1 Protocols as Generalized Actor Rewrite Theories

Actors [1] are a widely used model for distributed systems, in which concurrent objects communicate through asynchronous message passing. When an actor consumes a received message it can change its state, send new messages, and create new actors. Actor systems can be formally specified and executed as *actor rewrite theories* [17]. In *rewriting logic* [16] a concurrent system can be formally specified and executed as a *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$, with (Σ, E) an equational theory with function symbols Σ and equations E that specifies the concurrent system's *states* as elements of the initial algebra (algebraic data type) $\mathbb{T}_{\Sigma/E}$ of (Σ, E), and where the system's *local concurrent transitions* are specified by *rewrite rules* in R of the form $u(x_1, \ldots, x_n) \rightarrow v(x_1, \ldots, x_n)$, where $u(x_1, \ldots, x_n)$ is a Σ-expression identifying those local state fragments that can make a local transition to a state fragment of the form $v(x_1, \ldots, x_n)$. Under natural assumptions, a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is *executable* by rewriting. Maude [5] is a declarative programming language whose programs, called *system modules*, are rewrite theories declared as `mod` (Σ, E, R) `endm`. Maude has a functional sublanguage of *functional modules*, which are equational theories declared as `fmod` (Σ, E) `endfm`, specifying an algebraic data type $\mathbb{T}_{\Sigma/E}$. For example, (more on this in Section 4.3) lingos are algebraic data types that can be naturally specified as functional modules in Maude.

Any *communication protocol* $\mathcal{P}$ can be formally specified and programmed as a *generalized actor rewrite theory* $\mathcal{P} = (\Sigma, E \cup B, R)$ [12]. The rewrite rules

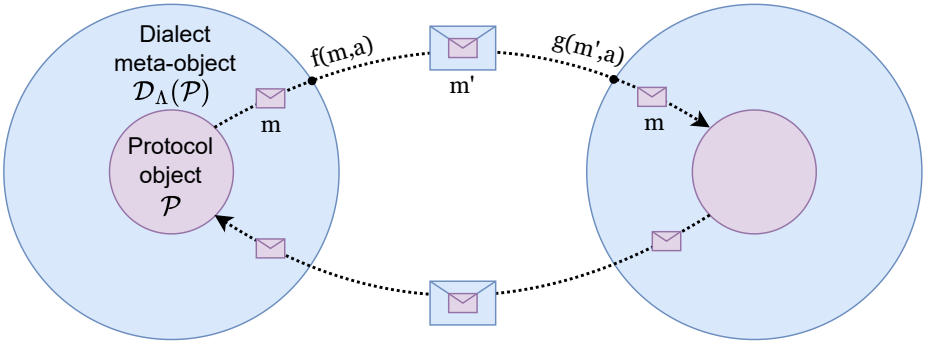


Fig. 1: Dialect meta-objects (enhanced from [8]).

of an actor rewrite theory are of the form:

$$(to\ B\ from\ A : M)\ <\ B : Cl\ |\ atts\ > \rightarrow <\ B : Cl\ |\ atts'\ >\quad msgs\quad actors.$$

They specify how actor B of class Cl, upon receiving a message from actor A, can change its local state (the attribute-value pairs *atts*) to *atts′* and may generate several new messages *msgs* and several new actors *actors*. Any such rule specifies an *asynchronous* local concurrent transition in a distributed state (called a *configuration*) which is a multiset of objects and messages built up with a multiset union operation _ _ (juxtaposition) satisfying the axioms B of associativity, commutativity and empty multiset $\emptyset$ as unit element [17]. Generalized actor rewrite theories allow also rules of the form:

$$<\ B : Cl\ |\ atts\ > \rightarrow <\ B : Cl\ |\ atts'\ >\quad msgs\quad objs.$$

describing *active actors* that can also change their state and generate new messages and actors without receiving a message.

Because of space limitations, we refer the reader to [9] for details on how lingos can be used as Formal Patterns.

4.2 The Dialect Transformation as a Formal Pattern

The *dialect transformation* can be formalized as a formal pattern of the form:

$$\mathcal{D} : GralActorRewThs \times PLingos \ni (\mathcal{P}, \Lambda_p) \mapsto \mathcal{D}_{\Lambda_p}(\mathcal{P}) \in GralActorRewThs$$

where $\mathcal{P}$ is a protocol, i.e., a rewrite theory in the class[8] *GralActorRewriteThs* of generalized actor rewrite theories, and $\mathcal{D}_{\Lambda_p}(\mathcal{P})$ is another protocol obtained from $\mathcal{P}$ by means of Λ_p, where Λ_p is a *lingo with parameter function* p in the class *PLingos*, that is, Λ_p adds to a lingo Λ a parameter function $p : \mathbb{N} \rightarrow A$.

The essential ideas about the dialect $\mathcal{D}_{\Lambda_p}(\mathcal{P})$ are summarized in Figure 1. Suppose that Alice and Bob are actors in the protocol $\mathcal{P}$ and Alice sends payload m to Bob in a message. What happens instead in $\mathcal{D}_{\Lambda_p}(\mathcal{P})$ is that *Alice and Bob*

[8] Hereafter we assume generalized actor systems whose actors do not create new actors.

$$
\begin{aligned}
&in\ : < O_1 : DC \mid in.buffer : M_{in}, atts' > \ (to\ O_1\ from\ O_2\ :\ P) \\
&\rightarrow\ < O_1 : DC \mid in.buffer : (M_{in} \cup (to\ O_1\ from\ O_2\ :\ P), atts' > \\
&out\ : < O_1 : DC \mid conf : (< O_1 : C \mid atts > \ (to\ O_2\ from\ O_1 : P) \cup\ M), \\
&\qquad\qquad peer.counters : R,\ atts' > \\
&\rightarrow\ < O_1 : DC \mid conf : (< O_1 : C \mid atts > \ M \setminus (to\ O_2\ from\ O_1\ :\ P)), \\
&\qquad\qquad \text{update}(peer.counters : R, atts') > \\
&\qquad (to\ O_2\ from\ O_1\ :\ f(P, param(R[O_2]))) \\
&deliver\ : < O_1 : DC \mid conf : (< O_1 : C \mid atts > M),\ in.buffer : M_{in}, \\
&\qquad\qquad peer.counters : R,\ atts' > \\
&\rightarrow\ < O_1 : DC \mid conf : (< O_1 : C \mid atts > M \cup \{g(M_{selected}, \text{param}(R[O_2]))\}, \\
&\qquad\qquad in.buffer : (M_{in} \setminus M_{selected}), \\
&\qquad\qquad update(peer.counters : R, atts') > \\
&\quad if\ size(M_{in}) \geq egressArity\ \wedge M_{selected} := take(egressArity, M_{in}) \wedge \\
&\quad\ fromOidTag(M_{selected}, R) = true\ .
\end{aligned}
$$

Fig. 2: Meta-Actor Rewrite Rules.

behave as before, that is, their original protocol $\mathcal{P}$ remains the same. However, unbeknownst to Alice and Bob, their communication is now *mediated* by their corresponding *meta-actors* (also named Alice, resp. Bob) inside which they are now *wrapped*. Alice's meta-actor *transforms* the original payload m into $f(m, a)$, and Bob's meta-actor *decodes* $f(m, a)$ into $g(f(m, a), a) = m$ using Λ_p and the *shared secret parameter* a. Of course, if Bob replies to Alice with another message, the roles are exchanged: now Bob's meta-actor will transform the new payload m' to $f(m', a')$ and Alice's meta-actor with get back m' because they now share a *new* secret parameter a'. Since parameters change all the time, dialects become *moving targets* for an attacker.

But what is a *meta-actor*? In rewriting logic a meta-actor of class *Dialect* will have the general form: $< o : Dialect \mid conf :< o : C \mid atts >, datts >$. That is, one of the attributes in the state of the meta-actor is called *conf* and consists of a *configuration* of objects and messages with a *single object*, namely, $< o : C \mid atts >$ and, at various times, either an incoming message addressed to o, or outgoing messages sent by o to other actors.

datts denotes the rest of the dialect meta-actor's state (i.e., it remaining attribute-value pairs). What are they? Besides the object attribute *conf*, *Dialect* meta-actors have the following additional attributes: (i) *in.buffer*: a buffer of incoming messages to which the lingo's decoding function g has not yet been applied; and (ii) *peer.counters*: a map from actor names to natural numbers that serves as a counter for the number of messages exchanged. It is also used to generate (using the parameter function p) the new secret parameters used by the lingo's f and g functions for communicating with other actors.

The *behavior* of a *Dialect* meta-actor is specified by the rewrite rules in Figure 2. These can be summarized as follows: (i) rule *in* collects incoming messages for the wrapped actor in the *in.buffer* attribute; (ii) rule *out* processes messages produced by the wrapped protocol, sending the transformed messages into the network, by applying the lingo's f function with the corresponding parameter; and (iii) rule *deliver* processes messages stored in the buffer, by applying the lingo's g function with the proper parameter, delivering the original version of the message to the protocol. Rules *out* and *deliver* apply f and g, respectively, and supply the parameter a by calling the lingo's param function, which computes a parameter $a \in A$ from a number $n \in \mathbb{N}$. The dialect meta-object keeps parameters changing by supplying to the param function the current message number from the map in *peer.counter*. This turns dialects into moving targets.

Example 11 (Transforming protocols by means of Protocol Dialects). Let us leverage the various lingos introduced in previous sections to incrementally construct more sophisticated, and possibly more secure, protocols through the use of dialects. We use as core protocol a parametric version of MQTT, denoted $Mqtt\{D\}$ and explained in [9], where D represents the data-type of the message payloads (e.g., bit-sequences of $n \in \mathbb{N}$). Furthermore, we denote any given lingo as $\Lambda_{id}\{D\}$, where id identifies the lingo and D indicates the input data type. For instance, $\Lambda_{D\&C}\{BitSEQ\{N\}\}$ refers to the divide and check lingo over bit-sequences of length $n \in \mathbb{N}$.

As a first transformation, we can use the lingo presented in Example 1. To do so, we can instantiate a dialect $\mathcal{D}_{\Lambda_p}(\mathcal{P})$ over MQTT, using the xor lingo over a payload consisting of 1 byte, i.e. a bit-sequence (or vector) of length 8. The resulting protocol is denoted by $\mathcal{D}_{\Lambda_{XOR}\{BitSEQ\{8\}\}}(Mqtt\{BitSEQ\{8\}\})$.

It is important to note that we can, in a very intuitive way, *upgrade* or adapt the new protocol to meet specific needs. For instance, we can increase the parameter size by just increasing the input parameter. This leads to a new protocol such as $\mathcal{D}_{\Lambda_{XOR}\{BitSEQ\{256\}\}}(Mqtt\{BitSEQ\{256\}\})$, which operates over 32-byte payloads.

A similar approach can be applied to both parameters of a Dialect: the lingo and the underlying protocol. For instance, consider replacing the XOR{N} lingo for the XOR-BSEQ lingo introduced in Example 2. These kind of substitutions are valid provided that both the input data type of the lingo and protocol coincide. For this purpose, we can instantiate the protocol's data type to be the natural numbers, that is bit sequences of arbitrary length. Accordingly, the resulting protocol can be expressed as $\mathcal{D}_{\Lambda_{xor.BSeq}}(Mqtt\{Nat\})$. Furthermore, we can change the lingo to the divide and check lingo given in Example 3 in a straightforward way since both XOR-BSEQ and D&C share input data types. The resulting protocol is $\mathcal{D}_{\Lambda_{D\&C}}(Mqtt\{Nat\})$.

Furthermore, we can use lingo transformations to construct more sophisticated dialects. For example, one may horizontally compose XOR-BSEQ with D&C, as illustrated in Example 9. Then, we can use this new lingo to yield a new protocol defined as $\mathcal{D}_{\Lambda_{xor.BSeq} \oplus_{\overline{d}_0} \Lambda_{D\&C}}(Mqtt\{Nat\})$. Another protocol can be obtained by using the functional composition from Example 10. The resulting

protocol would be $\mathcal{D}_{\Lambda_{xor.BSeq} \odot \Lambda_{D\&C}}(Mqtt\{Nat\})$. For a more detailed description of all these instantiations, we refer the reader to [9].

Last but not least, this work provides a substantial extension and simplification of the previous theory of dialects in [8] that includes, not only new lingo properties and formal patterns, but the *replacement* of the former *horizontal*, resp. *vertical*, dialect composition operations in [8] by the simpler and more efficient notions of *horizontal*, resp. *functional*, lingo composition operations.

4.3 Lingos, Dialects, and their Formal Patterns in Maude

As shown in [9] and Section 4.2, dialects can be formalized as equational theories with initial semantics, and dialect transformations as formal patterns composing or transforming them. Likewise, we have seen that a dialect $\mathcal{D}_{\Lambda_p}(\mathcal{P})$ is a formal pattern transforming a generalized actor rewrite theory $\mathcal{P}$ into another generalized actor rewrite theory. Since Maude programs are exactly either equational theories with initial semantics, or rewrite theories, both lingos and dialects, as well as their transformations, can be made executable in Maude. In fact, all the theoretical concepts and examples presented in this paper have been formally specified in Maude at the same time that their theoretical properties were proved. But the interest of these formal specifications is not just theoretical. On the one hand, the D-transformation proposed in [13] provides a formal pattern to automatically transform generalized actor rewrite theories into distributed implementations; on the other hand, dialects formalized in Maude can be formally analyzed by various model checking verification techniques, both in Maude itself [5] and, for probabilistic properties, in the QMaude tool [23].

5 Concluding Remarks

Dialects are a resource-efficient approach as a first line of defense against outside attackers. Lingos are invertible message transformations used by dialects to thwart attackers from maliciously disrupting communication. In this paper we propose new kinds of lingos and make them stronger through lingo transformations, including composition operations, all of them new. These lingo transformations and composition operations are *formal patterns* with desirable security properties. We also propose a refined and simpler definition for dialects by replacing two former dialect composition operations in [8] by considerably simpler and more efficient lingo composition operations. All these concepts and transformations have been formally specified and made executable in Maude.

Our next step is to explore lingos in action as they would be used by dialects in the face of an on-path attacker. We have been developing a formal intruder model that, when combined with the dialect specifications provided in this paper, can be used for statistical and probabilistic model checking of dialects. This model provides the capacity to specify the probability that the intruder correctly guesses secret parameters. For this we are using QMaude [23], which supports probabilistic and statistical model checking analysis of Maude specifications.

Acknowledgements We cordially thank Daniel Galán for his important contributions to the early development of these ideas in [8]. We also thank Zachary J Flores for his comments. S. Escobar and V. García have been partially supported by the grant CIPROM/2022/6 funded by Generalitat Valenciana and by the grant PID2024-162030OB-100 funded by MCIN/AEI/10.13039/501100011033 and ERDF A way of making Europe.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA (1986)
2. Bellare, M., Desai, A., Pointcheval, D., Rogaway, P.: Relations among notions of security for public-key encryption schemes. In: Annual International Cryptology Conference. pp. 26–45. Springer (1998)
3. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. Journal of cryptology **21**(4), 469–491 (2008)
4. Chadha, R., Gunter, C.A., Meseguer, J., Shankesi, R., Viswanathan, M.: Modular preservation of safety properties by cookie-based dos-protection wrappers. In: International Conference on Formal Methods for Open Object-Based Distributed Systems. pp. 39–58. Springer (2008)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Oliet, N.M., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic. Lecture Notes in Computer Science, Springer (July 2007). https://doi.org/10.1007/978-3-540-71999-1, http://dx.doi.org/http://dx.doi.org/10.1007/978-3-540-71999-1
6. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.: Programming open distributed systems in maude. In: Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming. pp. 1–12 (2024)
7. Eckhardt, J., Mühlbauer, T., AlTurki, M., Meseguer, J., Wirsing, M.: Stable availability under denial of service attacks through formal patterns. In: International Conference on Fundamental Approaches to Software Engineering. pp. 78–93. Springer (2012)
8. Galán, D., García, V., Escobar, S., Meadows, C., Meseguer, J.: Protocol dialects as formal patterns. In: European Symposium on Research in Computer Security. pp. 42–61. Springer (2023)
9. García, V., Escobar, S., Meadows, C., Meseguer, J.: A theory of composable lingos for protocol dialects. Tech. rep. (2026), https://doi.org/10.48550/arXiv.2603.19908
10. Gogineni, K., Mei, Y., Venkataramani, G., Lan, T.: Can you speak my dialect?: A Framework for Server Authentication using Communication Protocol Dialects. arXiv preprint arXiv:2202.00500 (2022). https://doi.org/10.48550/ARXIV.2202.00500, https://arxiv.org/abs/2202.00500, publisher: arXiv Version Number: 1
11. Gogineni, K., Mei, Y., Venkataramani, G., Lan, T.: Verify-pro: A framework for server authentication using communication protocol dialects. In: MILCOM 2022-2022 IEEE Military Communications Conference (MILCOM). pp. 450–457. IEEE (2022)

12. Liu, S., Meseguer, J., Ölveczky, P.C., Zhang, M., Basin, D.A.: Bridging the semantic gap between qualitative and quantitative models of distributed systems. Proc. ACM Program. Lang. **6**(OOPSLA2), 315–344 (2022)
13. Liu, S., Sandur, A., Meseguer, J., Ölveczky, P.C., Wang, Q.: Generating correct-by-construction distributed implementations from formal Maude designs. In: Proc. NASA Formal Methods: 12th International Symposium, NFM 2020,. Lecture Notes in Computer Science, vol. 12229, pp. 22–40. Springer (2020)
14. Lukaszewski, D., Xie, G.G.: Towards software defined layer 4.5 customization. In: 2022 IEEE 8th International Conference on Network Softwarization (NetSoft). pp. 330–338. IEEE (2022)
15. Mei, Y., Gogineni, K., Lan, T., Venkataramani, G.: MPD: moving target defense through communication protocol dialects. In: García-Alfaro, J., Li, S., Poovendran, R., Debar, H., Yung, M. (eds.) Security and Privacy in Communication Networks - 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6-9, 2021, Proceedings, Part I. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 398, pp. 100–119. Springer (2021). https://doi.org/10.1007/978-3-030-90019-9_6
16. Meseguer, J.: Conditional rewriting logic as a united model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992). https://doi.org/10.1016/0304-3975(92)90182-F, https://doi.org/10.1016/0304-3975(92)90182-F
17. Meseguer, J.: A Logical Theory of Concurrent Objects and its realization in the Maude Language. In: Agha, G., Wegner, P., Yonezawa, A. (eds.) Research Directions in Concurrent Object-Oriented Programming, pp. 314–390. MIT Press (1993)
18. Meseguer, J.: Taming distributed system complexity through formal patterns. Sci. Comput. Program. **83**, 3–34 (2014). https://doi.org/10.1016/j.scico.2013.07.004, https://doi.org/10.1016/j.scico.2013.07.004
19. Meseguer, J.: Building correct-by-construction systems with formal patterns. In: Madeira, A., Martins, M.A. (eds.) Recent Trends in Algebraic Development Techniques - 26th IFIP WG 1.3 International Workshop, WADT 2022, Aveiro, Portugal, June 28-30, 2022, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13710, pp. 3–24. Springer (2022)
20. Meseguer, J.: Capturing system designs with formal executable specifications. In: Boronat, A., Fraser, G. (eds.) Fundamental Approaches to Software Engineering - 28th International Conference, FASE 2025. Lecture Notes in Computer Science, vol. 15693, pp. 1–32. Springer (2025)
21. Nigam, V., Talcott, C.: Automated construction of security integrity wrappers for industry 4.0 applications. Journal of Logical and Algebraic Methods in Programming **126**, 100745 (2022)
22. Ren, T., Williams, R., Ganguly, S., De Carli, L., Lu, L.: Breaking embedded software homogeneity with protocol mutations. In: Security and Privacy in Communication Networks: 18th EAI International Conference, SecureComm 2022, Virtual Event, October 2022, Proceedings. pp. 770–790. Springer (2023)
23. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: Qmaude: quantitative specification and verification in rewriting logic. In: International Symposium on Formal Methods. pp. 240–259. Springer (2023)
24. Sjoholmsierchio, M.: Software-Defined Networks: Protocol Dialects. Master's thesis, Naval Postgraduate School, Monterey, California, USA (Dec 2019), http://hdl.handle.net/10945/64066
25. Sun, M., Meseguer, J., Sha, L.: A formal pattern architecture for safe medical systems. In: International Workshop on Rewriting Logic and its Applications. pp. 157–173. Springer (2010)

26. Talcott, C.: Dialects for coap-like messaging protocols. arXiv preprint arXiv:2405.13295 (2024)

Bounded Structural Model Finding with Symbolic Data Constraints

Artur Boronat[1]*

University of Leicester, Leicester, UK
artur.boronat@leicester.ac.uk

Abstract. Bounded model finding is a key technique for validating software designs, usually obtained by translating high-level specifications into SAT/SMT problems. Although effective, such translations introduce a semantic gap and a dependency on external tools. We present the *Maude Model Finder* (MMF), a native approach that brings bounded model finding to the Maude rewriting logic framework. MMF provides a schema-parametric engine that repurposes symbolic reachability for structural solving, generating finite object configurations from class declarations and graph and data constraints without bespoke generators. Technically, MMF explores a constrained symbolic rewriting system over runtime states modulo an equational theory; SMT is used only to prune states by constraint satisfiability and to discharge entailment checks for subsumption and folding. We contribute a bounded, obligation-driven calculus that separates object creation from reference assignment and supports symmetry reduction by folding via Maude's ACU matching. We establish termination, soundness, and completeness of the bounded construction within the declared bounds, and justify folding via a coverage-preserving subsumption test. We focus on the calculus and its properties, illustrating it on running examples supported by a Maude prototype.

Keywords: Rewriting Logic · Maude · Bounded Model Finding · Symbolic Rewriting · Constrained Configurations · SMT

1 Introduction

Maude is a high-performance specification and analysis environment based on rewriting logic and membership equational logic [6, 7]. Its mainstream analysis capabilities target *dynamic* properties, including reachability search and LTL model checking for finite-state systems [2], and symbolic analysis for richer state spaces, including symbolic reachability and rewriting modulo SMT [3, 16]. These strengths make Maude an attractive semantic platform, but they do not directly support a complementary task that is central in modelling language engineering: bounded (structural) model finding.

* This work was supported by the European Space Agency under ESA Contract 4000133105/20/NL/AF (P-STEP project).

Bounded model finding aims to generate finite object graphs within declared bounds that satisfy (i) class and role[1] multiplicities, and (ii) structural and data constraints. Such bounded instances are essential to debug metamodels and Object Constraint Language (OCL) constraints, validate design assumptions, and obtain concrete witnesses and counterexamples. Tools such as Alloy and the USE validator provide this capability through SAT/SMT-based instance generation [11, 13]. In Maude, while one can execute specifications and explore behaviours, there is no out-of-the-box, schema-parametric mechanism dedicated to bounded *structural* instance generation for object configurations. In our earlier work on model subtyping with OCL constraints and safe reuse, bounded witnesses were obtained via the USE validator, using its Kodkod-based back end to check satisfiability of OCL constraints [4, 5]. This paper internalises bounded witness and counterexample generation within Maude.

We present the *Maude Model Finder* (MMF), a schema-parametric bounded model finding approach for Maude object configurations. MMF formulates bounded model construction as exploration of a constrained symbolic rewriting system over *runtime states* modulo oriented equations E and algebraic axioms B (such as associativity, commutativity, and identity). SMT is used only as a background solver to prune states by constraint satisfiability and to discharge entailment checks needed for subsumption and folding [3]. Concretely, we contribute: (1) a bounded, obligation-driven construction calculus that separates object allocation from reference assignment and incrementally accumulates symbolic data constraints; (2) redundancy reduction via folding, combining matching modulo ACU, shape indexing, and SMT-backed entailment; and (3) meta-theoretic guarantees for the bounded calculus, including termination, soundness, and completeness within the declared bounds, and soundness of folding with respect to semantic coverage. We focus on the calculus and its properties, illustrating it on running examples supported by a Maude prototype.

The paper is organised as follows. Section 2 presents the MMF specification style through a running example. Sections 3 and 4 introduce the semantic foundations and the symbolic construction calculus. Section 5 describes the reachability engine and its reduction pipeline, and Section 6 summarises the correctness results; full statements and proofs are available in the extended arXiv version of this paper. Section 7 reports a proof-of-concept evaluation, followed by related work and conclusions.

2 Specification with the Maude Model Finder (MMF)

We illustrate MMF on a small organizational schema with classes, references, bounds, and symbolic data constraints. An MMF specification describes a bounded family of admissible object graphs in three parts: (i) a *schema* declaring classes and roles, (ii) a *scope* fixing finite class and role bounds, and (iii) *constraints*, split into structural predicates over graphs and data invariants.

[1] A *role* is a typed reference between classes with a declared multiplicity range.

MMF generates models by *constrained symbolic rewriting* over *runtime states* $\langle C \,\|\, \phi \rangle$, where C is an object-graph configuration (matched modulo ACU axioms) and ϕ is an accumulated Boolean constraint over symbolic attribute variables. Exploration proceeds in two phases: an object allocation phase creates objects to meet class bounds, followed by a reference assignment phase that enumerates admissible role values within role bounds. Branches are pruned as soon as either (i) a monotonic structural error pattern is detected in the partial graph, or (ii) the constraint becomes SMT-unsatisfiable; non-monotonic global constraints are checked only on completed candidates (normal forms).

Schema and bounds. The running example has companies, employees, and projects: a `Company` has a mandatory CEO and a set of projects; an `Employee` may have a manager; a `Project` has a non-empty set of members. The schema is written in Maude's object-oriented style (Listing 1.1); role multiplicities are reflected by carrier sorts (e.g., `Oid`, `Maybe{Oid}`, `Set{Oid}`, `NeSet{Oid}`), while the effective branching space is fixed by explicit bounds. Listing 1.2 declares a finite scope via a constant `boundsDecl`: `classB` fixes class cardinalities and `roleB` constrains, per role, the admissible target class and multiplicity range.

Listing 1.1. Schema for the CEO example.

```
omod ORG-PROJ-SCHEMA is
  protecting MMF .
  class Company  | ceo : Oid , projects : Set{Oid} .
  class Project  | members : NeSet{Oid} .
  class Employee | manager : Maybe{Oid} .
endom
```

Listing 1.2. Example bounds for the CEO running example.

```
eq boundsDecl =
  classB(Company,  1, 1) ;
  classB(Employee, 2, 2) ;
  classB(Project,  0, 2) ;
  roleB(Company,  ceo,      Employee, 1, 1) ;
  roleB(Company,  projects, Project,  0, 2) ;
  roleB(Employee, manager,  Employee, 0, 1) ;
  roleB(Project,  members,  Employee, 1, 2) .
```

Structural constraints. MMF supports two structural error predicates. First, `specErrorPredGraphPartial` is evaluated on *partial* graphs to enable early pruning. It includes intrinsic well-formedness checks (e.g., unique object identifiers, no dangling edges, and adherence to upper bounds) and can be extended with user-defined monotonic error patterns; once triggered, such errors cannot be repaired by further refinement. Second, `specErrorPredGraph` is evaluated only on completed candidates (normal forms), to express non-monotonic constraints that need the full graph. In this example, we forbid a manager for the CEO and require the management relation to be acyclic (Listing 1.3). Both predicates return a value of sort `TruthPred`, whose sole constructor is `tt`. `TruthPred` is a dedicated sort distinct from Maude's `Bool`: error predicates are structurally matched

against `tt` to detect violations, whereas `Bool` is reserved for SMT constraint terms that are passed to the solver. Keeping the two sorts separate avoids ambiguity when Boolean SMT formulas are manipulated as first-class terms alongside structural graph checks.

Listing 1.3. Global constraints (cycle detection helper omitted).

```
op ceoHasManager : Configuration ~> TruthPred .
var C E1 : Oid .  var E2 : Maybe{Oid} .
var CAS AS : AttributeSet .  var CF : Configuration .
eq ceoHasManager(< C : Company | ceo : E1, CAS >
                 < E1 : Employee | manager : E2, AS > CF) = tt .
eq specErrorPredGraph(CF) =
  ceoHasManager(CF) ttor managerCyclic(CF) .
  --- ttor: short-circuit disjunction (tt-or) over TruthPred; managerCyclic: DFS
     ↪ cycle check (omitted)
```

Data invariants via constraints. Data-level properties are expressed by equational hooks[2] that contribute to the constraint ϕ. The hook `specInvOnCreate` is applied when an object is created; `specInvOnSetRef` is applied when a role value is committed. In the example, each employee is given a symbolic integer `level` constrained to a finite range; managers must have strictly higher rank (smaller `level`), and the CEO must be at level 0 (Listing 1.4). Here `i(O,AN)`, `b(O,AN)`, and `r(O,AN)` denote the symbolic SMT variables for the integer-, Boolean-, and real-valued attribute `AN` of object `O`, respectively (formalised in Section 4).

Listing 1.4. Equational constraints for the CEO example.

```
var E C : Oid .  var M : Maybe{Oid} .
eq specInvOnCreate(Employee, E) =
  i(E, level) >= (0).Integer and i(E, level) < (3).Integer .
ceq specInvOnSetRef(Employee, manager, E, M) =
  i(M, level) < i(E, level)
if M :: Oid .
eq specInvOnSetRef(Employee, manager, E, empty) = (true).Boolean .
eq specInvOnSetRef(Company, ceo, C, E) =
  i(E, level) === (0).Integer .
```

Execution modes. An MMF specification consists of the schema, `boundsDecl`, and the structural and data constraints above. The `find` mode enumerates valid models with `findAll`, while `findFirst` searches for a single witness:

```
reduce in MF-DRIVER-META : findAll(initModel(find)) .
```

```
reduce in MF-DRIVER-META : findFirst(initModel(find)) .
```

If `findFirst` yields no solution (or `findAll` an empty set), the specification is unsatisfiable within the declared bounds.

[2] We use *equational hook* to mean a Maude equation, named by convention, that fires on a specific construction event and returns a Boolean constraint contribution. This is distinct from Maude's built-in hook mechanism for foreign function interfaces.

Counterexample search (`check`). The `check` mode reuses the same bounded construction semantics and pruning, but it searches for a counterexample instead of valid models. A counterexample is an admissible normal form that violates a user-declared mixed property combining a graph pattern and attribute constraints. Graph patterns are executable predicates over the graph component and, when monotonic, can be evaluated on partial graphs to prune early. Attribute violations are searched by conjoining the negation of a constraint into ϕ; a satisfiable final constraint witnesses a violating valuation. We illustrate `check` with two small properties for the CEO example. The first property is structural and detects a non-CEO employee without a manager (Listing 1.5).

Listing 1.5. Graph property for `check`: non-CEO employee without manager.

```
op nonCeoWithoutManager : Configuration ~> TruthPred .
ceq nonCeoWithoutManager(< C : Company  | ceo : CEO, CAS >
     < E : Employee | manager : empty, AS > CF) = tt if E =/= CEO .
eq errorPredPropGraph(CF) = nonCeoWithoutManager(CF) .
```

Semantically, this graph property corresponds to an invariant $\mathsf{Inv}^{\mathsf{prop}}_{\mathsf{graph}}$. Operationally, `check` uses `errorPredPropGraph` as an executable error predicate, which returns `tt` precisely when $\mathsf{Inv}^{\mathsf{prop}}_{\mathsf{graph}}$ is violated.

The second is an attribute property that negates the intended level range and searches for a satisfying violation (Listing 1.6).

Listing 1.6. Attribute-level property for `check`: level range (searched as a violation).

```
op propInvOnCreate : ClassId Oid -> Boolean .
eq propInvOnCreate(Employee, E) =
  i(E, level) >= (0).Integer and i(E, level) < (3).Integer .
```

3 Semantic Foundations

We separate the *declarative meaning* of a specification from its *operationalization* by the MMF calculus.

Definition 1 (Core model specification). *A* core model specification $\mathcal{M} = (\mathcal{S}, \mathsf{Inv}^{\mathsf{part}}_{\mathsf{graph}}, \mathsf{Inv}^{\mathsf{full}}_{\mathsf{graph}}, \mathsf{Inv}_{\mathsf{attr}})$ *defines the validity criteria for constrained model instances. Here* $\mathcal{S}$ *is a Maude schema module declaring classes, attributes, and subclass relations;* $\mathsf{Inv}^{\mathsf{part}}_{\mathsf{graph}}$ *and* $\mathsf{Inv}^{\mathsf{full}}_{\mathsf{graph}}$ *are graph invariants for partial and completed graphs, respectively, with* $\mathsf{Inv}^{\mathsf{part}}_{\mathsf{graph}}$ *required to be monotonic; and* $\mathsf{Inv}_{\mathsf{attr}}$ *is the attribute invariant.*

We view $\mathsf{Inv}^{\mathsf{part}}_{\mathsf{graph}}$ as comprising an *intrinsic well-formedness* condition together with schema-provided monotonic constraints. Concretely, the intrinsic component requires that (i) object identifiers are unique, (ii) for every reference role, the current reference set does not exceed the declared upper multiplicity, and (iii) the given bounds are consistent, in the sense that every target class has an upper scope large enough to satisfy the maximum lower multiplicity demanded by any incoming role.

Let G range over Maude object configurations—multisets of object records $\langle o : C \mid \mathrm{AS} \rangle$ under multiset union modulo ACU. We write $G : \mathcal{S}$ to denote that G *conforms* to the schema, meaning that every object and reference in G respects the declarations and typing in $\mathcal{S}$.

Remark 1 (Invariants vs. error predicates). At the semantic level, $\mathsf{Inv}^{\mathsf{part}}_{\mathsf{graph}}$ and $\mathsf{Inv}^{\mathsf{full}}_{\mathsf{graph}}$ are invariants. Operationally, MMF's *error predicates* return tt when an invariant is *violated*. For partial graphs, this check combines built-in and schema-provided monotonic constraints. Thus, pruning checks the *negation* of semantic invariants.

Definition 2 (Bounded model specification). *A* bounded model specification $\mathcal{M}_b = (\mathcal{M}, b)$ *consists of a core specification* $\mathcal{M}$ *together with a scope* b*. The scope assigns to each class* C *a finite cardinality interval* $[\ell_C, u_C]$ *and to each role* $r : C \to D$ *a finite multiplicity interval* $[\ell_r, u_r]$*, bounding the number of targets that each object of class* C *may have along* r*.*

We write $G : \mathcal{S}_b$ iff $G : \mathcal{S}$ and all bounds in b hold in G: each class C occurs in G with cardinality in $[\ell_C, u_C]$, and for each object $o : C$ the set of target object identifiers assigned to role r of o, denoted $o.r$, has cardinality in $[\ell_r, u_r]$.

Definition 3 (Constrained model instance). *A* constrained model instance *is a pair* $M = \langle G, \rho \rangle$ *where* G *is a* graph topology *(a finite set of typed object records with role assignments, i.e., an object configuration* $G : \mathcal{S}$*) and* ρ *is a ground valuation for the attribute-level symbols occurring in* G *and in the attribute invariant* $\mathsf{Inv}_{\mathsf{attr}}(G)$*.*

Definition 4 (Conformance). *Let* $\mathcal{M}_b$ *be a bounded specification. A constrained model instance* $M = \langle G, \rho \rangle$ conforms *to* $\mathcal{M}_b$*, written* $M : \mathcal{M}_b$*, iff:*

$$G : \mathcal{S}_b \ \wedge \ (G \models \mathsf{Inv}^{\mathsf{part}}_{\mathsf{graph}}) \ \wedge \ (G \models \mathsf{Inv}^{\mathsf{full}}_{\mathsf{graph}}) \ \wedge \ (\rho \models \mathsf{Inv}_{\mathsf{attr}}(G)).$$

Definition 5 (Bounded semantics). *Define* $[\![\mathcal{M}_b]\!] = \{ M \mid M : \mathcal{M}_b \}$.

4 Symbolic State and Transition Semantics

We now turn to the formal machinery enabling automatic model finding, the *MMF calculus*. Unlike operational semantics that evolves a system over time, MMF employs a *constructive* symbolic transition system where configurations denote partial graphs and transitions represent refinement decisions. This design incrementally builds the graph, explicitly separating *existence* from *topology*.

This section formalizes the symbolic transition semantics of MMF. We first define runtime states in Section 4.1, which encapsulate the partial graph and symbolic constraints. We then detail the two-phase construction process: the object generation phase in Section 4.3, which allocates objects and their data attributes, and the graph creation phase in Section 4.4, which enumerates reference topologies over the fixed object universe.

4.1 States in the MMF Runtime

The MMF calculus operationalises the bounded satisfiability problem for a bounded model specification $\mathcal{M}_b$ as a finite reachability search over runtime states. Let `Configuration` be the Maude configuration sort, i.e., multisets of objects and control tokens under multiset union modulo ACU. We write C, C' : `Configuration`. We write $\mathsf{graph}(G)$ for the object-graph fragment (a multiset of object records), and ϕ for Boolean constraints over the background SMT theories. Class and role identifiers are K and R; object identifiers O are drawn from a finite pool bounded by b.

Definition 6 (Runtime states and graph projection). *A* runtime state *is a constrained configuration $S \equiv \langle\, C \parallel \phi \,\rangle$, where C is a multiset of objects including* $\mathsf{graph}(G)$ *and control tokens modulo ACU, and ϕ is a Boolean constraint. Let $\pi(C)$ be the* graph projection *that erases all runtime tokens from C and keeps only the object records.*

Semantically, S denotes a set of constrained model instances $M = \langle \pi(C), \rho \rangle$ such that $\rho \models \phi$; its denotation is given in Definition 11. Object identifiers are treated as fixed uninterpreted symbols during construction; only attribute-level symbols are assigned values by the constraint valuations. A configuration C packages both the evolving graph and bounded construction control. Its core tokens include a *phase marker* $\mathsf{phase}(q)$ with $q \in \{\mathsf{objectBuild}, \mathsf{refBuild}\}$; an *identifier universe* $\mathsf{allIds}(OS)$ indexed by class; an *allocation cursor* $\mathsf{freshIds}(OL)$ and the set of allocated identifiers $\mathsf{takenIds}(TI)$; *lower-bound obligations* $\mathsf{missing}(K, n)$ and *feasibility obligations* $\mathsf{need}(K, n)$; and a *reference-task pool* $\mathsf{refPool}(RTL)$ discharged in phase $\mathsf{refBuild}$. These tokens are operational devices: they ensure boundedness, phase discipline, and exhaustive enumeration.

Remark 2 (Role of runtime tokens). Runtime tokens (identifier pools, obligation counters, task lists, phase markers) are part of C because they drive the operationalization, support termination, and enable completeness by systematic enumeration. They do *not* appear in the semantic satisfaction relation $\models$; the link is mediated by the projection π and the state denotation $[\![\cdot]\!]$.

4.2 Overview of the MMF Construction Calculus

The MMF calculus is *two-phase*: in phase $\mathsf{objectBuild}$ it allocates a bounded set of objects, and in phase $\mathsf{refBuild}$ it assigns reference-valued attributes. Let B be the structural axioms involved in the constructors of $\mathcal{M}$ and runtime tokens. Each one-step move is a rule application on the runtime configuration C modulo B, followed by equational normalization by the oriented equations $\boldsymbol{E}$ modulo B.

We distinguish three layers of rewriting: a *raw* rule step operates on the configuration C (modifying the graph topology G); a *normalized* step applies equational simplification; and a *constrained* step lifts the normalized structural transition to the level of runtime states by accumulating the attribute constraints implied by the structural change.

Definition 7 (Raw rewriting step). *We write* $C \to^{\ell,\sigma}_{R/B} C_1$ *iff there exists a rule in* R *with label* ℓ *and a substitution* σ *(obtained by matching modulo* B*) such that applying the instantiated rule* $\ell\sigma$ *rewrites* C *to* C_1 *modulo* B*.*

Definition 8 (Normalized rewriting step). *We write* $C \Rightarrow^{\ell,\sigma}_{R,\boldsymbol{E}/B} C'$ *iff there exists an intermediate* C_1 *such that* $C \to^{\ell,\sigma}_{R/B} C_1$ *and* $C' = C_1 \downarrow_{\boldsymbol{E}/B}$*, where* $C_1 \downarrow_{\boldsymbol{E}/B}$ *denotes the canonical form of* C_1 *with respect to the oriented equations* $\boldsymbol{E}$ *modulo axioms* B*.*

Remark 3 (Maude implementation of rewriting steps). The raw step $C \to^{\ell,\sigma}_{R/B} C_1$ is realised via Maude's reflective `metaApply`, which applies a single rewrite rule and returns the result modulo B. Normalization $\downarrow_{\boldsymbol{E}/B}$ corresponds to `metaReduce`, which computes the $\boldsymbol{E}/B$-canonical form. These are the standard reflective operations used throughout the MMF engine (Section 5).

Only two rule families add user attribute constraints to the guard: object creation (Obj-Gen) and reference commitment (Ref-Choose). We factor these event-specific contributions through two hook shorthands:

$$\mathsf{hook}_{\mathsf{create}}(K, o) \triangleq \mathsf{specInvOnCreate}(K, o)$$
$$\mathsf{hook}_{\mathsf{setref}}(K, R, o, v) \triangleq \mathsf{specInvOnSetRef}(K, R, o, v)$$

and define the auxiliary function $\mathsf{hook}_\ell(\sigma)$ to capture the rule-indexed injection logic:

$$\mathsf{hook}_\ell(\sigma) \triangleq \begin{cases} \mathsf{hook}_{\mathsf{create}}(K\sigma, O\sigma) & \text{if } \ell = \text{Obj-Gen} \\ \mathsf{hook}_{\mathsf{setref}}(K\sigma, R\sigma, O\sigma, v\sigma) & \text{if } \ell = \text{Ref-Choose} \\ \mathsf{true} & \text{otherwise} \end{cases}$$

Definition 9 (Constrained rewriting step). *We write*

$$\langle\, C \parallel \phi \,\rangle \Rightarrow^{\ell,\sigma}_{R,\boldsymbol{E}/B} \langle\, C' \parallel \phi' \,\rangle \qquad \textit{if} \qquad C \Rightarrow^{\ell,\sigma}_{R,\boldsymbol{E}/B} C',$$

and the constraint is strengthened by the corresponding hook and normalized: $\phi' = \mathsf{simpB}\big(\phi \wedge \mathsf{hook}_\ell(\sigma)\big) \downarrow_{\boldsymbol{E}/B}$[3]*.*

Remark 4 (Contextual conventions). Unless stated otherwise, all runtime configurations and constraints are maintained in $\boldsymbol{E}/B$-canonical form. In the rule presentations below, we write $\langle\, C \parallel \phi \,\rangle \Rightarrow \langle\, C' \parallel \phi' \,\rangle$ as shorthand for the constrained rewriting step $\langle\, C \parallel \phi \,\rangle \Rightarrow^{\ell,\sigma}_{R,\boldsymbol{E}/B} \langle\, C' \parallel \phi' \,\rangle$; the rule label ℓ, matching substitution σ, and theory parameters $R, \boldsymbol{E}, B$ are implicit from context. We write $S \Rightarrow S'$ when $S \Rightarrow^{\ell,\sigma}_{R,\boldsymbol{E}/B} S'$ holds for some ℓ and σ (Definition 9); in `check` mode, the step additionally updates the witness component ψ_P as described below.

[3] The simpB function implements a deterministic simplification strategy that normalizes Boolean constraints by flattening associative-commutative operators, ordering literals, pushing negations into relational atoms, performing arithmetic constant folding, and eliminating identity elements.

Let $\mathsf{init}(\mathcal{M}_b, \mathsf{mode})$ be the canonical initial runtime state. The configuration C_0 contains $\mathsf{phase}(\mathsf{objectBuild})$, an identifier universe $\mathsf{allIds}(OS)$ and allocation list $\mathsf{freshIds}(OL)$, where OL is a fixed linearization of the identifiers in OS induced by the class order $\mathsf{classList}$ and the per-class upper bounds in b. Its head determines the next identifier to be consumed and hence the canonical allocation order. It also contains the initial bounded-work obligations: for every class C in the schema, C_0 includes a token $\mathsf{missing}(C, \ell_C)$, where ℓ_C is the class lower bound from b, and a token $\mathsf{need}(C, \mathrm{inNeed}(C))$, where $\mathrm{inNeed}(C)$ is the maximum lower multiplicity among roles targeting C. Finally, $\phi_0 = \mathsf{true}$. If $\mathsf{mode} = \texttt{find}$, then $\mathsf{init}(\mathcal{M}_b, \texttt{find}) = \langle\, C_0 \,\|\, \phi_0 \,\rangle$. If $\mathsf{mode} = \texttt{check}$, then $\mathsf{init}(\mathcal{M}_b, \mathsf{check}) = \langle\, C_0 \,\|\, \phi_0 \,\|\, \psi_0 \,\rangle$ with $\psi_0 = \mathsf{false}$. In check mode, ϕ accumulates hard constraints, as usual, but ψ_P accumulates an existential violation witness.

Property instrumentation (`check`). In `check` mode, the calculus is additionally parameterised by a mixed property with a graph component and an attribute component. Semantically, the graph component is an invariant $\mathsf{Inv}^{\mathsf{prop}}_{\mathsf{graph}}$ evaluated on projected graphs. Operationally, the implementation provides an error predicate $\mathsf{errorPredPropGraph}$ such that $\mathsf{errorPredPropGraph}(G) = \mathsf{tt}$ iff $G \not\models \mathsf{Inv}^{\mathsf{prop}}_{\mathsf{graph}}$. The attribute component is specified by hook equations $\mathsf{propInvOnCreate}$ and $\mathsf{propInvOnSetRef}$ (Section 2). Operationally, `check` reuses the same construction rules and the same hard-constraint accumulation. In addition, it records whether some property violation can be witnessed by maintaining a separate Boolean witness component ψ_P that accumulates violation evidence disjunctively across partial states. Formally, for a rule instance with label ℓ and matching substitution σ, define

$$\mathsf{propViol}_\ell(\sigma) \triangleq \begin{cases} \neg\, \mathsf{propInvOnCreate}(K\sigma, O\sigma) & \text{if } \ell = \textsc{Obj-Gen} \\ \neg\, \mathsf{propInvOnSetRef}(K\sigma, R\sigma, O\sigma, v\sigma) & \text{if } \ell = \textsc{Ref-Choose} \\ \mathsf{false} & \text{otherwise.} \end{cases}$$

In `check` mode, the hard constraint is updated as usual. The witness is updated by disjunction: $\psi'_P = \mathsf{simpB}\big(\psi_P \vee \mathsf{propViol}_\ell(\sigma)\big)$. Accordingly, in `check` mode we write

$$\langle\, C \,\|\, \phi \,\|\, \psi_P \,\rangle \Rightarrow^{\ell,\sigma}_{R,\boldsymbol{E}/B} \langle\, C' \,\|\, \phi' \,\|\, \psi'_P \,\rangle$$

iff $C \Rightarrow^{\ell,\sigma}_{R,\boldsymbol{E}/B} C'$ and the above two updates hold.

The calculus runs in two modes: `find` enumerates valid models satisfying all structural and data invariants, while `check` searches for a counterexample to a user-declared property. Normal-form acceptance is defined accordingly.

Definition 10 (Normal forms, admissibility, acceptance). *Normal form* $\mathsf{NF}(S)$ *and* $\mathsf{Admissible}(S)$ *are defined as follows:*

- $\mathsf{NF}(S)$ *holds iff no calculus rule applies to* S*, i.e., iff* $\neg\exists S'.\ S \Rightarrow S'$.
- $\mathsf{Admissible}(S)$ *holds iff* $\pi(C) \models \mathsf{Inv}^{\mathsf{part}}_{\mathsf{graph}}$ *and* $sat(\phi)$.

- $\mathsf{Accept}_{\texttt{find}}(S)$ *holds iff* $\mathsf{NF}(S) \wedge \mathsf{Admissible}(S) \wedge \big(\pi(C) \models \mathsf{Inv}^{\mathsf{full}}_{\mathsf{graph}}\big)$.
- $\mathsf{Accept}_{\texttt{check}}(S)$ *holds iff* $\mathsf{Accept}_{\texttt{find}}(S)$ *and, additionally,*

$$\pi(C) \not\models \mathsf{Inv}^{\mathsf{prop}}_{\mathsf{graph}} \ \vee \ sat(\phi \wedge \psi_P).$$

Remark 5 (Counterexamples in check *mode).* In check mode, hard constraints are accumulated in ϕ and define the denotation $[\![S]\!]$. Property violations are accumulated in the witness component ψ_P by disjunction. Therefore, if $sat(\phi \wedge \psi_P)$ holds, there exists a valuation ρ such that $\rho \models \phi$ and $\rho \models \psi_P$. Then $\langle \pi(C), \rho \rangle \in [\![S]\!]$. Moreover, under ρ the attribute-level property is violated at least once. Graph-level violations are witnessed independently by $\pi(C) \not\models \mathsf{Inv}^{\mathsf{prop}}_{\mathsf{graph}}$.

Definition 11 (State semantics). *Let ρ range over valuations that assign* ground *values to all attribute-level symbols occurring in ϕ (e.g., symbols of the form $i(O, AN)$, $r(O, AN)$, $b(O, AN)$ for an object identifier O and attribute name AN), while leaving the object identifiers occurring in C unchanged. Define the* semantics *of a runtime state by*

$$[\![\langle\, C \parallel \phi \,\rangle]\!] \ = \ \{\, \langle \pi(C), \rho \rangle \mid \rho \models \phi \,\} \ = \ [\![\langle\, C \parallel \phi \parallel \psi_P \,\rangle]\!].$$

Intuitively, $[\![\langle\, C \parallel \phi \,\rangle]\!]$ is the set of constrained model instances $M = \langle \pi(C), \rho \rangle$ such that ρ satisfies the accumulated constraint ϕ. The projection $\pi(C)$ provides the graph topology and the object identifiers, which are treated as fixed uninterpreted symbols during construction. The valuation ρ supplies ground values for the attribute-level symbols occurring in ϕ. In check mode, the witness component ψ_P affects acceptance but not denotation.

4.3 Object Generation Phase (objectBuild)

Phase objectBuild allocates fresh objects and initializes reference-assignment obligations. We assume mode find and a configuration containing phase(objectBuild). The transition relation $\Rightarrow$ is given by constrained rewriting rules over runtime states. Rule Obj-Gen matches the configuration modulo ACU to extract the next identifier O_K from $\mathsf{freshIds}(O_K :: OL)$ (thus fixing a canonical allocation order). It consumes O_K, creates a fresh object, and decrements the class-obligation counters $\mathsf{missing}(K, N)$ and $\mathsf{need}(K, L)$. Here $\ominus$ denotes saturating subtraction, i.e., $n \ominus 1 = \max(n - 1, 0)$.

Object creation performs (i) *structural initialization* and (ii) *scheduling of later reference choices*: $\mathsf{initAttrs}(K, O_K)$ populates attributes with well-typed placeholders (e.g., $\emptyset$ for sets and a distinguished value for single references), while addTasks enqueues one task $\mathsf{refIdx}(O_K, R, K', 0, REM)$ per role $R : K \to K'$, where 0 is the initial rank and REM is a remaining-rank budget that represents the (finite) bounds-induced choice space. The constraint is strengthened with the user hook $\mathsf{specInvOnCreate}(K, O_K)$ and canonicalized by simpB.

$$\textsc{Obj-Gen}\quad \langle \begin{array}{l} \mathsf{graph}(G),\\ \mathsf{freshIds}(O_K :: OL),\\ \mathsf{takenIds}(TI),\\ \mathsf{missing}(K,N),\\ \mathsf{need}(K,L),\\ \mathsf{refPool}(RTL)\ldots \end{array} \| \phi \rangle \;\Rightarrow\; \langle \begin{array}{l} \mathsf{graph}(G,\ new),\\ \mathsf{freshIds}(OL),\\ \mathsf{takenIds}(O_K, TI),\\ \mathsf{missing}(K, N \ominus 1),\\ \mathsf{need}(K, L \ominus 1),\\ \mathsf{refPool}(RTL')\ldots \end{array} \| \phi' \rangle$$

where

$$\begin{aligned} \phi' &= \mathsf{simpB}(\phi \wedge \mathsf{hook}_{\mathsf{create}}(K, O_K)),\\ RTL' &= \mathsf{addTasks}(O_K, K, \mathsf{roleList}(K), \mathsf{allIds}(OS), RTL),\\ new &= \langle O_K : K \mid \mathsf{initAttrs}(K, O_K, \mathsf{allIds}(OS))\rangle. \end{aligned}$$

Rule Obj-Skip allows exploring graphs smaller than the maximum scope. It applies only when lower-bound obligations are met ($\mathsf{missing}(K,0)$ and $\mathsf{need}(K,0)$):

$$\textsc{Obj-Skip}\quad \langle \begin{array}{l} \mathsf{freshIds}(O_K :: OL),\\ \mathsf{missing}(K,0),\\ \mathsf{need}(K,0)\ldots \end{array} \| \phi \rangle \;\Rightarrow\; \langle \begin{array}{l} \mathsf{freshIds}(OL),\\ \mathsf{missing}(K,0),\\ \mathsf{need}(K,0)\ldots \end{array} \| \phi \rangle$$

where the constraint is unchanged and the allocation cursor advances by dropping O_K.

Remark 6 (Partial states and global invariants). Intermediate states during objectBuild are *partial*: they do not necessarily satisfy non-monotonic global invariants $\mathsf{Inv}^{\mathsf{full}}_{\mathsf{graph}}$. Only monotonic constraints (captured by specErrorPredGraphPartial) are checked during construction. Full invariants are verified only at acceptance.

The objectBuild phase consists of the repeated application of these symbolic transition rules until $\mathsf{freshIds}(\mathsf{nil})$ is exhausted. This exhaustion triggers the transition to the refBuild phase. Crucially, as the calculus switches phases, it restricts the global $\mathsf{allIds}(OS)$ component to exactly the set of identifiers present in takenIds. This filtering step ensures that the subsequent reference generation phase considers only the objects that were actually allocated, preventing the creation of dangling edges to skipped or non-existent objects by construction.

4.4 Graph Creation Phase (refBuild)

In the refBuild phase, reference tasks are discharged. A task $RT = \mathsf{refIdx}(O_K, R, K', \mathsf{curr}, REM)$ stores a *rank cursor* curr together with a *remaining-rank budget* REM. It represents the interval of admissible ranks $[\mathsf{curr},\ \mathsf{curr} + REM]$ in the canonical enumeration of role values.

A key semantic choice in refBuild is that role values are not generated by incremental extension, but are instead selected as elements of a finite, *intrinsically defined* domain determined solely by the declared multiplicity bounds and

the identifier universe. Fix a reference task $\mathsf{refIdx}(O_K, R, K', \mathsf{curr}, REM)$ and let OS be the class-indexed identifier set associated with K' in $\mathsf{allIds}(OS)$. Write $P = [o_1, \ldots, o_n]$ for the canonical list obtained from OS under a fixed total order on identifiers, and let ℓ, u be the lower and upper multiplicity bounds for role R (with $0 \le \ell \le u \le n$). The set $\mathcal{D}_{\ell,u}(OS)$ of all admissible reference assignments is the union of all subsets of targets whose size falls within the declared bounds:

$$\mathcal{D}_{\ell,u}(OS) \;=\; \bigcup_{m=\ell}^{u} \{\, S \subseteq OS \mid |S| = m \,\}, \qquad U_{\max} \;=\; |\mathcal{D}_{\ell,u}(OS)| - 1.$$

Initially, a task for this role is created with $\mathsf{curr} = 0$ and $REM = U_{\max}$, so that the represented rank interval is exactly $[0, U_{\max}]$. MMF induces a canonical enumeration over this space without materializing it. The order is defined hierarchically: first by subset size m (from ℓ to u), and then, within each size, by lexicographic unranking induced by the list order P.

The core mechanism is the unranking function $\mathsf{unrank}(P, m, j)$, which acts as a stateless cursor. It reconstructs the j-th subset of size m directly from P using the standard combinatorial number system for ranking and unranking combinations (see, e.g., [12]). The intrinsic selector $\mathsf{choiceRef}$ is then defined by delegating to the ranked enumeration:

$$\mathsf{choiceRef}(OS, K, R, K', \mathsf{curr}) \;=\; \mathsf{rankedFromTo}(P, \ell, u, \mathsf{curr}).$$

Rule Ref-Choose triggers a symbolic transition that commits to the assignment defined by the current rank curr. It calculates the reference v using $\mathsf{choiceRef}$ and performs the semantic update to the object's attribute set:

Ref-Choose

$$\begin{array}{l} \mathsf{allIds}(OS) \\ \langle\, \mathsf{graph}(\langle O_K : K \mid \mathrm{AS}\rangle \ldots),\ \|\, \phi \,\rangle \\ \quad \mathsf{refPool}(RT :: RTL) \ldots \end{array} \;\Rightarrow\; \begin{array}{l} \mathsf{allIds}(OS) \\ \langle\, \mathsf{graph}(\langle O_K : K \mid \mathrm{AS}'\rangle \ldots),\ \|\, \phi' \,\rangle \\ \quad \mathsf{refPool}(RTL) \ldots \end{array}$$

where $RT = \mathsf{refIdx}(O_K, R, K', \mathsf{curr}, REM)$, $v = \mathsf{choiceRef}(OS, K, R, K', \mathsf{curr})$, $\mathrm{AS}' = \mathsf{setRef}(\mathrm{AS}, R, v)$, and $\phi' = \mathsf{simpB}(\phi \wedge \mathsf{hook}_{\mathsf{setref}}(K, R, O_K, v))$.

Rule Ref-Skip represents the decision to *discard* the current assignment and continue searching. It increments the cursor curr to explore the next candidate and decrements the remaining-rank budget REM, thereby shrinking the residual rank interval while preserving the global upper rank $\mathsf{curr} + REM$.

$$\text{Ref-Skip} \quad \langle\, \mathsf{refPool}(RT :: RTL) \ldots \|\, \phi \,\rangle \;\Rightarrow\; \langle\, \mathsf{refPool}(RT' :: RTL) \ldots \|\, \phi \,\rangle$$

where $RT = \mathsf{refIdx}(O_K, R, K', \mathsf{curr}, REM)$, $REM > 0$, and $RT' = \mathsf{refIdx}(O_K, R, K', \mathsf{curr}{+}1, REM{-}1)$. Repeated application of Ref-Skip advances through the bounds-induced domain of admissible values. When $REM = 0$, the only option is to choose (or fail if the domain is empty), ensuring termination. Completion of $\mathsf{refBuild}$ yields a normal form NF, checked for global acceptance.

Remark 7 (Finite enumeration). Each reference task has a finite choice space determined by b, so termination follows: Ref-Skip decreases the remaining interval width, and Ref-Choose removes the task from the pool.

5 Symbolic Reachability Engine

Section 4 defined the symbolic transition relation $\Rightarrow$ over runtime states $\langle C \,\|\, \phi \rangle$. We now define a meta-level exploration strategy that traverses $\Rightarrow$ under a fixed reduction pipeline comprising exact deduplication, admissibility pruning, and optional folding. The exploration engine maintains a loop state

$$L = (PQ,\ \mathit{SeenMap},\ \mathit{SeenSuccMap},\ \mathit{ExactMap},\ \mathit{Profiler}),$$

organising the exploration components. The priority queue PQ serves as a worklist of pending nodes, where each node packages a runtime state with an index key and a cached NF flag. Heuristic search and folding are facilitated by *SeenMap*, which stores retained representatives per index key for coverage-based pruning, and *SeenSuccMap*, which memoises exact successor states to suppress duplicates. Finally, *ExactMap* collects accepted normal-form states as they are found, while the *Profiler* tracks instrumentation counters.

5.1 Meta-level Exploration Semantics

The search procedure is parameterized by a configuration term `Opt`, which controls active reduction strategies (e.g., symmetry handling, folding). Starting from an initial state S_0, the loop initializes the state components L and iterates as long as the worklist is non-empty. In each step, the engine selects a pending node via Select and delegates it to processNode. The loop terminates when the frontier is exhausted or, in `findFirst` mode, as soon as a witness is recorded.

The processNode routine bifurcates based on the state's status. If S is a normal form ($\mathsf{NF}(S)$), it constitutes a candidate result; the engine validates it via the mode-specific acceptance predicate and, if successful, records it in *ExactMap*. If S is expandable, the engine computes its one-step symbolic successors using the expansion operator Step(S) (implemented via Maude's reflective `metaApply`). These raw successors then traverse the reduction pipeline: FilterExact eliminates duplicates against *SeenSuccMap*; Prune discards inadmissible states (violating $\mathsf{Inv}^{\mathsf{part}}_{\mathsf{graph}}$ or with an unsatisfiable constraint); and, if enabled by `Opt`, a folding stage enforces semantic coverage against *SeenMap*. In the current implementation, folding is applied only to successors that are already in normal form. Finally, surviving successors are added to the worklist via Insert.

5.2 Subsumption and Folding

This subsection formalizes the redundancy elimination mechanism used by the exploration loop. The key idea is to maintain, among explored normal forms, a

covering antichain with respect to a *semantic embedding preorder*. This preorder soundly approximates the semantic *coverage* relation and supports both queue-time pruning (FoldSift) and repository maintenance (FoldRefresh).

Recall from Definition 11 that a runtime state $S \equiv \langle\, C \,\|\, \phi \,\rangle$ denotes a set $[\![S]\!]$ of constrained model instances. Each constrained model instance is a pair $M = \langle \pi(C), \rho \rangle$ consisting of a graph topology and a satisfying valuation. We recall the *semantic coverage preorder* $\preceq$:

$$S_1 \preceq S_2 \quad \text{iff} \quad [\![S_2]\!] \subseteq [\![S_1]\!].$$

Intuitively, S_1 covers S_2 if every constrained model instance represented by S_2 is also represented by S_1. Coverage is a preorder, not an equivalence. The engine realizes the preorder $\preceq_{\text{MMF}}$ via $\texttt{subsumes}(S_1, S_2)$, using three criteria to determine coverage. First, an *index equality* gate checks $\eta(S_1) = \eta(S_2)$ to filter incompatible candidates. Second, *structural embedding* employs `metaMatch` to find a substitution σ such that $\pi(C_2) =_{E \cup B} \pi(C_1)\sigma$. Third, *constraint entailment* verifies via SMT that ϕ_2 entails $\phi_1\sigma$. Meeting these guarantees S_1 represents every constrained model instance of S_2, making `subsumes` a sound (albeit incomplete) semantic coverage test.

Folding maintains a covering antichain $\mathcal{A}$ of normal-form representatives during exploration. Thus, $\mathcal{A}$ contains no mutually subsuming representatives yet covers all explored normal forms. Operationally, FoldSift discards a candidate S only if $\mathsf{NF}(S)$ and some $R \in \mathcal{A}$ satisfies $\texttt{subsumes}(R, S)$. Conversely, FoldRefresh inserts new representatives by removing any $R \in \mathcal{A}$ subsumed by the incoming state. Hence, coverage is sound: every constrained model instance in the bounded semantics remains represented by at least one retained representative in the exploration graph. MMF does not compute graph isomorphism modulo arbitrary identifier permutations. Instead, symmetry reduction comes from representing configurations modulo the equational theory $(\Sigma, E \cup B)$ (including ACU for multiset composition) and from shape indexing. Within a bucket, structural embedding is decided by matching modulo $E \cup B$ via `metaMatch`, while data-level constraints in ϕ are discharged by SMT.

6 Correctness of the MMF Calculus

We summarise the main metatheoretic properties of the bounded construction calculus and its search engine. The bounded model specification $\mathcal{M}_b$ fixes a finite universe and role ranges, so all guarantees are relative to the declared bounds. Soundness ensures that any accepted normal form denotes only constrained model instances that conform to $\mathcal{M}_b$, so reported witnesses are never spurious. Completeness ensures that every constrained model instance admitted by $\mathcal{M}_b$ is represented by some retained accepted state, so folding and subsumption do not discard valid solutions. Together, these results show that the calculus explores the entire bounded search space modulo coverage-preserving reductions.

Termination and well-formedness explain why exploration is finite and why intermediate states remain meaningful. Runtime well-formedness $\mathsf{WF}(S)$ cap-

run	ms	rewrites	popped	NF	pruned	folded
`findAll(find)`	2181	7.92M	450	52	354	88
`findFirst(find)`	136	284.8K	29	1	27	0
`findAll(check)`	2409	11.11M	374	40	298	64
`findFirst(check)`	144	317.6K	29	1	21	0

Table 1. Mode comparison under the baseline pipeline. *pruned* = `prunedBad`+`prunedUnsat`.

tures intrinsic structural consistency of partial graphs and the correct accumulation of constraint obligations in ϕ, and each rule preserves it. Under standard assumptions on equational normalization and finite enumeration of reference tasks, the lexicographic measure on remaining identifiers and task widths strictly decreases. Consequently, the bounded search terminates and pruning by monotonic structural constraints and SMT unsatisfiability remains correct. full statements and proofs are available in the extended arXiv version of this paper.

7 Evaluation

We evaluate MMF on the CEO benchmark (Company–Employee–Project) from Section 2, which combines structural constraints (e.g., "CEO has no manager", acyclicity) with symbolic integer invariants. The goal is a proof-of-concept assessment of feasibility rather than an exhaustive performance study. Tables 1 and 2 use the bounds from Listing 1.2: 1 company, 2 employees, and 0–2 projects. Table 3 varies the employee bound from 2 to 5. Timeouts are set at 120 s. All runs use a deterministic worklist policy (`useHeuristicPQ=false`). We report wall-clock time (`ms`), Maude rewrites, and driver counters: expanded states (`popped`), normal forms (`NF`), successors generated/enqueued, and eliminations by exact deduplication, structural pruning (`prunedBad`), SMT unsatisfiability (`prunedUnsat`), and folding (`folded`). Unless stated otherwise, the baseline enables SMT pruning, folding, and shape indexing (`useSymbContraintFiltering=true`, `useFolding=true`, `useIndex=true`), and removes redundant solutions at insertion (`removeRedundantSolutions=true`).

Table 1 compares `findAll`/`findFirst` and `find`/`check` under the baseline pipeline. For `findAll`, more than half of generated successors are eliminated before queueing, with structural pruning being the largest contributor and SMT pruning and folding providing additional reductions. `findAll(check)` expands fewer states than `findAll(find)` but is slightly slower, indicating higher per-state overhead in `check` (property-side evaluation and constraint processing). For `findFirst`, folding does not contribute on this scope (`folded`=0); runtime is largely determined by pruning and successor generation.

Table 2 isolates the reduction mechanisms for `findAll(find)`. Disabling SMT pruning while keeping folding enabled times out, whereas disabling both SMT pruning and folding completes but is an order of magnitude slower than baseline; thus SMT satisfiability filtering is essential and also stabilizes folding

run	ms	rewrites	popped	NF	inserted	redundant
`A1-baseline`	2185	7.92M	450	52	52	0
`A2-noSMT`	timeout	-	-	-	-	-
`A3-noFolding`	1258	2.20M	494	96	52	44
`A4-folding-noIndex`	timeout	-	-	-	-	-
`A5-noSMT-noFolding`	20299	28.01M	2054	192	100	92

Table 2. Ablations for `findAll(find)`. Timeouts occur at 120 s.

employees	ms (base)	rewrites (base)	folded (base)	ms (noFold)	rewrites (noFold)
2	97	0.10M	0	88	0.05M
3	147	0.32M	0	107	0.05M
4	737	4.08M	3	172	0.13M
5	3098	20.27M	6	264	0.24M

Table 3. Scalability for `findFirst(check)` as the employee bound increases.

(which relies on entailment checks). Disabling folding decreases runtime but increases the number of normal forms; redundant solutions are then filtered only at insertion time, showing that folding prevents exploring many redundant candidates to completion. Folding without indexing times out, indicating that shape indexing is a prerequisite for making folding feasible.

Table 3 scales `findFirst(check)` from 2 to 5 employees, comparing baseline to `noFolding`. Baseline time increases sharply with the bound, while `noFolding` scales much more gently; at 5 employees, `noFolding` is over an order of magnitude faster. Since folding eliminates only a few successors at these bounds, this suggests that subsumption checks dominate cost in witness-oriented search when folding successes are rare. All `folding-noIndex` scalability runs timed out, consistent with the ablation results: indexing is necessary for folding to be viable.

Overall, structural pruning removes most successors in exhaustive search, while SMT pruning is essential for feasibility and stabilises folding. Folding reduces redundant candidates when shape indexing makes subsumption checks practical. In `check` mode, witness search incurs higher per-state overhead but remains practical on the explored bounds.

The evaluation focuses on one benchmark family; folding effectiveness may differ for specifications with stronger symmetries or weaker constraints. We report single-run timings; although runs are deterministic, wall time can vary with system load. Rewrite counts are a stable internal proxy for work but do not isolate SMT solver time. Timeout-based conclusions depend on the 120 s cutoff, but the observed blow-ups are consistent across the "noSMT+folding" and "folding-noIndex" configurations.

8 Related work

Bounded Model Finding. MMF shares the goal of bounded model finders like Alloy [11] and USE [10,13]: generating concrete object graphs that satisfy structural

constraints. Unlike these tools, which typically rely on translation to SAT/SMT solvers (often creating a semantic gap), MMF operates as a native symbolic rewriting engine within rewriting logic. This allows domain logic and constraints to be defined directly in Maude [7], providing a uniform environment for specification, analysis, and verification.

Symbolic Reachability and Symbolic Rewriting. Symbolic analysis in Maude ranges from narrowing-based reachability to symbolic rewriting modulo SMT [3, 14, 16]. MMF repurposes this symbolic infrastructure for bounded structural model finding. Technically, MMF explores a constrained symbolic rewriting system over runtime states that refine partial object graphs, rather than using narrowing to solve for substitutions that drive temporal reachability. The correctness of our calculus rests on established results in this domain. Our soundness result mirrors the simulation results of Bae and Rocha [3] and Arias et al. [1], ensuring that every symbolic step corresponds to a valid refinement of the concrete model space. Our completeness result follows from a bounded two-phase enumeration argument within the user-declared bounds, guaranteeing that our generation strategy can construct any satisfying concrete model instance that exists within those bounds.

State Space Folding. To ensure feasibility, MMF employs state folding to prune redundant search paths. This technique relies on a subsumption relation that approximates semantic containment. As formalized by Arias et al. [1] and Meseguer [15], a correctly defined subsumption relation ($\preceq$) ensures that if a state S is subsumed by an existing representative S', pruning S preserves completeness because all constrained model instances denoted by S are already covered by S'. MMF implements this via graph shape indexing and logical entailment checks.

Comparison with specific approaches. Riesco's testing techniques exploit Maude's symbolic engines (notably narrowing) to generate concrete witnesses [17]. However, Riesco targets test input generation for functional correctness of operational semantics, whereas MMF focuses on structural model finding under schema constraints, using SMT integration primarily for satisfiability of data invariants rather than full symbolic execution. Escobar et al. develop folding optimisations for SMT-based symbolic reachability based on narrowing [9]. MMF adapts these principles but specializes the redundancy layer (shape indexing, ACU-matching, and entailment-based subsumption) to exploit order-only symmetries induced by the equational theory (notably ACU) and to reduce redundancy in strictly bounded structural generation.

9 Conclusion and Future Work

This paper presented the Maude Model Finder (MMF), a native approach to bounded model finding in the Maude ecosystem. By formalizing bounded model generation as symbolic transition exploration over runtime states within rewriting logic, MMF allows users to validate structural and data-level constraints

without departing from Maude's framework. Our approach synthesizes the structural generation techniques of SAT-based finders with symbolic rewriting modulo SMT, using SMT strictly as a background solver for constraint satisfiability and entailment, and leveraging folding to quotient redundant symbolic states. We establish termination, soundness, and completeness of the bounded construction within the declared bounds, and justify folding via a coverage-preserving subsumption test.

Several avenues for future work remain. First, we plan to generalize the current meta-level exploration engine by integrating Maude's strategy language [8], offering users a declarative way to customize search heuristics and pruning policies. Second, while our current symmetry reduction relies on canonical construction and post-hoc folding, we aim to investigate constructive symmetry breaking techniques to further prune the frontier. Finally, we intend to expand the supported constraint language to cover a richer subset of OCL, enhancing MMF's utility for MDE applications.

Acknowledgements

The author is grateful to José Meseguer and Santiago Escobar for insightful discussions and detailed feedback on drafts of this article.

References

1. Arias, J., Bae, K., Olarte, C., Ölveczky, P.C., Petrucci, L.: A Rewriting-logic-with-SMT-based Formal Analysis and Parameter Synthesis Framework for Parametric Time Petri Nets. Fundamenta Informaticae **192**(3-4), 261–312 (September 2024). https://doi.org/10.3233/FI-242195, publisher: SAGE Publications
2. Bae, K.: Rewriting-based model checking methods. Ph.D. thesis, University of Illinois at Urbana-Champaign (September 2014), https://hdl.handle.net/2142/50553
3. Bae, K., Rocha, C.: Symbolic state space reduction with guarded terms for rewriting modulo SMT. Science of Computer Programming **178**, 20–42 (June 2019). https://doi.org/10.1016/j.scico.2019.03.006
4. Boronat, A.: Structural model subtyping with OCL constraints. In: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017. pp. 194–205. ACM (2017)
5. Boronat, A.: Safe reuse in modelling language engineering using model subtyping with OCL constraints. Software and Systems Modeling **22**(3), 797–818 (Jun 2023). https://doi.org/10.1007/s10270-022-01028-7
6. 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, Lecture Notes in Computer Science, vol. 4350. Springer, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-40212-1

7. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.: Programming and symbolic computation in Maude. Journal of Logical and Algebraic Methods in Programming **110**, 100497 (January 2020). https://doi.org/10.1016/j.jlamp.2019.100497
8. Eker, S., Martí-Oliet, N., Meseguer, J., Rubio, R., Verdejo, A.: The Maude strategy language. Journal of Logical and Algebraic Methods in Programming **134**, 100887 (August 2023). https://doi.org/10.1016/j.jlamp.2023.100887
9. Escobar, S., López-Rueda, R., Sapiña, J.: Symbolic Analysis by Using Folding Narrowing with Irreducibility and SMT Constraints. In: Proceedings of the 9th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems. pp. 14–25. FTSCS 2023, Association for Computing Machinery, New York, NY, USA (Oct 2023). https://doi.org/10.1145/3623503.3623537
10. Gogolla, M., Burgueño, L., Vallecillo, A.: Model finding and model completion with USE **2245**, 194–200 (2018), https://ceur-ws.org/Vol-2245/ocl_paper_9.pdf
11. Jackson, D.: Alloy: a language and tool for exploring software designs. Commun. ACM **62**(9), 66–76 (August 2019). https://doi.org/10.1145/3338843
12. Knuth, D.E.: The Art of Computer Programming. Volume 4a, Part 1: Combinatorial Algorithms. Addison-Wesley, Boston, Mass. ; London (2011)
13. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: Bishop, J., Vallecillo, A. (eds.) Objects, Models, Components, Patterns. pp. 290–306. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21952-8_21
14. López-Rueda, R., Escobar, S., Sapiña, J.: An efficient canonical narrowing implementation with irreducibility and SMT constraints for generic symbolic protocol analysis. Journal of Logical and Algebraic Methods in Programming **135**, 100895 (Oct 2023). https://doi.org/10.1016/j.jlamp.2023.100895
15. Meseguer, J.: Symbolic computation and verification methods in maude. In: Logic-Based Program Synthesis and Transformation - 34th International Symposium, LOPSTR 2025, Rende, Italy, September 9-10, 2025, Proceedings. Lecture Notes in Computer Science, Springer (2025)
16. Meseguer, J., Thati, P.: Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. Higher-Order and Symbolic Computation **20**(1), 123–160 (June 2007). https://doi.org/10.1007/s10990-007-9000-6
17. Riesco, A.: Using Narrowing to Test Maude Specifications. In: Durán, F. (ed.) Rewriting Logic and Its Applications. pp. 201–220. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34005-5_11

Space-time deterministic graph rewriting

Pablo Arrighi[1], Marin Costes[1], Gilles Dowek[1], and Luidnel Maignan[2]

[1] Université Paris-Saclay, Inria, CNRS, LMF, 91190 Gif-sur-Yvette, France
[2] Univ Paris Est Creteil, LACL, 94000 Creteil, France

Abstract. We study non-terminating graph rewriting models, whose local rules are applied non-deterministically—and yet enjoy a strong form of determinism, namely space-time determinism. For terminating computation, it is well-known that the property of confluence may ensure a deterministic end result. In the context of distributed, non-terminating computation however, confluence alone is too weak a property. Here we provide sufficient conditions so that asynchronous local rule applications produce well-determined events in the space-time unfolding of the graph, regardless of their application orders. Our first two examples are asynchronous simulations of dynamical systems. Our third example features time dilation, in the spirit of general relativity, as a prime illustration of a phenomenon that is fundamentally asynchronous yet gives rise to a consistent space-time.

Keywords: Causal graph dynamics · Cellular automata · Covariance · Commutation · Strong confluence · Distributed computation · Task dependencies · DAG · Poset · Space-like cut · Foliation.

1 Introduction

1.1 Context

Dynamical systems refer to the global and synchronous (or almost synchronous [27]) evolution of an entire configuration at time t into another configuration at time $t + 1$, and then $t + 2$, etc., iteratively. Dynamical systems often have spatial dimensions too, as the configurations are often grid-based (e.g. for representing particles, fluids, traffic jams... But they can also be graph-based (e.g. representing physical systems, computer processes, biochemical agents, economical agents, users of social networks, etc.). One can think of cellular automata, lattice-gas automata, parallel graph rewriting, causal graph dynamics [1,6,3] or so-called global transformations [26] for instance.

Synchronism is often criticised however, on the basis that: 1/ In some contexts synchronization mechanisms are considered a costly resource, which hinders the use of parallelism to achieve high performance computing [30]. 2/ It is often dubbed as physically unrealistic. Some authors argue that nature has no central clock, and thus cannot apply the same rule everywhere at once [25]. For instance, relativistic physics departs from the idea of a global time across the universe, because its 'time covariance' symmetry makes it perfectly legitimate to evolve just a small region of space, whilst keeping the rest unchanged.

Asynchronism—the application of the local rule at totally arbitrary places, non-deterministically—fits this picture better and is also well studied, especially when it is

more compelling to leave the evaluation strategy under-determined. This is typically the case in rewriting theory, when the rewrite rules arise as a computationally-oriented version of an equality, e.g. $1 + 1 \to 2$. Then, term $1 + 1 + 1$ may evolve into $2 + 1$ or $1 + 2$, non-deterministically. This feels right, because: 1/ an underlying symmetry tells us there is no reason to favour one over the other and 2/ ultimately, if what matters is the end result, we are reassured by the fact that $2 + 1 \to 3$ and $1 + 2 \to 3$.

More generally, *confluence* of a set of rewrite rules is the property that if the evolutions $a \to^* b$ and $a \to^* c$ are possible, then the evolutions $b \to^* d$ and $c \to^* d$ are also possible for some d. In the context of terminating computation, this ensures that the non-determinism introduced by the order of application of the rewrite rules does not matter to the end result, yielding a well-determined, unique normal form, e.g. 3 in the above example. Contributions [15,23,20,10,22] deal with the case when the computational process eventually produces a graph as a result, through successive rewrites.

1.2 Motivation

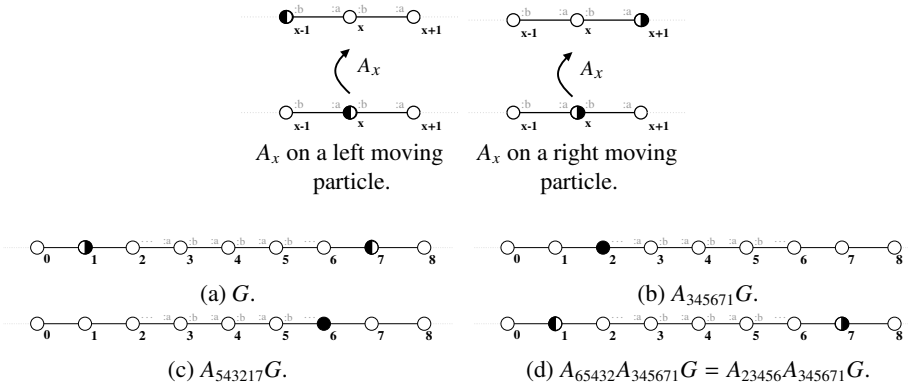


Fig. 1: *Confluence yet inconsistencies.* The local rule transports left-moving particles to the left, and right-moving particles to the right, where left/right are distinguished thanks to the a/b ports. (a) We start with a left-moving particle in 7 and a right-moving particle in 1. (b)&(c) The point of collision between the two particles is not well defined, it depends on the evaluation strategy. Here $A_{71}G$ is short for $A_7(A_1G)$. (d). Still, the system is confluent, as the divergent configurations can be both evolved into the last.

We are interested in computational processes that generate not just one result, but a non-terminating succession of them. Unfortunately, confluence alone hardly provides any guarantee in such cases as, for instance, rewrite rules yielding every possible result are computationally meaningless but trivially proven confluent. The use of labelling techniques [24] aiming at safeguarding the successive results from being rewritten can still save the day at this stage. But things will go out of hand if we are interested in modelling not just one computational process, but an entire network of interacting

processes, each generating results that cannot be safeguarded, because they are being consumed by the neighbouring processes to produce their next result, etc., i.e. non-terminating distributed computation over linear resources, e.g. particle systems driven by number-conserving rules. For instance, Fig. 1 shows left and right-moving particles on a line, where left versus right is indicated by the ports along the edges. The local rule simply has them move, which means consuming the particle at vertex, to give it to another. A little thought shows that the system is confluent. Yet, depending upon the chosen order of applications, one finds that one particle propagates much faster than the other, yielding the collision event to occur at very different places. This means that spacetime events ill-defined, inconsistent. Whilst this example is designed to emphasize these issues, it is clear that, when it comes to dynamical systems, asynchronous evaluation strategies may lead to inconsistent results, nonphysical effects (e.g. superluminal signalling), and pathological dynamical behaviours (e.g. over Boolean networks [31,13] and cellular automata [35,17]). This is a major deterrent to their usage, and the notion of confluence is no fix to that.

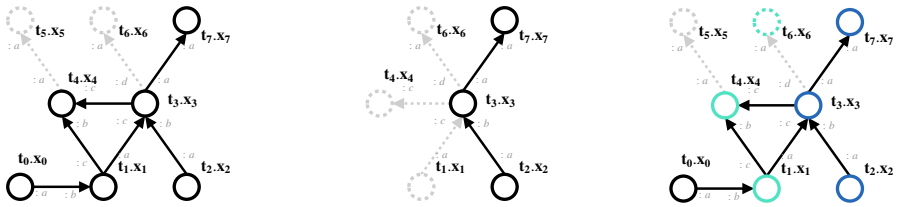
In order to fix this, we introduce the notion *space-time determinism* in two variants. In both cases, the idea is to move beyond the notion of confluence, which traditionally compares configurations globally while ignoring their spatial and local structures, and introduce a spatially and locally refined notion instead. It will ensure the existence of well-defined, interrelated events constitutive of a global "space-time diagram" (in the sense of cellular automata or Physics). The first approach, referred to as *weak space-time determinism*, focuses on maintaining consistent a certain class of events—that can be thought of as "results"—across all possible evaluation strategies. The second approach, called *full space-time determinism*, considers all events instead—they can be thought of as both "partial results" and results of the previous class. Intuitively, it asks that two snapshots of the evolving graph that have consumed the same information in some region, be consistent upon the (partial) results in this region—regardless of the evaluation strategies used to compute each snapshot. The main theorems of this paper show that some natural, local conditions upon the local rule, entail (weak or full) space-time determinism.

Dynamical geometry. We do not restrict ourselves to work over a fixed lattice or a fixed boolean network, here. Our processes are initially laid out in some network, but then they can connect with the neighbours of their neighbours, disconnect etc., making the network evolve. An important insight is that in order to obtain space-determinism, the network needs be directed, and understood as a DAG of dependencies between the processes. The DAG of dependencies is both a constraint upon the evolution, and a subject of the evolution, allowing us to express intriguing effects such as time dilation, reminiscent of general relativity, see Fig. 9.

Applications. Whilst mainly of theoretical nature, this work does suggest efficient parallel schemes for the implementation of dynamical system and distributed systems—doing away with any expensive clock synchronisation mechanisms and replacing it with just a cheap DAG. Another strand of applications lies in Physics, namely seeking for mathematically sound, constructive frameworks for discrete models of general relativity [36]. From a general philosophy of science point of view, we find it compelling to reconcile asynchronism and determinism.

Plan of the paper. Sec. 2 lays out the formalism for graphs and local rules. Sec. 3 provides an example of how to perform the asynchronous simulation of a particle system, at the cost of introducing "metric" information in the form of these DAGs so as to capture the relative advancement of the computation in a region with respect to another. Sec. 4 generalizes this construct to simulate any cellular automaton. Sec. 5 shows how, having introduced this DAG, one can manipulate it to achieve time dilation effects. An analogy is drawn with various concepts coming from general relativity, such as time covariance, metric, background-independence and dynamical geometry. Sec. 6 rigorously defines the weak/full space-determinism properties motivated by these examples. It introduces a set of local conditions, also met by the examples, and demonstrates how these entail weak/full space-determinism. These theorems constitute our main technical contributions, with full proofs found provided in the long version of this paper. Sec. 7 summarizes the results, compares them to the related works, and provides some perspectives.

2 Graphs and local rules



(a) G (b) $G_{\{x_2,x_3,x_7\}}$ (c) G with X and X^-.

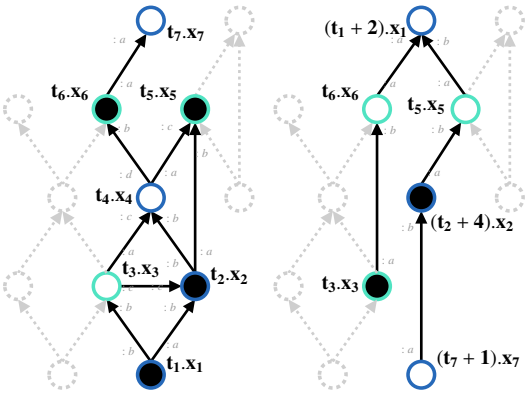
Fig. 2: *Induced subgraph and borders (a)(b):* We consider a graph G and its induced subgraph $G_{\{x_2,x_3,x_7\}}$. Graphs have borders, as shown pointed by the dashed lines for (a) G and (b) $G_{\{x_2,x_3,x_7\}}$. *Interior of a set (c):* We consider a set of positions $X = \{x_1, x_2, x_3, x_4, x_6, x_7\}$. Positions in this set are either interior (denoted X^- and shown in dark blue) or in the boundary ($X \setminus X^-$ in cyan).

We start by formally introducing the type of graphs that we consider: directed acyclic labelled port graphs. The use of labelled port graphs is is totally standard in distributed computing [12] This is because ports are mandatory for expressiveness, in order to be able to tell a neighbouring process from another. So are the labels of the nodes, in order to capture the internal state of each process. The DAG structure is mandatory to capture the dependency between the processes: this is also quite common [9,30].

Each vertex $t.x$ may be understood as an event, featuring a computational process at position x and time tag t.

Definition 1 (Positions, ports, states, names). *Let $\mathcal{X}$ be an infinite countable set of* positions*. Let π be a finite set of* ports *and Σ be a finite set of* states*. The set of* names *is defined as $\mathcal{V} := \{\, t.x \mid t \in \mathbb{Z}, x \in \mathcal{X} \,\}$.*

For any subsets $U \subseteq \mathcal{V}$ and $X \subseteq \mathcal{X}$, let us define $(U:\pi) := \{\,(u:a) \mid u \in U,\, a \in \pi\,\}$ and $\mathcal{X}(U) = \{\,x \mid t.x \in U\,\}$. Let us also denote $\overline{X} := \mathcal{X} \setminus X$, and $\mathbb{Z}.X := \{\,t.x \mid t \in \mathbb{Z}, x \in X\,\}$. Given any $u = t.x \in \mathcal{V}$, let us denote $t'.u$ for $(t'+t).x \in \mathcal{V}$.



(a) G (b) $A_{x_1} G$

Fig. 3: A local rule application A_{x_1} is a function from the entire graph to the entire graph which only affects the vertices at position $\mathcal{N}_{x_1}(G)$. These are either interior vertices (circled in dark blue) or boundary vertices (circled in cyan). These boundary vertices, and their external edges, must be preserved, although their internal (here white/black) internal states can be update. The whole interior can be modified almost arbitrarily.

Definition 2 (Graphs, Past). *A* graph *G is given by a tuple $(\mathrm{I}_G, \mathrm{B}_G, \mathrm{E}_G, \sigma_G)$ where:*

- *$\mathrm{I}_G \subseteq \mathcal{V}$ has its elements called* internal vertices of *G,*
- *$\mathrm{B}_G \subseteq \mathcal{V}$ has its elements called* border vertices of *G,*
- *$\mathrm{E}_G \subseteq ((\mathrm{I}_G \cup \mathrm{B}_G):\pi)^2 \setminus (\mathrm{B}_G:\pi)^2$ has its elements called* (oriented) edges*, and*
- *$\sigma_G : \mathrm{I}_G \to \Sigma$ maps each internal vertex to its states.*

We denote $\mathrm{V}_G := \mathrm{I}_G \cup \mathrm{B}_G$. This tuple has to be such that:

- *Vertex partitioning:* $\mathrm{I}_G \cap \mathrm{B}_G = \emptyset$,
- *Unicity of positions:* $\forall t.x, t'.x' \in \mathrm{V}_G,\ x = x' \Rightarrow t = t'$,
- *Port non-saturation:* $\forall (u:a, v:b), (u':a', v':b') \in \mathrm{E}_G,\ u:a \neq v':b' \ \wedge\ u:a = u':a' \Leftrightarrow v:b = v':b'$,
- *Border attachment:* $\forall u \in \mathrm{B}_G,\ \exists (v:a, v':a') \in \mathrm{E}_G,\ u \in \{\,v, v'\,\}$*, and*
- *Acyclicity:* $\forall n \in \mathbb{N},\ \forall \langle (u_i : a_i, v_i : b_i)\rangle_{i\in\{1,\dots,n+1\}} \in {\mathrm{E}_G}^{n+1}$ *s.t.* $(\forall i \in \{1,\dots,n\},\ v_i = u_{i+1}),\ u_1 \neq v_n$.

We denote by $\mathcal{G}$ the set of all graphs, and by $\mathrm{Past}(G) \subseteq \mathrm{V}_G$ *the vertices of G with no incoming edges.*

Summarizing, each position x only appears once, each vertex port $:a$ can only be used once per vertex, and border vertices are only there to express dangling edges of internal vertices. The $\mathrm{Past}(G)$ vertices intuitively stand for computational processes that are no longer awaiting for results by others, and are therefore ready to be executed.

Notations. Given a subset $U \subseteq \mathcal{V}$, we define the induced subgraph $G_U \sqsubseteq G$ as the graph whose internal vertices I_{G_U} are given by $I_G \cap U$, and whose edges E_{G_U} are all those edges of G which touch a vertex in I_{G_U}. Thus, its border vertices B_{G_U} are those nodes of $V_G \setminus I_{G_U}$ which lie at distance one of I_{G_U} in G. For a subset $X \subseteq \mathcal{X}$, we write G_X for the induced subgraph $G_{\mathbb{Z}.X}$ (see Fig. 2).
We also introduce the operation $G \sqcup H$, which is only defined if G and H can both be obtained as induced subgraphs of the some graph, in which case $G \sqcup H$ is the smallest such graph. In particular, this implies that if $u \in I_G$, $v \in \mathrm{B}_G$, $v \in I_H$ and $(u\!:\!a, v\!:\!b) \in E_G$, it must be the case that $(u\!:\!a, v\!:\!b) \in E_H$ and $u \in V_H$. This union will allow us to express locality.
We denote X^- the set $\{x \in X \cap \mathcal{X}(\mathrm{I}_G) \mid \mathrm{V}_{G_x} \subseteq \mathbb{Z}.X\}$, and call these the interior vertices of X in G. Its other vertices, namely $X \setminus X^-$, are referred to as the boundary of X in G, see Fig. 2. Notice that the notion of border and the notion of boundary are quite different.

Notice that $G = G_X \sqcup G_{\overline{X}}$. This decomposition will allow us to define the action of our local rules. We will proceed as follow. First we define a *neighbourhood scheme* $\mathcal{N}$ which, given a position x, selects a subset of nearby positions $\mathcal{N}_x(G)$. Second we define *operators* $A_{(-)}$ which, given a position x, rewrite the graph G as $A_x G$. Third we say that $A_{(-)}$ is $\mathcal{N}$*-local* if the action of A_x is to replace just the left hand side of $G = G_{\mathcal{N}_x(G)} \sqcup G_{\overline{\mathcal{N}_x(G)}}$, independently of the right hand side—an operation which can likely be formalised as a double push-out [8].

Definition 3 (Neighbourhood scheme). *A* neighbourhood scheme $\mathcal{N}$ *is an operator from* $\mathcal{X} \times \mathcal{G}$ *to* $\mathcal{P}(\mathcal{X})$ *mapping a pair* (x, G) *to* $\mathcal{N}_x(G) \subseteq \mathcal{X}$ *such that :*

- *Reachability: For every* $y \in \mathcal{N}_x(G)$ *there exists a directed path from* x *to* y*.*
- *Extensivity:* $G_{\mathcal{N}_x} \sqsubseteq H \sqsubseteq G$ *implies* $G_{\mathcal{N}_x} = H_{\mathcal{N}_x}$;

When G *is clear from context, we write* $\mathcal{N}_x$ *instead of* $\mathcal{N}_x(G)$*.* $\mathcal{N}_x^-$ *refers to the internal vertices of* $\mathcal{N}_x$ *with respect to* G*.*

Notice that our definition of neighbourhood schemes is quite permissive. It even allows for unbounded, or infinite neighbourhood (if the directed edges are interpreted as lightlike this still respects bounded speed of propagation of information), even though for all practical purposes one is likely to use a bounded neighbourhood. Our results will apply so long as the abstract conditions are met. For some of them we need extensivity, which is there to forbid that global criteria be used to decide whether some closeby vertex belongs to the neighbourhood or not. It demands that the neighbourhood $\mathcal{N}_x(G)$ as computed by the function $\mathcal{N}$ over a graph G, be the same as that computed over any subgraph H of G that comprises the neighbourhood. It can be understood as a form of graph-locality of the neighbourhood scheme $\mathcal{N}$ itself. Typically, if $\mathcal{N}_x(G)$ is computed step by step starting from x until 'hitting wall' i.e. a local ending criterion, then it will be strongly extensive, i.e. $G_{\mathcal{N}_x} \sqsubseteq G$ implies $G_{\mathcal{N}_x} = H_{\mathcal{N}_x}$. Extensivity is a bit more permissive, e.g. in a segment, we can grab the first dozen closest neighbours if they exist, and decide to keep none of them otherwise.

Next we define the notion of a local rule $A_{(-)}$. Before we proceed, remember that our graphs can never contain both vertices $u = t.x$ and $u' = (t+1).x$. It follows that the local rule A_x will act unambiguously on the unique vertex of the form $u = t.x$. Keep in mind also that our directed edges stand for dependencies between events, i.e. if $u = t.x$ points

to $v = t'.y$, then v is considered ahead in time of u, and thus frozen awaiting information from u. The action of some local rule A_y on v is therefore trivial, preventing y to be computed too far ahead and thereby providing a weak synchronisation mechanism. The action of A_x on u is non-trivial only if u is minimal aka Past. Such a u can be thought of as lagging behind in time, and no longer awaiting for any information—it has reached its local normal form. The action of A_x is to dispose of it by communicating its information to v and other dependencies, and creating vertex $u' = (t + \Delta t).x$ in some provisional state.

Definition 4 (Local rule). *A* local rule *is an operator over graphs $A_{(-)} : \mathcal{X} \to (\mathcal{G} \to \mathcal{G})$ which is $\mathcal{N}$-local for some neighborhood scheme $\mathcal{N}$, i.e. for all G in $\mathcal{G}$, for all x in X,*

$$A_x G = \begin{cases} (A_x G_{\mathcal{N}_x}) \sqcup G_{\overline{\mathcal{N}_x}} & \text{if } x \in \mathrm{Past}(G) \\ G & \text{otherwise} \end{cases}$$

where the action of A_x on $G_{\mathcal{N}_x}$ must be context-preserving—i.e. *it preserves the border $B_G = B_{A_x G}$ and its border edges $e \in \mathrm{E}_G \setminus (\mathrm{I}_G : \pi)^2 \implies e \in \mathrm{E}_{A_x G}$. From now on A_{yx} will stand for $A_y A_x$. We say that $\omega \in \mathcal{X}^*$ is a valid sequence in G if, for all $\omega_1, \omega_2 \in \mathcal{X}^*$ such that $\omega = \omega_2 x \omega_1$, we have $x \in \mathcal{X}(\mathrm{Past}(A_{\omega_1} G))$. We denote $\Omega_G(A) \subseteq \mathcal{X}^*$ the set of valid sequences in G.*

The fact that $A_x(G_{\mathcal{N}_x}) \sqcup G_{\overline{\mathcal{N}_x}}$ needs to exist for any G implies that $A_x(G_{\mathcal{N}_x})$ and $G_{\overline{\mathcal{N}_x}}$ must "agree". A consequence is that new vertices $V_{A_x G} \setminus V_G$ are exclusively vertices at positions $\mathcal{N}_x^-(G)$ with time tags modified (e.g. the vertices circled in dark blue in Fig. 3). More precisely,

Lemma 1. (Position preservation). *Let $A_x : \mathcal{G} \to \mathcal{G}$ be a local rule, then $\mathcal{X}(I_{A_x G}) \subseteq \mathcal{X}(I_G)$ and all new names are at positions of $\mathcal{N}_x{}^-$—i.e. $\mathcal{X}(I_{A_x} \setminus I_G) \subseteq \mathcal{N}_x{}^-$.*

An additional property that can be required, without difficulty, of our operators, is renaming-invariance [4]. We will skip this here.

Having defined the considered graphs and the kind of local transformations allowed on them, let us state our goal informally. Consider a graph G, a local rule $A_{(-)}$ and all possible sequences $\omega, \omega', \ldots \in \Omega_G(A)$. If one applies A_ω, one obtains one possible evolution of the system. But $A_{\omega'}$, $A_{\omega''}$,... are equally legitimate different orders of local rule applications. We aim to define precisely what it means for all these possible evolutions to actually agree on a consistent common story, i.e. a *consistent space-time diagram*. To motivate the remaining formalization, we consider examples. Later we give sufficient conditions for a local rule to induce such consistent space-time diagrams.

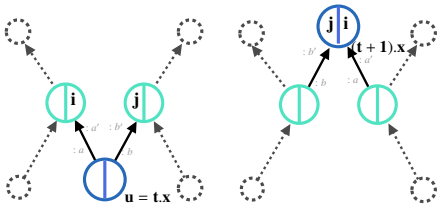
Definition 5 (Space-time diagram). *Given a graph G and a local rule $A_{(-)}$,* space-time diagram *$\mathcal{M}_A(G) := \{ A_\omega G \mid \omega \in \Omega_G(A) \}$ is the set of all generated graphs. We sometimes omit A and G.*

To visualize how the graphs in $\mathcal{M}$ share common vertices and edges, some *space-time backgrounds* are depicted (Figs 5, and 9). These space-time backgrounds are "pseudo-graphs", i.e. graph in a common sense, without any of the port-constraints of Def. 2 imposed. Each pseudo-graph M is defined by:

$$V_M = \bigcup_{G \in \mathcal{M}} V_G \qquad\qquad E_M = \bigcup_{G \in \mathcal{M}} \mathrm{E}_G.$$

3 Particle system example

We now show how asynchronous applications of a local rule can represent a dynamical system of left and right-moving particles, consistently, thereby fixing issues of Fig. 1.



(a) $G_{N_{x_1}}$. (b) $A_{x_1} G_{N_{x_1}}$.

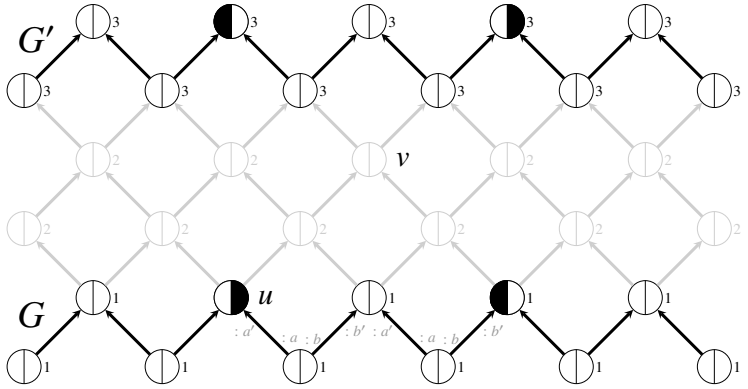
Fig. 4: *The local rule for the particle system.* A_x acts by consuming the internal state $i = \sigma^r_G(v)$ of vertex v (and symmetrically with $j = \sigma^l_G(w)$), thereby moving those particles at x if they are present. It also updates ports from a, b to a', b', flips the arrows pointing to x, and increments its timetag, in order to move the vertex from past $u = t.x$ to future $u' = 1.u = (t + 1).x$. Dashed edges and vertices do not influence the local rule.

Again the vertices of our graphs have names of the form $u = t.x$, they must be thought of as events in space-time. Here internal states are pairs of bits representing the presence of a left-moving particle or not, and of a right-moving particle or not, i.e. $\sigma_G(u) = (\sigma^l_G(u), \sigma^r_G(u))$. Each edge goes either from port $:a$ to $:a'$ or from port $:b$ to $:b'$, thereby indicating a spatial direction (a versus b) and a temporal orientation (unprimed versus primed). Altogether each graph must be thought of, not as a space-time diagram, but as a 'space-like cuts' of a space-time diagram, i.e. a snapshot, cf. Fig. 5a.

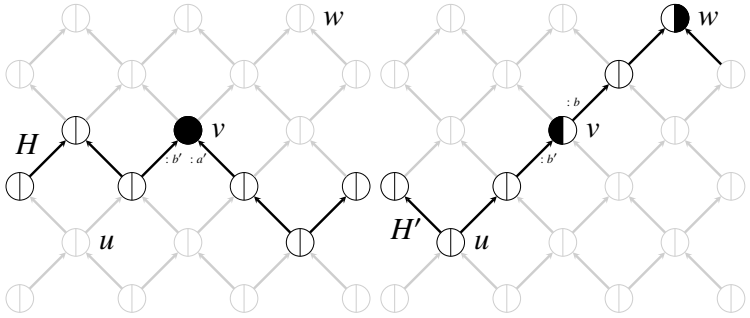
The local rule is given by Fig. 4. It allows us to evolve one graph G, understood as a space-like cut (a snapshot), into another later space-like cut G', as in Fig. 5a. It is possible to convince oneself that any sequence of applications of the local rule in the vicinity of the two particles leads the collision occurring at v and nowhere else.
Notice also that the graph G can be evolved asynchronously into either H as in Fig. 5b or H' as in Fig. 5c. These graphs agree upon the trajectory of the moving particles. Let us now turn our attention towards considerations that we eventually formalize in Sec. 6. By the above two remarks, this local rule feels "space-time deterministic", as there is a sense in which it produces well-determined events in space-time. But this property is a bit subtle to formulate, e.g. the state associated to the event v seems different in H and H', as the particle got 'consumed' from v to w. Still, the state of v remains a function of the way it gets traversed by the snapshot H or H', as represented here by its set of incoming ports to v. This motivates the notions of full consistency and full space-determinism which we give in Def. 6.

Instead of entering this subtlety, we could have restricted ourselves to local rules that do not consume the particle, e.g. leaving a trail of the left-mover at v in H'. But the 'particle consumption mechanism' of this example is something that we do want to be able to capture. This, we admit, is for reasons that lie beyond the scope of this paper, but we can mention them in passing: 1/ In quantum theory, if the particle is not consumed as it moves, it leaves a trail behind that entangles several sites. Due to this

entanglement, ignoring the trail decoheres the particle, making it to behave classically. This is essentially the no-cloning theorem. We want to make sure this work is compatible with a quantum extension. 2/ In quantum field theory, it is also the case that the quantum state associated to a point in space-time is well-determined only if we specify the angle at which it gets cut by the spacelike cut. The study of this example sets the aim of the paper: to understand when asynchronism meets space-time determinism.



(a) Particle system example



(b) $\sigma_H(v) = (0, 1) \ldots$ (c) but $\sigma_{H'}(v) = (0, 0)$.

Fig. 5: *Particle system example (a):* In black we highlight just the graphs G and G' belonging to the space-time diagram $\mathcal{M}_A(G)$. The local rule here moves the left-side particle towards the right and the right-side particle towards the left. Note how the problem raised by Fig. 1 is solved. The only point in space-time which can contain both particles is u. *States depend on the cut (b)(c):* Both graphs H and H' belong to the same space-time diagram $\mathcal{M}_A(H_0)$ with H_0 containing a right-moving particle in u. They both contain vertex v. In H', the particle that was present in H has moved to point w. The state associated to v thus depends on the way it is cut, which is captured by its set of incoming ports e.g. $\{a', b'\}$ versus just $\{a'\}$.

4 Simulating synchronous cellular automata

Beyond the above particle system example, we now show that we can simulate any 1D cellular automata (CA for short) in spite of its synchronicity. The construction borrows to the well-studied 'marching soldiers' scheme [18], as best formalised by [29] , but relies on the DAG of dependency rather than extra states as its mechanism for local, relative synchronisation. Without loss of generality [21], we simulate radius half CA.

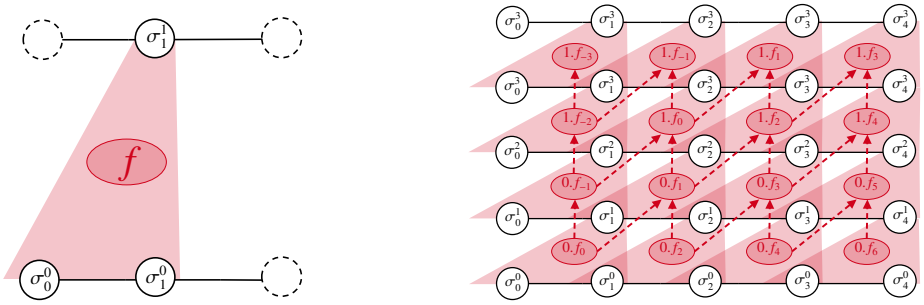
First let us recall that a radius one half cellular automaton acting on an alphabet Σ_{CA} is a global function $F : {\Sigma_{CA}}^{\mathbb{Z}} \to {\Sigma_{CA}}^{\mathbb{Z}}$ defined by the synchronous application of a local function $f : {\Sigma_{CA}}^2 \to \Sigma_{CA}$ everywhere at once, cf. Fig. 6a. The dependencies between the different applications of f are shown in Fig. 6b, notice the similarity with Fig. 5a. This suggests that the natural way to encode this cellular automaton in our model is to use vertices (a.k.a events) to represent each application of f.

Second we define $\Sigma = (\Sigma_{CA} \cup \epsilon)^2$ and pick the same neighbourhood scheme and ports as in the previous example. We define the local rule as depicted in Fig. 7a and Fig. 7b. Its action can be understood as follow: 1/ applied to the vertex named $0.f_0$, it applies $f(\sigma^0_0, \sigma^0_1)$ 2/ and stores the result both in the right part of the vertex $0.f_{-1}$ and in the left part of the vertex $0.f_1$. 3/ Finally it creates the vertex $1.f_0$, which is awaiting for the output of the $0.f_{-1}$ and $0.f_1$ applications, as represented by the two incoming edges.

We encode the initial configuration σ^0 by a graph G (see Fig. 7c) which contains two copies of each internal state in σ^0. The dynamics will indeed generate every configuration $\sigma^i = F^i(\sigma^0)$. The space-time diagram is similar to that of Fig. 5a. The scheme can easily be generalised to d-dimensional CA, and is likely to work for any local synchronous evolution in a broad sense ; the idea being to use vertices to encode applications of the local rule, and edges to represent applications causal relationships.

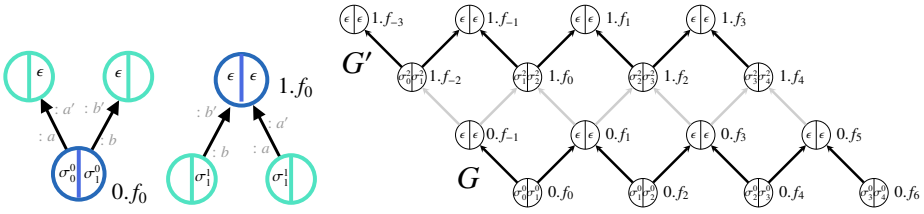
5 Beyond synchronous simulation

The previous examples are simulations of synchronous systems. Beyond these, we are interested in expressing systems that remain deterministic but are genuinely asynchronous. Such local rules still have fully consistent space-time diagrams: the only thing which is lost is the possibility to interpret these through the lens of a global clock. A paradigmatic, physics-inspired example for this whole class of systems is the following time dilation example. The time dilation example extends the internal state space Σ used in Sec. 3 with two more states: the green and red states. The same local rule is extended so that if one such green/red particle is found at position x, it will stay there, oscillating between both colours and altering the very texture of space-time, cf. Fig. 8. This results in time dilation as can be seen from Fig. 9. For instance, if two identically made clocks were modeled out of a signal oscillating from left to right and right to left between two neighbouring nodes, the clock lying on the right-hand-side of the green/red particle would tick twice as slowly as the one lying on the left-hand-side. Yet, the very same local rule is being applied left and right of the green/red particle. Such a phenomenon is reminiscent of general relativity, e.g. time flows slightly slower on Earth than it does in the stratosphere, as measured by identically made atomic clocks. Yet, the same laws of Physics apply in the stratosphere and on Earth. How did we get there?



(a) Radius one half locality ... (b) ... results in the above global evolution.

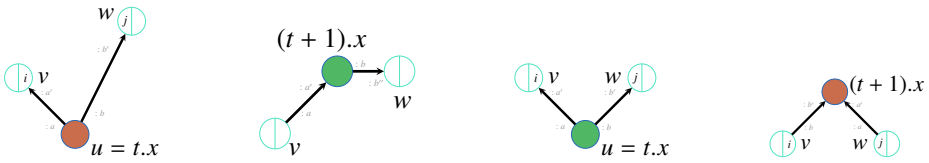
Fig. 6: *Radius one half local rule (a):* A radius one half local rule f takes two states as input ($\sigma_0^0, \sigma_1^0 \in \Sigma_{CA}$) and outputs $\sigma_1^1 = f(\sigma_0^0, \sigma_1^0)$. *Global evolution (b):* The initial configuration is an infinite one dimensional array $(\sigma_i^0)_{i\in\mathbb{Z}}$. We consider the cellular automaton which apply f homogeneously in space. In black we depicted the successive configurations computed by this automaton (σ_k^i is just short for $f(\sigma_{k-1}^{i-1}, \sigma_k^{i-1})$). In red we drew each time the local rule f gets applied. We gave a name to each occurrence to introduce more easily the simulation scheme of Fig. 7c. Finally we used red arrows to depict the dependencies between each occurrence of f, for example since $0.f_1$ needs the output σ_1^1 of $0.f_0$ to compute, we drew an arrow $0.f_0 \to 0.f_1$.



(a) $G_{\mathcal{N}_{f_0}}$. (b) $A_{f_0} G_{\mathcal{N}_{f_0}}$. (c) Initial graph and space-time diagram.

Fig. 7: *The local rule for CA simulation (a) and (b):* A_{f_0} acts exactly as A_x in Fig. 4, except for the internal states. If $\sigma(0.f_0) = (\sigma_0^0, \sigma_1^0)$ does not contain ϵ, it computes $\sigma_1^1 = f(\sigma_0^0, \sigma_1^0)$ and stores the result in both neighbours. *Space-time diagram of the simulation (c)* of the cellular automaton in Fig. 6b. The highlighted G encodes the initial cellular automaton configuration σ^0 of the CA, whilst G' encodes σ^2 the configuration obtained after two time steps of the cellular automaton.

We argue that, to some extent, the construction of this model of computation mimics some of the key steps of the derivation of general relativity theory from physical symmetries—as found in standard textbooks [28]. The developed analogy can safely be skipped by the reader with lesser interest in Physics. Indeed, let us remind the reader that the textbook derivation proceeds by: 1/ Assuming the existence of a well-determined space-time. 2/ Requiring covariance, i.e. invariance under changes of coordinates, which implies a form of asynchronism as one can choose coordinates whereby one region



(a) $G_{\mathcal{N}_x}$. (b) $A_x(G_{\mathcal{N}_x})$. (c) $H_{\mathcal{N}_x}$. (d) $A_x(H_{\mathcal{N}_x})$.

Fig. 8: *The local rule for time dilation.* We define here the behaviour of A_x when $u = t.x$ is colored—i.e. $\sigma(u) = green \vee \sigma(u) = red$. In both the cases (a) and (c), incoming particles get destroyed and we flip the color at x. On a green vertex case (c), A_x behaves as expected. On a red vertex case (a) A_x creates an anomaly. The edge between $(t + 1).x$ and w is reversed, thus we will be forced to update v once more, before we update w.

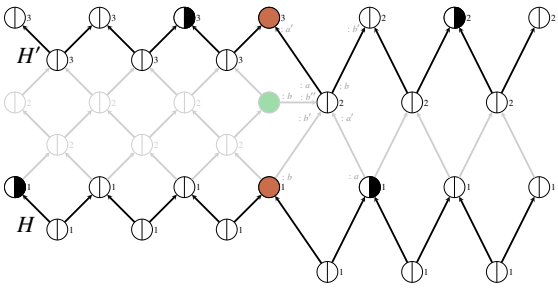


Fig. 9: *Time dilation example.* In black we highlight two graphs H and H' belonging to the space-time diagram. We start with two particles, one left and one right of the red. Notice how, even if the same local rule gets applied everywhere, times flows twice as fast for the left particle.

of a space-like cut will evolve (large time lapse), but not the other (small time lapse). 3/ Concluding that in order to obtain covariance, one needs to provide extra causality structure at each point, namely the metric field. 4/ Assuming background-invariance, namely enabling the possibility that space-time be curved by the presence of this newly allowed metric. 5/ Providing a dynamic upon the metric itself.
Here, in the discrete, we: 1/ enforced a well-determined space-time, 2/ in spite of an asynchronous evaluation strategy, 3/ by introducing an extra causality structure, namely a DAG of dependencies. 4/ We then allowed ourselves to consider graphs with exotic such DAG, and 5/ rules manipulating them.

6 Obtaining space-time determinism

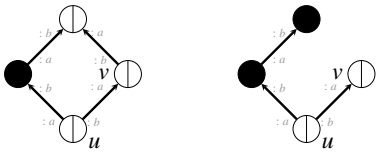
The space-time determinism of a local rule $A_{(-)}$ is the idea that the graphs that it generates from a seed G, which altogether form a space-time diagram $\mathcal{M}_A(G)$, are consistent with one another, as regards the events that they both describe. One might have hoped for a naive notion of consistency of $\mathcal{M}_A(G)$, whereby any two graphs of the space-time diagram must agree on the state of a vertex $u \in \mathcal{V}$, if it so happens to appear in both of them. But the example in Sec. 3 (Fig. 5) shows that things are more subtle. Its discussion motivates a definition of full consistency whereby any two graphs of $\mathcal{M}_A(G)$ must agree on the state of a vertex v (in terms of its neighbourhood and internal state) whenever they agree on the set of incoming ports to that vertex v. In particular this implies that the state

of v should be the same for every graph of $\mathcal{M}_A(G)$ such that $v \in \mathrm{Past}(G)$, which can be understood as stating that the "result state" (a.k.a. normal form) at v is well-determined. We refer to this weaker demand as weak consistency (see Fig. 10).

Definition 6 (Consistency). *Two graphs G and H are* fully consistent *iff for all $v \in \mathrm{I}_H \cap \mathrm{I}_G$:*

$$\pi.\mathrm{E}_G(v) = \pi.\mathrm{E}_H(v) \Longrightarrow G_v = H_v,$$

where $\mathrm{E}_G(v)$ is the set of edges ending in v, and $\pi.\mathrm{E}_G(v)$ denotes the set of incoming ports of v, i.e. $\pi.E := \{\, b \mid (u : a, v : b) \in E \,\}$. Full consistency is denoted $G \mathbin{/\!\!/} H$. The graphs G and H are called weakly consistent if they respect this condition in the special case where $\mathrm{E}_G(v) = \emptyset$. We say that a space-time diagram $\mathcal{M}$ is fully (respectively weakly*)* consistent *if each pair of graphs in $\mathcal{M}$ is fully (resp. weakly) consistent. Finally a local rule $A_{(-)}$ is said to be fully* (resp. weakly) space-time deterministic *if and only if for all graphs G, $\mathcal{M}_A(G)$ is fully (resp. weakly) consistent.*



(a) G (b) H

Fig. 10: *Weak consistency versus full consistency.* The set of graphs $\{G, H\}$ is weakly consistent but not fully consistent. Indeed we have $G_u = H_u$ but full consistency fails in v because we have $\mathrm{E}_G(v) = \mathrm{E}_H(v) = \{ : a \}$ but $G_v \neq H_v$.

We now embark in the quest for a set of properties ensuring that an $\mathcal{N}$-local rule $A_{(-)}$ generates only fully consistent space-time diagrams. We start with two properties. The first one asks for timetags to only increase. The second one, akin to strong confluence or sequential independence in parallel graph transformations, states that independent rule applications on a graph G applied in any order lead to the same graph.

Definition 7 (Time-increasing commutative local rules). *A local rule $A_{(-)}$ is*

- time-increasing *iff* $\forall t.y \in V_G,\ \forall t'.y \in V_{A_x G},\ t \leq t'$, *with* $t < t'$ *if* $y = x$;
- commutative *iff* $\forall x, y \in \mathcal{X}(\mathrm{Past}(G)),\ A_x A_y(G) = A_y A_x(G)$.

Notice that a past vertex must remain so by locality, hence the well-definiteness of commutativity. These two properties place strong constraints on past vertices of a space-time diagram $\mathcal{M}_A(G)$: they entail weak consistency. Moreover each space-like cut is determined by its set of past elements.

Proposition 1. (Obtaining weak consistency). *Let $A_{(-)}$ be a time-increasing commutative local rule. For all graphs G, $\mathcal{M}_A(G)$ is weakly consistent.*

Proposition 2. (Pasts determine space-like cuts) *Let $A_{(-)}$ be a time-increasing commutative local rule, G a graph and $H, J \in \mathcal{M}_A(G)$. If* $\mathrm{Past}(J) = \mathrm{Past}(H)$ *then* $J = H$.

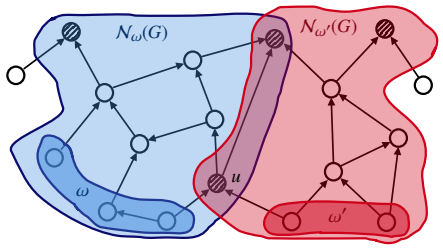


Fig. 11: *Privacy and port decreasing.* (*a*) Given a set of disjoint vertices ω' privacy demands that the neighbourhood $\mathcal{N}_\omega$ (blue) and $\mathcal{N}_{\omega'}$ (red) only intersect on their boundaries (hatched vertices). (*b*) The port decreasing condition demands that, in order to modify a vertex, A_ω must pay the price of decreasing one of its incoming private ports. Here, $u \in G_{\mathcal{N}_\omega} \cap G_{\mathcal{N}_{\omega'}}$ can be modified by both A_ω and $A_{\omega'}$ as each of them has private access to it.

However, in the example of Sec. 3, weak consistency is trivially realized even for non-trivial dynamics. This example motivated the definition of full consistency. In order to have local conditions that entail full consistency, we ask that neighbourhood schemes be private and that local rules be port decreasing. Note that these properties hold for the neighbourhood schemes and local rules used in Secs 3, 4 and 5.

In the following we denote $\mathcal{N}_\omega(G) = \bigcup_{\beta x \alpha = \omega} \mathcal{N}_x(A_\alpha G)$ the union of the neighbourhood of each vertex $x \in \omega$, at the time A_x is to be computed. Privacy then demands that any disjoint sequences ω, ω' have their $\mathcal{N}_\omega(G)$ and $\mathcal{N}_{\omega'}(G)$ intersecting only on the vertices of their boundaries (see Fig. 11), a property akin to parallel independence [15]. Privacy ensures that concurrent influences happen only at these joint boundaries, an essential ingredient of full consistency, as illustrated in Fig. 12a and 12b. They also make it easier to check whether two valid operator A_x and A_y commute. Indeed when $\mathcal{N}_y(G) = \mathcal{N}_y(A_x G)$ and $\mathcal{N}_x(G) = \mathcal{N}_x(A_y G)$, privacy allows us to only check commutation on the boundaries to ensure it globally.

Definition 8 (Privacy). *A neighbourhood scheme $\mathcal{N}$ is private, iff for any graph G and any disjoint valid sequences $\omega, \omega' \in \Omega_G(A)$, we have $\mathcal{N}^-_{\omega'}(G) \cap \mathcal{N}_\omega(G) = \emptyset$.*

Privacy may seem hard to check at first, especially given that $\mathcal{N}_\omega(G)$ depends on $A_{(-)}$. In the long version of this paper we introduce a concrete $\mathcal{N}'_\omega(G)$ which does not depend on $A_{(-)}$ as it assumes the "worse-case scenario" for it. Since $\mathcal{N}'_\omega(G)$ is private, it suffices to check that $\mathcal{N}_x(G) \subseteq \mathcal{N}'_x(G)$ to prove its privacy.

Lastly we want to forbid that a local rule modifies a vertex without altering its incoming ports, as this would immediately infringe full consistency (see Fig. 12c and 12d). Moreover the modification needs be decreasing—otherwise we could apply A_x twice in a row and get to the exact same counterexample. This idea that A_x should decrease the incoming ports of the vertex v it modifies can be understood as a natural way of 'locally reflecting the progress of the computation of v', i.e. geometrically accounting for the fact that the dependency between x and v has been reduced, and hence their space-time relationship has changed. In Sec. 3 the local rule is port decreasing because we suppress an incoming edge each time we modify a vertex. In Sec. 5 the local rule is port decreasing for more subtle reasons in the case of a red vertex: we decrease b' into b'' (see Fig. 8a and 8b).

(a) G (b) A_uG/A_vG (c) G (d) A_uG

Fig. 12: *Inconsistent dynamics examples.* The example of $(a)\&(b)$ relies on a neighbourhood scheme which is not private as $w \in \mathcal{N}_u^-(G) \cap \mathcal{N}_v^-(G)$. It follows that A_uG and A_vG disagree on the internal states of w, whilst both updating its incoming port in the same fashion (b becomes $a < b$). The example of $(c)\&(d)$ is not port decreasing. It follows that G and A_uG disagree on v but its set of incoming ports stay the same.

Definition 9 (Port decreasing). *An $\mathcal{N}$-local rule $A_{(-)}$ is* port decreasing *iff for any $G \in \mathcal{G}$, $x \in \mathcal{X}(\mathrm{Past}(G))$ and $u \in V_G \cap V_{A_xG}$ such that $G_u \neq (A_xG)_u$, we have*

$$\pi.(\mathrm{E}_G(V_G, u) \setminus \mathrm{E}_G(\overline{\mathcal{N}_\omega^-}, u)) > \pi.(\mathrm{E}_{A_xG}(V_{A_xG}, u) \setminus \mathrm{E}_G(\overline{\mathcal{N}_\omega^-}, u))$$

for an order $\leq$ over sets of ports which is for any $A, B, A', B' \in \mathcal{P}(\pi)$:

- ***inclusive*** *i.e.* $A \supseteq A' \implies A \geq A'$
- ***monotonous*** *i.e.* $A \cap B = \emptyset \ \wedge \ A \geq A' \ \wedge \ B > B' \implies A \cup B > A' \uplus B'$

where $\mathrm{E}_G(Y, u)$ is the set of edges in G from any $v \in \mathbb{z}.Y$ to u, and $\pi.E$ is as in Def. 6.

Note we ask a port decreasing operator to decrease the port of a *private edge*, i.e. an edge starting from $\mathcal{N}^-$. Without this assumption we can give a counterexample by considering the graph of Fig. 11. Say both A_ω and $A_{\omega'}$ were to modify the internal state of v and decrease the port associated to the shared edge coming from u. Then we could have $\pi.\mathrm{E}_{A_\omega G}(v) = \pi.\mathrm{E}_{A_{\omega'}G}(v)$ whilst $\sigma_{A_{\omega'}G}(v) \neq \sigma_{A_\omega G}(v)$. Finally we state our main result.

Theorem 1. (Obtaining full consistency) *Let $A_{(-)}$ be a port-decreasing time-increasing commutative $\mathcal{N}$-local rule, with $\mathcal{N}$ a private neighbourhood scheme. For all graphs G, $\mathcal{M}_A(G)$ is fully consistent.*

The proof is quite intricated, but we can give the following intuition. On the one hand the commutation hypothesis ensures that $A_\omega G \mathbin{/\!/} A_{\omega'}G$ as long as ω' is a permutation of ω. On the other hand as long as $\mathcal{N}$ is extensive, monotonous and private two disjoint operators necessarily modify different subgraphs, albeit with a common boundary. Then, by decreasing the private incoming port of each modified vertex, we obtain full consistency by dismissing its premise.

Hence, strong confluence augmented with a number of hypotheses leads to full consistency. Indeed, full consistency is a stronger requirement than confluence, as it shifts the focus from global configurations comparison to the structured, local comparison of spacetime events.

7 Conclusion

Summary of results. We introduced graphs that can be thought of as space-like cuts of space-time diagrams. The vertices have names of the form $u = t.x$, they can be

thought of as events. The edges can be thought of as dependencies between events, i.e. if $u = t.x$ points to $v = t'.y$, then v is ahead in time of u, awaiting for information from u. The action of a local rule A_x on u is non-trivial only if u is minimal: it disposes of it by communicating its information to v and other dependencies, and creates vertex $u' = (t + 1).x$ in some provisional state.
We argued that the right notion of full space-time determinism is the full consistency of its space-time diagram: the state of each event (in terms of its internal state and connectivity) needs be a function of its set of incoming ports, as these represent the angle at which the space-like cut traverses the event. We gave sufficient conditions for the asynchronous applications of a local rule to be fully space-deterministic: they must be commuting and port-decreasing, with respect to an extensive, monotonous, private notion of neighbourhood. We also looked at a weaker notion of consistency, requiring that only the normal forms of events across space-time be well-determined: commutation alone suffices then.
Throughout, we argued of the potential implications for distributed computation (weak space-time determinism), asynchronous simulation of dynamical systems (full consistency and our first and second example), and discrete toy models of general relativity (our third example).

Related works. Geometry is dynamical in our work. We thus hope it makes useful addition to the already wide literature on Graph Rewriting [34,14]. We are aware that the dominating vocabulary to describe them is now that of Category theory, in which notions of space-time diagrams have been developed e.g. [7].
We instead use the vocabulary of dynamical systems, as we came to consider Graph Rewriting through a series of works generalising CA to synchronous, causal graph dynamics [2], and tilings to graph subshifts [5]. We are confident that abstracting away the essential features of our formalism could yield interesting categorical frameworks, e.g. à la [26].
The closest works however turn out to come from varied communities. In algorithmic complexity [30] uses a DAG of dependencies representation to reduce the synchronisation costs of simulating a class of synchronous algorithms—we use it in the more context of dynamical systems in order to relax synchronism altogether, whilst safeguarding space-time determinism. In computational Physics [19] promotes the lattice of dependencies of local rule applications to a notion space-time, and advocates a notion of 'causal invariance' based on the unicity of this lattice, as formalised in the context of string rewriting [33]—we formalise space-determinism for graph rewriting mathematically, without reference to this lattice of dependencies of applications, and provide local conditions to achieve it. In the network reliability community, [18] obtains a result similar to Prop. 1—our local rules are allowed to modify the neighbouring vertices and the graph per se, and we reach full consistency.

Perspectives. Clock synchronisation is an expensive overhead for parallel simulation of dynamical systems, as well as numerous distributed computation applications. The hereby developed theoretical framework says we can just do away with them and still obtain strong forms of space-time determinism, provided that the local rule meets certain requirements. We are looking forward to see this being applied in practice. Along the way of practical applications, one ought to first try and adapt existing graph rewriting

software tools [32,16,11] for the sake of automatically checking full consistency.
We, on the other hand, are likely to focus on the reversible and quantum regimes of these graph rewriting models. Another, important open problem is to understand whether space-times of fully space-time deterministic local rules correspond to expansive graph subshifts [5], in the same way that space-times of cellular automata correspond to expansive tilings. One of the difficulties in establishing such a result is that the naming conventions we adopt for our vertices systematically prevent our space-times from being cyclic, even when the dynamics described is periodic—whereas the corresponding graph subshift will in fact be cyclic.

Acknowledgements We wish to thank Nicolas Behr for helpful discussions. This project was partially funded by the European Union through the MSCA SE project QCOMICAL (Grant Agreement ID: 101182520), by the French National Research Agency (ANR): projects TaQC ANR-22-CE47-0012 and within the framework of 'Plan France 2030', under the research projects EPIQ ANR-22-PETQ-0007, OQULUS ANR-23-PETQ-0013, HQI-Acquisition ANR-22-PNCQ-0001 and HQI-R&D ANR-22-PNCQ-0002, and by the WOST, WithOut SpaceTime project (https://withoutspacetime.org), grant ID# 63683 from the John Templeton Foundation (JTF). The opinions expressed in this work are those of the author(s) and do not necessarily reflect the views of the JTF.

References

1. Arrighi, P., Dowek, G.: Causal graph dynamics (long version). Information and Computation **223**, 78–93 (2013)
2. Arrighi, P., Martiel, S., Nesme, V.: Cellular automata over generalized Cayley graphs. Mathematical Structures in Computer Science **18**, 340–383 (2018), pre-print arXiv:1212.0027
3. Arrighi, P., Martiel, S., Perdrix, S.: Block representation of reversible causal graph dynamics. In: FCT 2015. pp. 351–363. Springer (2015)
4. Arrighi, P., Christodoulou, M., Durbec, A.: On quantum superpositions of graphs. arXiv preprint arXiv:2010.13579 (2020)
5. Arrighi, P., Durbec, A., Guillon, P.: Graph subshifts. In: Vedova, G.D., Dundua, B., Lempp, S., Manea, F. (eds.) Computability in Europe (CiE 2023). Lecture Notes in Computer Science, vol. 13967, pp. 261–274. Springer (2023). doi 10.1007/978-3-031-36978-0_21
6. Arrighi, P., Martiel, S., Perdrix, S.: Reversible causal graph dynamics: invertibility, block representation, vertex-preservation. Natural Computing **19**(1), 157–178 (2020), pre-print arXiv:1502.04368
7. Baldan, P., Corradini, A., Heindel, T., König, B., Sobociński, P.: Unfolding grammars in adhesive categories. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. pp. 350–366. Springer (2009)
8. Behr, N., Harmer, R., Krivine, J.: Fundamentals of compositional rewriting theory. Journal of Logical and Algebraic Methods in Programming **135**, 100893 (2023)
9. Bombelli, L., Lee, J., Meyer, D., Sorkin, R.D.: Space-time as a causal set. Physical review letters **59**(5), 521 (1987)
10. Bonchi, F., Gadducci, F., Kissinger, A., Sobociński, P., Zanasi, F.: Confluence of graph rewriting with interfaces. In: Yang, H. (ed.) ESOP 2017. pp. 141–169. Springer (2017)
11. Boutillier, P., Camporesi, F., Coquet, J., Feret, J., Lỳ, K.Q., Théret, N., Vignet, P.: Kasa: A static analyzer for kappa. In: CMSB 2018. pp. 285–291. Springer (2018)

12. Chalopin, J., Das, S., Widmayer, P.: Deterministic symmetric rendezvous in arbitrary graphs: overcoming anonymity, failures and uncertainty. In: Search Theory, pp. 175–195. Springer (2013)
13. Chatain, T., Haar, S., Paulevé, L.: Boolean networks: Beyond generalized asynchronicity. In: Baetens, J.M., Kutrib, M. (eds.) AUTOMATA 2018. pp. 29–42. Springer (2018)
14. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. Springer-Verlag New York Inc (2006)
15. Ehrig, H., Kreowski, H.J.: Parallelism of manipulations in multidimensional information structures. In: Mazurkiewicz, A. (ed.) MFCS 1976. pp. 284–293. Springer (1976)
16. Ermel, C., Rudolf, M., Taentzer, G.: The agg approach: Language and environment. In: Handbook of Graph Grammars, vol. 2, pp. 551–603. World Scientific (1999)
17. Fatès, N., Éric Thierry, Morvan, M., Schabanel, N.: Fully asynchronous behavior of double-quiescent elementary cellular automata. Theoretical Computer Science **362**(1), 1–16 (2006). DOI 10.1016/j.tcs.2006.05.036
18. Gács, P.: Deterministic computations whose history is independent of the order of asynchronous updating. arXiv preprint cs/0101026 (2001)
19. Gorard, J.: Some relativistic and gravitational properties of the wolfram model. arXiv preprint arXiv:2004.14810 (2020)
20. Hristakiev, I., Plump, D.: Checking graph programs for confluence. In: STAF 2017 Collocated Workshops. pp. 92–108. Springer (2018)
21. Ibarra, O.H., Jiang, T.: On the computing power of one-way cellular arrays. In: ICALP 1987. p. 550562. Springer, London, UK. (1987)
22. Jouannaud, J.P., Orejas, F.: Unification of drags and confluence of drag rewriting. Journal of Logical and Algebraic Methods in Programming **131**, 100845 (2023). DOI 10.1016/j.jlamp.2022.100845
23. Lambers, L., Ehrig, H., Orejas, F.: Efficient conflict detection in graph transformation systems by essential critical pairs. Electronic Notes in Theoretical Computer Science **211**, 17–26 (2008). DOI 10.1016/j.entcs.2008.04.026, GT-VMT 2006
24. Levy, J.J.: An algebraic interpretation of the $\lambda\beta$k-calculus and a labelled λ-calculus. In: Böhm, C. (ed.) λ-Calculus and Computer Science Theory. pp. 147–165. Springer (1975)
25. Lumer, E.D., Nicolis, G.: Synchronous versus asynchronous dynamics in spatially distributed systems. Physica D: Nonlinear Phenomena **71**(4), 440–452 (1994)
26. Maignan, L., Spicher, A.: Global graph transformations. In: Plump, D. (ed.) GCM 2015. CEUR, vol. 1403, pp. 34–49 (2015)
27. Mairesse, J., Marcovici, I.: Around probabilistic cellular automata. Theoretical Computer Science **559**, 42–72 (2014). DOI 10.1016/j.tcs.2014.09.009
28. Misner, C.W., Thorne, K.S., Wheeler, J.A.: Gravitation. W. H. Freeman and Company (1973)
29. Nehaniv, C.L.: Asynchronous automata networks can emulate any synchronous automata network. International Journal of Algebra and Computation **14**(05n06), 719–739 (2004)
30. Nishimura, N.: Efficient asynchronous simulation of a class of synchronous parallel algorithms. Journal of Computer and System Sciences **50**(1), 98–113 (1995)
31. Noual, M., Sené, S.: Synchronism versus asynchronism in monotonic boolean automata networks. Natural Computing **17**, 393–402 (2018)
32. Ölveczky, P.C., Meseguer, J.: The real-time maude tool. In: TACAS 2008. pp. 332–336. Springer (2008)
33. van Oostrom, V.: On causal equivalence by tracing in string rewriting. In: Grabmayer, C. (ed.) TERMGRAPH@FSCD 2022. EPTCS, vol. 377, pp. 27–43 (2022). DOI 10.4204/EPTCS.377.2
34. Rozenberg, G.: Handbook of Graph Grammars: Foundations, vol. 1. World Scientific (2003)
35. Schönfisch, B., de Roos, A.: Synchronous and asynchronous updating in cellular automata. Biosystems **51**(3), 123–143 (1999). DOI 10.1016/S0303-2647(99)00025-8
36. Sorkin, R.: Time-evolution problem in Regge calculus. Phys. Rev. D. **12**(2), 385–396 (1975)

A Maude Framework for Efficient Analysis of Multiformalism Models

Lorenzo Capra[1] [0000−0002−1029−1169]

Università degli Studi di Milano
Via Celoria 18, Milan, Italy
capra@di.unimi.it

Abstract. We propose a new quantitative analysis method for realistic adaptive systems that unifies two strands of work: (1) a compositional, symmetry-aware construction of a quotient transition system for lumped Markov processes in Rewritable (Stochastic) Petri nets (RwSPNs) using algebraic operators and structured node labeling, and (2) the use of Maude as a multiformalism modeling environment in which multiple formalisms coexist within one framework. We generalize the compositional RwSPN approach to this multiformalism setting and present encouraging preliminary results from an edge-computing case study.

This work is based on a recently proposed approach to precisely compute Markov chain transition rates from Maude's labeled transition systems.

Keywords: Maude · Multiformalism models · Reconfigurable distributed systems.

1 Introduction

Maude [15] is a purely declarative, high-performance, and expressive language with rewriting logic semantics [6]. A Maude system module is an executable specification of a distributed system. The Maude runtime support has various built-in model checking facilities. Maude has also been used as a logical framework for other formalisms, such as Petri Nets (PN), which lack native support for intuitively modeling adaptable systems.

In this paper, we present a quantitative analysis methodology for realistic adaptive systems that unifies two existing approaches. The first derives quotient transition systems for lumped Markov processes in Rewritable (Stochastic) Petri nets (RwSPNs) via compositional algebraic operators and structured node labeling, capturing symmetries as graph morphisms [9,13,11]. The second uses Maude as a multiformalism environment to represent complex systems through multiple coexisting formalisms in a unified framework [3]. We extend the modular RwSPN approach to this multiformalism setting by modeling interconnected nodes in a heterogeneous network as instances of generic "monoid" structures with symmetric compositional operators, generalizing those in [13].

The method can be seamlessly integrated with a recent approach [10] that precisely determines the state-transition rates of the continuous-time Markov chain (CTMC) induced by the labeled transition system (TS) generated from an executable specification in Maude with stochastic parameters. Exploiting symmetries yields a TS quotient whose strong bisimulation guaranties the *exact lumpability* of the corresponding CTMC.

Related Works Several approaches to timed and probabilistic analysis are available in Maude. [24] provides a survey, although it is no longer current. The framework proposed in [33] supports deterministic time specifications for the analysis of real-time systems. The method introduced in [1], which relies on probabilistic rewrite theories associated with actors, supports probabilistic discrete-event simulation. More recently, [29] proposed an extensive approach for applying Maude to stochastic analysis through a probabilistic extension of its strategy language. The rewriting-logic semantics associates a labeled transition system with ground terms; each state transition may involve many rewrites. The method in [10] builds a Markov process directly from Maude specifications with stochastic parameters, using object-level "meta-operators" to enumerate rule matches and compute transition rates. Extending this to a multiformalism framework improves the modeling of realistic systems. The use of behavioral equivalences within the Maude framework is well established (see, e.g., [22]). Our method, however, differs in that it focuses on identifying symmetries that mirror the modular structure of models constructed and manipulated via compositional operators.

The paper first reviews the compositional method of [13,11] for constructing a lumped CTMC from rewritable (stochastic) PNs. It then extends this technique to a multiformalism framework whose interconnected nodes are monoid structures sharing a distributed state. Preliminary results are presented on an edge-computing case study; a conclusion outlines contributions and future work. For the sake of completeness, several code snippets are included throughout the work. It is assumed that the reader already has knowledge of Maude syntax and the basic notions of rewriting logic. An intuitive presentation of the PT formalism [28] is provided in Section 2.

2 Modular rewritable (stochastic) Petri nets

In this section, we summarize the approach introduced in [13,11] and extend the underlying PN class to fit the requirements of the case study (Section 4). We extensively use *parameterized* modules, in which *theories* specify interfaces via syntactic and semantic constraints on parameter modules. Views (which can be parameterized in turn) link theories to the target modules by mapping their sorts and operators.

A rewritable Place-Transition net (RwPT) [27,26,13] is an algebraic formalism for mutable systems that combines the firing rule with structural rewriting. We consider its stochastic variant (RwSPN), introduced in [11] and extended in [10], where PT nets are enriched with bounded places and flush arcs; the encoding in Maude is available in a small module hierarchy in https://github.com/lgcapra/rewpt.

PT nodes are defined by the modules `PLACE{PL:: TRIV}` and `TRAN{L:: TRIV}`, whose parameters are node labels (the theory `TRIV` only declares the formal sort `Elt`); this flexible labeling is central. Multisets of places (module `PBAG{PL:: TRIV}`) are weighted sums: for instance, `3 . p(1) + 1 . p(2)` of sort `Pbag` (from `PBAG{Nat}`) denotes three copies of p_1 and one of p_2. The associative-commutative sum `_+_` (identity `nilP`) is a constructor. The places may be bounded; e.g., `p("a", 3)` (from `PLACE{String}`) has capacity 3, while `p("a")` is unbounded. The following excerpts introduce the PT signature, parameterized by node labels. This construction is based on the predefined `MAP` module via the (parameterized) `Ematrix` view.

```
fmod PT-NET{L :: TRIV, PL :: TRIV} is
 generated-by MAP{L, Ematrix{PL}} * (sort Map{L, Ematrix{PL}} to Net,
    sort Entry{L, Ematrix{PL}} to Tran, op emptyM to emptyNet) .
 var L : L$Elt . var T : Tran . vars N N' : Net . var P : Place .
 var M : Pbag . var K : NzNat . var EQ : Ematrix . var Q : Tmatrix . var S : Pset . var F :
      FlushArcs .
 op l : Tran -> L$Elt . op q : Tran -> Ematrix .
 op m : Tran -> Tmatrix . op f : Tran -> FlushArcs .
 ops I O H : Tran -> Pbag .
 eq l(L |-> EQ) = L .
 eq I(T) = I(m(T)) .
 ...
 eq L |-> [nilP, nilP] = emptyNet . *** removes isolated transitions
 op enabled : Tran Pbag -> [Bool] . *** doesn't consider place capacity
 eq enabled(L |-> EQ, M) = enabled(EQ, M) .
 op firing : Tran Pbag -> Pbag .
 eq firing(L |-> EQ, M) = firing(EQ, M) .
endfm

fmod PT-SYS{L :: TRIV, PL :: TRIV} is
  protecting PT-NET{L, PL} .
  sort System .
  var N : Net . var M : Pbag .
  op __ : Net Pbag -> System [ctor] .
  op n : System -> Net . op m : System -> Pbag .
  ...
endfm
```

Transitions (terms of sort `Tran`) are designated by labels linked to local adjacency lists, which by default are `Pbag` triples in `[]` (terms of sort `Tmatrix`). A typical transition is `t(L) |-> [I,O,H]` where `L` is the label and `I,O,H` are the pre-, post-, and optional inhibitor edges. In an `Ematrix` term (`Tmatrix < Ematrix`) of the form `[_,_,_] f: F`, the adjacency lists are followed by flush arcs (subterm `F`; see the example below).

PT nets are defined modularly using AC juxtaposition `_;_ : Net Net -> Net` (identity `emptyNet`) and the subsort relation `Tran < Net`, both inherited (after renaming) from the `MAP` module.[1] The module `PT-SYS` extends the PT signature with a marking: a term `System` is the juxtaposition of a `Net` and a `Pbag`. Separating PT nets from PT systems improves reuse. The operators for transition enabling and firing redirect to the homonymous ones in `EMATRIX{PL :: TRIV}` (excerpt below). The enabling operator is partial: it applies only to well-formed terms where a place cannot be both ordinary and flush input (or output), and every flush output place has a corresponding input.

```
var E : Ematrix . vars Q Q' : Tmatrix . var F : FlushArcs . var M : Pbag .
op enabled : Ematrix Pbag -> [Bool] . *** partial function
ceq enabled(Q f: F, M) = I(Q) <= M and-then H(Q) > M and-then enabled(F, M)
    if well-def(Q f: F) .
op firing : Ematrix Pbag -> Pbag .
ceq firing(Q f: F, M) = M + O(Q) + O(Q') - I(Q) - I(Q') if Q' := flushInOut(F, M) .
```

[1] Importing a module in a *generated-by* mode means that no additional data values are created.

The SPN signature (SPN-SIG) integrates stochastic parameters into PT-NET: Transition labels include a String (tag), a Float (rate parameter), and a Nat (firing policy).

```
fmod SPN-SIG{PL :: TRIV} is
  protecting PT-NET{Tlab{String}, PL} . protecting CONVERSION .
  var M : Pbag . var T : Tran .
  op tag : Tran -> L$Elt . op rate : Tran -> Float . op pol : Tran -> Nat .
  eq tag(T) = tag(l(T)) .
  op firingRate : Tran Pbag -> [Float] .
  eq firingRate(T, M) = if pol(T) == 1 then rate(T) else rate(T) * float(if pol(T) == 0
       then enabDegree(I(T), M) else min(pol(T), enabDegree(I(T), M)) fi) fi .
  ...
endfm

fmod SPN-SYS{PL :: TRIV} is *** marked SPN
  protecting SPN-SIG{PL} . protecting PT-SYS{Tlab{String}, PL} .
endfm
```

For example, this term from SPN-SIG{Nat}

```
t("a",1.5,0) |-> [2 . p(2), 1 . p(2), 2 . p(1)] f: p(3) >> p(4,7)
```

describes a transition with tag "a", rate parameter $\mu = 1.5$, and firing policy 0. This transition requires at least two tokens in p_2, at least one in p_3, and at most one in p_1. In addition, it cannot exceed the capacity of $p_{(4,7)}$ (7). When it fires, it removes one token from p_2 and all tokens from p_3, moving the latter to $p_{(4,7)}$. The operator firingRate defines state-dependent rates based on the *enabling degree* (*ed*), i.e., the number of instances that can fire simultaneously. Under the infinite-server policy (0), the firing rate is $ed \cdot \mu$; under the k-server policy, with $k > 0$, it is $\min(ed, k) \cdot \mu$.

The system module SPN-EMU extends the stateful PT signature with the rule firing, which defines the marking change and the transition rate while considering place capacities. The seemingly redundant free variable R (bound to firing(T, M) via a matching equation) is used in [10] to identify all significant rule's matches.

```
mod SPN-EMU{PL :: TRIV} is
 including SPN-SYS{PL} .
 vars M M' : Pbag . var T : Tran . vars N N' : Net .
 crl [firing] : N M => N M' if T ; N' := N /\ enabled(T, M) /\
  M' := firing(T, M) /\ not(overCap(M')) /\ R:Float := firingRate(T, M) .
endm
```

An RwSPN is characterized by a module M comprising SPN-EMU and two constants, net0 and m0, for the underlying PT net and its initial marking; M may also specify other rules that modify the PT net structure.

2.1 Exploiting symmetries in RwSPN

The methodology in [13] extends rewritable PT nets with process-algebra-like compositional operators, simplifying modeling and enabling symmetry-based reduction for large nested models. A compact *quotient* transition system (TS) is then built through structured node labeling; it is strongly bisimilar to the standard TS and yields *exact lumpability* of the associated CTMC [7]. We summarize these ideas here.

1. A compact set of *net operators* supports modular construction of any `System` (`Net`) component. These operators implicitly encode symmetries in nested components through incrementally built place labels—lists of pairs `< String ; Nat >` defined in the `SLAB` module. The SPN signature is parameterized using this labeling scheme via the standard instantiation `SPN-SIG{Slab}`.
2. Symmetry reduces to a bipartite graph *morphism* ($\cong$) that preserves transition labels and markings. A morphism from a net to itself is an *automorphism*. The place labels encode the hierarchy of replica components. By convention, the hierarchy root is the label suffix: in a *well-labeled* `Net`, any two distinct maximal sets of places whose labels share the same (possibly empty) suffix, preceded by identical string pairs, are automorphic. For example:
   `{p(..< "S1" ; 1 >)}`$\cong$`{p(..< "S1" ; 2 >)}` (i)
   `{p(..< "S2" ; 1 > < "S1" ; 1 >)}`$\cong$`{p(..< "S2" ; 3 > < "S1" ; 1 >)}` (ii)
   In case (i), the two automorphic sets of places are associated with distinct instances of the top-level subsystem `"S1"`; in case (ii), they are associated with analogous components within a single `"S1"` subsystem.
3. Well-labeled `System` terms can be efficiently normalized by minimizing the `Pbag` subterm via index permutation on place labels (without backtracking), combined with name (index) abstraction, which makes label indices consecutive integers from zero at each nesting level. This normal form is the *least* with respect to a total order.
4. PT systems built via compositional operators are well-labeled by construction. Rewriting rules must preserve this property; hence, they are given in a "parametric" style and rely on suitable net-manipulation operators provided by the library [13].

Technically, we must only encapsulate the right-hand side of each rule (assumed to be of `System` type) in a suitable *normalization* operator. The strong bisimulation of the TS quotient generated by a normalized term (`net0 m0`) coincides with "exact lumpability"[2] in the associated CTMC [11], ensuring that states within an aggregate have equal probabilities. Correctly computing cumulative transition rates in the lumped CTMC requires handling the fact that a normalized term $\hat{s}$ can transition to multiple states $s', s'', \ldots$ with the same normal form $\hat{s}'$, which the standard TS generated by Maude cannot detect (notably when a rule has several matches). [10] accurately computes these rates through automated translation of the rules into nondeterministic operators.

We evaluated our methodology in a gracefully degrading production system [13] with N parallel replicas of a production line (PL), each with K mutually interchangeable components subject to controlled degradation. Upon faults, each PL autonomously reconfigures to remain partially operational (see [13,11]). The initial configuration is built by applying the core operator `repl&share`, which creates replicas that may share some places, nested. (https://github.com/lgcapra/rewpt/blob/main/modSPT/FTPL.maude)

In the following excerpt, `cycle` and `fault` denote the subnets that represent the basic production cycle and the fault process, respectively.

```
eq PL(K) = repl&share(cycle ; fault, K, "L", p (< "o" ; 0 >) U p(< "s" ; 0 >)) .
eq NPL(N, K) = repl&share(PL(K), N, "PL", p(< "s" ; 0 >)) .
```

[2] An equivalence relation in this CTMC is exactly lumpable [7] if, for each pair of equivalence classes, the sums of the transition rates from any state in one class to each state in the other (or the same) class are equal.

N	# states (quotient)	time (sec)	# states (conventional)	time (sec)	$R(10.000)$
2	451	0	1915	0	0.68
4	2123	3	124,148	21	0.79
6	4855	10	4,587,426	1530	0.86
8	8643	29	-	†	0.89
10	13487	66	-	†	0.90

Table 1: Conventional vs quotient TS – $R(t) := P(\text{TimeToAbsorption}) > t$ – † time out

Table 1 reports the results for this benchmark as the number of PL replicas grows, the system reliability at a fixed time instant is included (the CTMC has absorbing states). Without symmetry exploitation, the analytical solution is only possible for small setups.

3 Efficient multiformalism modeling

Modeling realistic systems with a single formalism is often inconvenient or infeasible. Multiformalism modeling [18,30,5,14,31,21] addresses this by using the most suitable formalism for each component while preserving coherence and exploiting modularity and formalism-specific features (e.g., *multisolution*). OsMoSys [17] and SIMTHESys [20,4] exemplify this approach: OsMoSys orchestrates workflows [25] that invoke external tools for different formalisms, while SIMTHESys defines custom formalisms via formalism-specific behavioral elements that generate dedicated evaluation tools.

Current multiformalism techniques offer limited support for dynamic modification of system components. The use of rewriting systems (especially Maude) in this context was first explored in [3,12]. We integrate the Maude-based multiformalism framework [3,12] by extending the compositional approach for rewritable SPNs [11], summarized in the previous section, to handle the analytical complexity of reconfigurable multiformalism models. In Section 4, we demonstrate the method in a case study.

3.1 A Maude framework for reconfigurable multiformalism

The multiformalism encoding in `Maude` relies on a compact, extensible module hierarchy, using advanced module operations to maximize the reuse of existing formalism encodings. We first present the generic core of the framework (later extended to support the compositional methodology based on structured labeling), then we present the components of the case study, one of which in detail. A complete account is available in the online repository at https://github.com/lgcapra/rewpt/tree/main/multiformalism.

Multiformalism framework The framework improves [3] by better describing the dynamics of nodes in heterogeneous networks, easing modeling, and allowing the focus to be on structural adaptation. A multiformalism model integrates various components (e.g., Petri nets, queueing networks, fault trees) that share a distributed state. This state is formalized by the theory of commutative monoids (`C-MONOID`), which defines a sort `Elt` and an AC juxtaposition operation, and serves as the parameter of the functional module `NETWORK{S::C-MONOID}` that provides the ADT of a multiformalism model.

The abstract syntax uses sort `Network`, its subsort `Node`, and an AC juxtaposition operator on `Networks`, written `_,_`, with identity `emptyNetW`. Thus, a multiformalism

model is a multiset (commutative monoid) of possibly heterogeneous `Network` components (nodes). Declaring suitable subsorts for node types yields a model of distinct components interacting through shared state. A term `NetSys` consists of a `Network` and a subterm of formal sort `S$Elt`, representing a *network* of interconnected nodes with a distributed *state*. Intuitively, a network of nodes n_i with shared state S is

```
n1, n2,..., nk : S
```

In our example, network nodes use a state representation analogous to the *marking* in PT nets, i.e., a multiset of places compactly encoded using the `PBAG{PL :: TRIV}` module. The AC sum `_+_` matches the expected syntax of `C-MONOID`. The module's parameter denotes the place label; this flexibility in choosing place labels is essential.

Using the parameterized view `S-Pag{PL::TRIV}`, we partially instantiate the parameter of `NETWORK{S::C-MONOID}` as a multiset of places (integer state variables), yielding `NETWORK-M{PL::TRIV}`. Varying the view gives alternative state notions, and its parameter fixes the label types later attached to places.

```
fth C-MONOID is
   sort Elt .
   op 0 : -> Elt .
   op _+_ : Elt Elt -> Elt [assoc comm id: 0] .
endfth

fmod NETWORK{S :: C-MONOID} is
 protecting EXT-BOOL .
 sorts Node Network NetSys .
 subsort Node < Network .
 op emptyNetW : -> Network [ctor] .
 op _,_ : Network Network -> Network [ctor assoc comm prec 123 id: emptyNetW] .
 op _:_ : Network S$Elt -> NetSys [ctor prec 125] .
 op netw : NetSys -> Network .
 op state : NetSys -> S$Elt .
 vars N N' : Network . var M : S$Elt .
 eq netw(N : M) = N .
 eq state(N : M) = M .
 op in : Network Network -> Bool .
 eq in((N, N'), N) = true .
 eq in(N, N') = false [owise] .
endfm

view S-Pbag{PL :: TRIV} from C-MONOID to PBAG{PL} is
 sort Elt to Pbag .
 op 0 to nilP .
endv

fmod NETWORK-M{PL :: TRIV} is
   protecting NETWORK{S-Pbag{PL}} .
endfm
```

Components used in the case-study Three node classes are considered: SPN extended with bounded places and flush I/O edges; (multi-class) Queueing Networks (MQNs),

used as reservoirs of elementary queues; and Join-the-Shortest-Queue (JSQ) nodes. The Markov-modulating process is modeled as a special SPN.

E-SPN We use the signature introduced in Section 2. The Buffer (Fig. 3a) and the modulating CTMC (Fig. 2) are modeled as SPNs, the latter with transition adjacency matrices `[1 . P, 1 . P' + 1 . s-t, 1 . s-t]`. Here, `s-t` is a special place shared by all SPN transitions, which is both an output and an inhibitor: when `s-t` is not marked, the transition from state `P` to `P'` pauses the CTMC evolution until the (quasi) instantaneous reconfiguration is complete.

Queue Networks We reuse the multi-class queue networks (MQNs) signature (not reported here). MQNs are sequences of `Server` elements (module `SERVER{PL::TRIV}`), written `P @ F` for a `Place` and exponential service rate `Float`. The MQN signature (`QUEUE{PL::TRIV}`) is based on `LIST{X::TRIV}`. A simple MQN (sort `SimpleQ`) is a non-empty list of servers (`NeList{Server}`) ending in a `Place`, written `NeL > P`. Elementary queues (sort `ElQueue`) contain a single server, and in general an MQN (sort `Queue`) is a `SimpleQ` with two `Pbag` terms (in `[_,_]`) for the input and inhibitor enabling conditions, i.e. `ElQueue < SimpleQ < Queue`. The operator `queueRate` returns the service rate at each MQN stage as a function of the global population. Within a `Network`, MQNs are composed by juxtaposition (`_,_`) and shared places. In the edge-server example, we build pools of homogeneous elementary queues for the CPU (Fig. 3b)) and GPU (Fig. 3d)) under different configurations. The Network component (Fig. 3c)) has two queues, and the Storage (Fig. 3e)) has one.

JSQ (Join the Shortest Queue) This component plays a pivotal role in the allocation of jobs to a group of queues according to the eponymous algorithm. The JSQ signature is provided below. A term of type `Jsq` is specified by an input place and a (possibly empty) set of output places that are entry points for its queues, written `P |> S`, where `S` is a set of places. The `jsqRate` operator defines state-dependent scheduling times.

```
fmod JSQ{PL :: TRIV} is
  protecting PSET{PL} . protecting FLOAT .
  var P : Bplace . var S : Pset . var L : PL$Elt . var I : Nat .
  sort Jsq .
  op _|>_ : Bplace Pset -> [Jsq] [ctor] . *** partial function ...
 cmb p(L, 1) |> S : Jsq if p(L, 1) in S = false . *** JSQs input places must have capacity 1
  op In : Jsq -> Bplace .
  eq In(P |> S) = P .
  op Out : Jsq -> Pset .
  eq Out(P |> S) = S .
  op newJsq : Bplace -> [Jsq] .
  eq newJsq(P) = P |> emptyPset .
  op newJsq : PL$Elt -> Jsq .
  eq newJsq(L) = p(L, 1) |> emptyPset .
  op jsqRate : Jsq Pbag -> Float .
  ...
endfm
```

3.2 Efficient analysis of multiformalism models

Multiformalism models of realistic scenarios often become too complex for analytical solutions. We address this intractability by extending the symmetry-based methodology for RwPNs [13] to the just introduced multiformalism framework. This method should be combined with other techniques—such as abstractions, stochastic approximations (e.g., PN transition aggregations), and hierarchical multi-level analysis—but it remains a necessary step for analytical or simulation-based approaches.

As before, we extend existing hierarchies, prioritizing reuse. We present a modular architecture parallel to Section 3.1 that incrementally enriches network-node labels through compositional operators. We first outline the generic architecture and then instantiate it for JSQ; the same pattern applies to each class of interconnected nodes.

The approach is based on the flexible labeling scheme of Section 3.1. The module `NETWORK{S :: C-MONOID}` is first partially instantiated in `NETWORK-M{PL :: TRIV}` via the view `S-Pbag`. A further instantiation with `Slab`, mapping `TRIV` to `SLAB`, yields structured labels such as pairs lists `< String ; Nat >`. The resulting module expression `NETWORK-M{Slab}` defines a network of components with state distributed over places, where the places are annotated with these structured labels.

Compositional multiformalism models: core generic part This part contains parameterized modules and views. Models a generic heterogeneous network as an *extended* monoid whose elements (nodes) support a small set of primitive operators to efficiently manipulate structured labels. This is captured by the theory `MO-NET`, based on `MONOID+`, a monoid with a subsort `El < Elt`, where `El` denotes base elements. The module `MO-NET-OP`, parameterized by `MO-NET`, defines a concise set of generic operations for manipulating and normalizing any monoid-based model with node-structured labels, systematically built on the node-level operators of `MO-NET`.

Some operations (e.g. `replaceWith`, `places`) support the modular normalization algorithm for network terms, while others (e.g. `addLab`) are used by the core compositional operator `repl&share`. For example, the excerpt from `MO-NET-OP` shows how `replaceWith`—which performs index substitution throughout the entire monoid-like network—is defined in terms of `replaceWithEl`, which is formally specified in `MO-NET` and operates at the level of individual nodes. Network elements are essentially collections of places, and a place-based notion of state is used, but the method easily generalizes to arbitrary structures and state notions.

The theory `MO-NET-SYS` extends `MO-NET` to describe heterogeneous networks of nodes sharing a distributed state (a multiset of places) via the juxtaposition of subterms, forming the sort `MoNetSys`. It serves as the formal parameter of `MO-NET-SYS-OP`, which extends `MO-NET-OP` with primitive operations to efficiently manipulate such networks. Normalization and name-abstraction algorithms lift directly, and additional mechanisms are provided, such as purging a state (removing non-encoded places) and associating a state with a network subterm. Advanced module operations include the view homonym of `MO-NET-SYS`, embedding `MO-NET` and thereby enabling `MO-NET-OP` (which requires a `MO-NET` parameter) to be imported into `MO-NET-SYS-OP`. We summarize the main features of this methodology. Some code excerpts are then reported to illustrate them.

1. Any subsystem that admits a structural characterization as a monoid can be specified systematically by encapsulating the `MO-NET-OP` module—or, when a state is present, the `MO-NET-SYS-OP` module—according to a well-established and simple schema.
2. This property holds uniformly at all abstraction levels: globally (network-wide), for individual components (e.g., a Stochastic Petri Net can be formalized via this schema) and within components, whenever suitable monoid substructures exist. This enables a simple, modular definition of hierarchies of cooperating components that undergo uniform, consistent normalization.
3. Foundational data structures such as sets, multisets, and lists—common in modeling—can be treated as monoids and represented in this schema, allowing homomorphic operations on the structured labels of network node places. For example, module `PBAG-MOD` (not reported here), which defines a multiset of places with structured labels, encapsulates `MO-NET-OP` and is included in theory `MO-NET-SYS` (and thus in `MO-NET-SYS-OP`) and in the SPN specification `SPN-SYS-MOD`. Similarly, module `PSET-MOD` defines sets of places, is based on `MO-NET-OP`, and is reused throughout the framework.
4. As noted previously, we use the same essentially normalization strategy as in [13,11] (Section 2.1). This converts a `N$MoNetSys` term into a minimal form determined by a total lexicographic order. It combines name (index) abstraction in basic network components (`abstract` in `MO-NET-OP` and `MO-NET-SYS-OP`) with a state-minimization algorithm based on simple index permutation without backtracking (operator `minimize` in `PBAG-MOD`). Its efficiency relies on well-structured labels generated by compositional operators (`repl&share` in `MO-NET-OP`), a property that must be preserved by all rewrite rules. Unlike [13], this method is fully general: it applies recursively to any monoid-like structure, encapsulating `MO-NET-SYS-OP` or `MO-NET-OP` depending on whether a component has a distributed or local state.
5. The methodology in [13] (Section 2.1) uses a hierarchical symmetry notion that, for PT nets, coincides with bipartite (sub)graph morphisms. In a multiformalism setting, this must be refined for heterogeneous node structures: in `JSQ` nodes, symmetry is given by morphisms on (sub)sets, while in `QUEUE` nodes it uses morphisms on (sub)lists. Intuitively, we say

   `n1, n2,..., nk : S` $\cong$ `n1', n2',..., nk' : S'`

   iff there exists a family of morphisms $\phi = \{\phi_i \mid i{:}\ldots k\}$ such that $\phi_i(n_i) = n'_i$ and $\phi(S) = S'$ (the ordering of the nodes n_i is inconsequential). We call ϕ an automorphism if each ϕ_i is an automorphism.
6. Labels on structural elements forming the nodes (places here, though other types are possible) explicitly capture the model hierarchy and implicitly encode its symmetries. In a *well-labeled* `Network`, any two distinct maximal sets of elements whose labels share the same suffix (possibly empty) and are preceded by the same sequence of label pairs are related by an automorphism. We assume compositional operators enforce well-labeling, and rewriting rules preserve it. Thus, the TS quotient induced by a stateful network term `n1, n2, ..., nk : S` is exactly lumpable.

Presented below are several excerpts from the general framework that offers a monoid-style characterization of heterogeneous network components (as well as data structures).

```
fth MONOID+ is
 sorts El Elt . subsort El < Elt .
 op nil : -> Elt .
 op _;_ : Elt Elt -> Elt [assoc id: nil] .
endfth

fth MO-NET is
  including MONOID+ . including PSET-LAB .
  op replaceWithEl : El NeLab Nat -> El .
  op addLabEl : El Lab Pset -> El .
  op placesEl : El List{String} -> Pset .
  op placesEl : El Lab -> Pset .
endfth

fmod MO-NET-OP{N :: MO-NET} is
 var J : Nat . vars K K' : NzNat . vars N N' : N$Elt . var E : N$El .
 vars L L' : Lab . var NeL : NeLab . vars W W' : String . var P : Place .
 vars S S' : Pset . var WL : List{String} . var NeWL : NeList{String} .
 op replaceWith : N$Elt NeLab Nat -> N$Elt .
 eq replaceWith(N, NeL, J) = $replaceWith(N, NeL, J, nil) .
 op $replaceWith : N$Elt NeLab Nat N$Elt -> N$Elt .
 eq $replaceWith(nil, NeL, J, N) = N .
 eq $replaceWith(E ; N,NeL,J,N') = $replaceWith(N,NeL,J, N' ; replaceWithEl(E,NeL,J)) .
 op addLab : N$Elt Lab Pset -> N$Elt .
 op places : N$Elt List{String} -> Pset .
 op places : N$Elt NeLab -> Pset .
 ...
 op in : N$Elt NeLab -> Bool . *** looks for a place with a label prefix
 ceq in(E ; N , NeL) = true if placesEl(E, NeL) =/= emptyPset .
 eq in(N, NeL) = false [owise] .
 op abstract : N$Elt -> N$Elt . **** "name" (index) abstraction
 ceq abstract(E ; N) = abstract(replaceWithEl(E, < W ; K > L, J) ;
  replaceWith(N , < W ; K > L, J)) if P U S := placesEl(E) /\
  L' < W ; K > L := lab(P) /\ J := sd(K,1) /\ not(in(N,< W ; J > L)) .
 eq abstract(N) = N [owise] .
 op repl&share : N$Elt NzNat String Pset -> N$Elt . *** replica of a node
 eq repl&share(N, K, W, S) = $repl&share(N, K, W, nil, S) .
 op $repl&share : N$Elt Nat String N$Elt Pset -> N$Elt .
 eq $repl&share(N, 0, W, N', S) = N' .
 ceq $repl&share(N,K,W,N',S) = $repl&share(N,J,W,(N' ; addLab(N,< W ; J >,
     S)), S) if J := sd(K,1) .
 op replica : N$Elt NzNat String -> N$Elt . *** default
 eq replica(N, K, W) = repl&share(N, K, W, emptyPset) .
endfm

fth MO-NET-SYS is
 including MO-NET . including PBAG-MOD .
 sort MoNetSys . *** stateful network
 op _:_ : Elt Pbag -> MoNetSys .
endfth
```

```
view MO-NET-SYS from MO-NET to MO-NET-SYS is endv *** target is a theory

fmod MO-NET-SYS-OP{N :: MO-NET-SYS} is
 protecting MO-NET-OP{MO-NET-SYS}{N} . protecting PSET-MOD .
 var J : Nat . var K : NzNat . var M : Pbag . var P : Place .
 var W : String . var L : Lab . var NeL : NeLab . var S : Pset .
 vars Sys Sys' : N$MoNetSys . var N : N$Elt .
 op nd : N$MoNetSys -> N$Elt . *** node structure
 op st : N$MoNetSys -> Pbag . *** node shared state
 eq nd(N : M) = N .
 eq st(N : M) = M .
 op replaceWith : N$MoNetSys NeLab Nat -> N$MoNetSys .
 eq replaceWith(N : M, NeL, J) = replaceWith(N, NeL, J) : replaceWith(M,
     NeL, J) .
 op abstract : N$MoNetSys -> N$MoNetSys .
 ceq abstract(N : M) = abstract(replaceWith(N : M, < W ; K > L, J))
    if P U S := places(N) /\ L' < W ; K > L := lab(P) /\ J := sd(K,1) /\
     not(in(S, < W ; J > L)) .
 eq abstract(Sys) = Sys [owise] .
 op normalize : N$MoNetSys -> N$MoNetSys .
 ceq normalize(Sys) = nd(Sys') : minimize(st(Sys'))
    if Sys' := abstract(Sys) .
 ...
endfm
```

3.3 Using the framework to exploit the symmetries of multiformalism models

We extend the multiformalism framework of Section 3.1 to exploit symmetries in monoid-like structures. We first consider the network level and then the component level.

The module NETWORK-LAB instantiates NETWORK-M through view Slab, giving structured labels to basic components, and imports PSET-MOD and PBAG-MOD, which define monoidal (multi)set structures of places with structured labels. It then provides an abstract specification of the operations required by theory MO-NET-SYS, reusing its operation names; these are later instantiated when concrete components are linked to the network. The view NETWORK-LAB maps MO-NET-SYS to the target module with default operator mappings, assigning (sub)sort El, the basic monoid elements, to Node, representing the network nodes. Finally, NETWORK-MOD encapsulates NETWORK-LAB within MO-NET-SYS-OP via the homonymous view, thus treating NETWORK-LAB as a monoid equipped with the operations needed for compositional modeling. Modules PBAG-MOD and PSET-MOD are instantiated from parameterized modules PBAG{PL::TRIV} and PSET{PL::TRIV}, respectively, and follow the same schema while wrapping the stateless monoid structure MO-NET-OP via views from theory MO-NET.

```
fmod NETWORK-LAB is
 protecting PBAG-MOD . protecting PSET-MOD . protecting NETWORK-M{Slab} .
 op replaceWithEl : Node NeLab Nat -> Node . *** formal operations
 op addLabEl : Node Lab Pset -> Node .
 op placesEl : Node List{String} -> Pset .
```

```
 op placesEl : Node Lab -> Pset .
endfm

view NETWORK-LAB from MO-NET-SYS to NETWORK-LAB is
 sort Elt to Network . sort El to Node . sort MoNetSys to NetSys .
 op nil to emptyNetW .
 op _;_ to _,_ .
endv

fmod NETWORK-MOD is
 protecting MO-NET-SYS-OP{NETWORK-LAB} .
endfm
```

Linking nodes to the `Network` A simple pattern connects any node type; we illustrate it with JSQ nodes. We define a functional module (`JSQ-NODE`) that bundles the node signature (`JSQ`) and the network ADT (`NETWORK-M`), instantiated by declaring the concrete node sort (`Jsq`) as a subsort of `Node`. For convenience, this module is wrapped in a system module (`JSQ-NODE-SYS`), which specifies the node's network behavior via a rewrite rule. To integrate heterogeneous components into an interacting network with shared distributed state and possible behavioral symmetries, we apply to each component a modeling pattern analogous to that used at the overall network level.

The next code excerpt applies this procedure to the JSQ component. From the parameterized module `JSQ`, we build a wrapper module `JSQ-MOD` that instantiates `JSQ` with structured place labels and imports `PSET-MOD`, since a JSQ is essentially a set of places. Another module then combines `JSQ-MOD` with `NETWORK-MOD`: the abstract operations of `NETWORK-MOD` (from theory `MO-NET`) receive concrete definitions for `Jsq` via subsort overloading from `Jsq < Node`, mapped directly to the corresponding operations in `PSET-MOD`. We apply the same pattern to Queue nodes, starting from `QUEUE`: a `Queue` is represented as a list of servers embedded in a monoid structure, analogous to the treatment of sets and multisets of places.

The SPN component uses a slightly modified version of `SPN-SIG`. An SPN is treated as a monoid (with `Tran` terms as elements and `_;_` as the associative operation) and embedded in `MO-NET-OP` via a suitable view, yielding a recursive structure where the heterogeneous network and some nodes are monoids. When SPN nodes must interconnect while keeping local state, we use the stateful monoid representation in `MO-NET-SYS-OP`.

```
fmod JSQ-MOD is
 protecting JSQ{Slab} . protecting PSET-MOD . *** set defined as a monoid
endfm

fmod JSQ-NODE-MOD is
  protecting JSQ-MOD . extending NETWORK-MOD .
  subsort Jsq < Node .
  var L : Lab . var NeL : NeLab . var P : Place . var S : Pset . var I : Nat .
  var Ls : List{String} .
  eq replaceWithEl(P |> S,NeL,I) = replaceWith(P,NeL,I) |> replaceWith(S,NeL, I) .
  eq addLabEl(P |> S, L, S) = addLab(P, L, S) |> addLab(S, L, S) .
  eq placesEl(P |> S, Ls) = places(S U P, Ls) .
```

```
  eq placesEl(P |> S, NeL) = places(S U P, NeL) .
  eq placesEl(P |> S) = S U P .
endfm

mod JSQ-NODE-SYS-MOD is
 including JSQ-NODE-MOD .
 var N : Network . var B : Pbag . var P : Bplace . var P' : Place .
 var NeS : NePset . var S : Pset .
 crl (N , P |> NeS) : 1 . P + B => (N , P |> NeS) : normalize(B + 1 . P')
    if P' U S := emin(B, NeS) /\ rate:Float := jsqRate(P |> NeS, B) .
endm
```

4 Case study: an edge server in a smart city

We consider a smart city case study, detailed and analyzed in [23]. Our goal is to assess the effectiveness of our symmetry-based approach. We present a small selection of quantitative results and refer to [23] for a more detailed analysis.

In smart cities, sensor data are continuously collected using ICT such as IoT and edge computing to monitor conditions and enable adaptive solutions for future traffic, behavior, events, transportation demand, and needs [2,19,16]. We consider an edge server that gathers sensor data and stores them on a blockchain [8], facing a variable workload as roadside sensors scale with vehicle density and cars move in and out of coverage. The server includes data acquisition, computing, networking, and temporary storage. A multi-core processor aggregates data into packets and writes them to the blockchain, pausing when the network is unavailable or power is low. If a watchdog is activated or storage exceeds a threshold, a backup radio is enabled to process and store buffered data. The edge server usually runs in energy-saving mode, with backup radio, storage, and extra CPU capacity off, scaling CPU utilization between 50% and 80%.

The nominal structure of the edge server is shown in Fig. 1, and the overall adaptive behavior of the system follows the state diagram in Fig. 2. The states L_0–L_3 represent increasing load levels, with transitions from L_i to L_j at rate q_{ij}, forming a canonical Markov-modulated process.

- L0: Buffer active (only here), 1 CPU core for computation+encryption, slow network (A) only. Buffer supports blockchain, which is more efficient on large datasets.
- L1: 3 CPU cores for computation+encryption, fast network (B) only.
- L2: 6 CPU cores, both networks (A+B), CPU only for computation, 1 GPU board for encryption.
- L3: all 8 CPU cores, both GPUs for encryption, both networks (A+B), and Storage.

Fig. 1 shows the functional blocks; Fig. 2 shows the allowed adaptations and active hardware. Fig. 3a) alternates two submodels: when the buffer is inactive, the lower submodel is omitted; otherwise, the upper PT models a buffer where packets in place n wait until packets accumulate K (enabling $t_{collect}$) or the timeout $T_{timeout}$ fires. The packets then move to the place m and to the CPU via $t_{continue}$; the dashed "flush" edges encode this. Fig. 3b) shows two related CPU submodels, the second parametric: the upper has a single queue CPU_1; the lower, parameterized by N, models $N > 1$ active cores with queues CPU_i ($1 \leq i \leq N$), with arrivals routed by JSQ. Fig. 3c) alternates

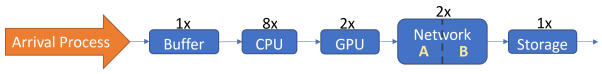
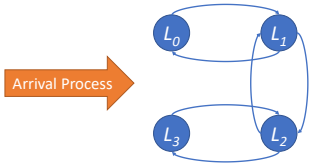


Fig. 1: The nominal model of the scenario considered.



	Buf	CPU	GPU	Net	Stor
L_0	1	1	0	A	0
L_1	0	3	0	B	0
L_2	0	6	1	A+B	0
L_3	0	8	2	A+B	1

Fig. 2: The MC-modulated arrival process and the corresponding configurations.

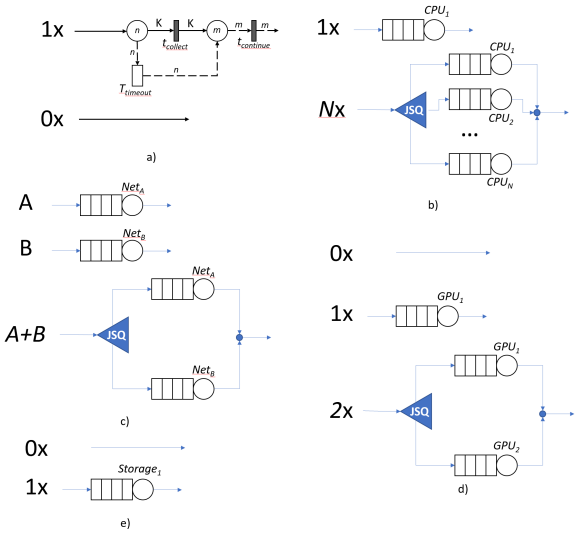


Fig. 3: a) the *Buffer*; b) the *CPU*; c) the *GPU*; d) the *Network*; e) the *Storage*.

two GPU submodels, mirroring the CPU case; when the GPU is used, the CPU demand decreases. Fig. 3d) has three operational submodels: only module A (queue Net_A), only B (queue Net_B) and both A and B (analogous to the 2-GPU case). Fig. 3e) offers two alternatives, with or without active storage: in the first, storage is absent; in the second, it is modeled by queue $Storage_1$.

4.1 Formalization and analysis of the case study

The case study is divided into several system modules. The core (listed below) includes three modules `<NTYPE>-NODE-SYS-MOD` that characterize the dynamics of each type of node present in the network, as explained. It also includes a fourth system module, `MC-EMU`, which formalizes the Markov modulating process as a specific stochastic Petri net (SPN). This module encapsulates `SPN-EMU` by redefining selected sorts and operators to avoid naming conflicts. Transitions are directly annotated with their rate parameters.

The CTMC component is intentionally left unconnected to `Network` because it acts as a meta-controller of the evolution of the network. (Maude provides meta-level modules, but we found it more convenient to represent this at the object level.)

```
mod MM-BLOCKCHAIN is
 including QUEUE-NODE-SYS-MOD . including SPN-NODE-SYS-MOD{String} .
 including JSQ-NODE-SYS-MOD . including MC-EMU .
 sort MMnetwork . *** markov modulated network
 op MM:_N:_ : System NetSys -> MMnetwork [ctor] .
 op mc : MMnetwork -> System .
 op sys : MMnetwork -> NetSys .
 var MC : System . var Sys : NetSys .
 eq mc (MM: MC N: Sys) = MC .
 eq sys(MM: MC N: Sys) = Sys .
endm
```

Through the constructor `op MM:_N:_ : System NetSys -> MMnetwork` we represent CTMC and network as distinct components, with a term `MMnetwork` denoting their amalgamation. The `System` subterm denotes a CTMC along with its initial state (given as a singleton multiset of places). The system module `MC-EMU` contains the SPN firing rule, which triggers the local rewrite encoding a state transition of the CTMC. Two additional modules complete the formalization of the case-study by defining compositional operators and rules for Markov-chain–driven network reconfiguration (details in https://github.com/lgcapra/rewpt/tree/main/multiformalism/MODmulti/BLOCKCHAIN-MOD.maude). In particular, the operators:

```
op pool : ElQueue NzNat -> Network
op connect&link : Jsq Network -> Network
```

build a pool of similar elementary queues with a single exit point and connect them to a JSQ. The first relies on the *symmetric* composition operator `repl&share`; the second is derived from the first. Such operators are crucial for scalability in both modeling and analysis. We conclude by listing a self-contained excerpt from the final system module (`BLOCKCHAIN-EXE`), showing the two rewriting rules that reconfigure the system between states L_0 and L_1 (Fig. 1). (Here, the transition rates are arbitrary; the precise stochastic parameters are in [23]).

```
op mmc : -> Q [memo] . *** modulating Markov chain
op L : Nat -> Place [memo] . *** states of the CTMC
eq L(i) = p(< "mc" ; i >) .
op s-t : -> Place [memo] .
eq s-t = p(< "s-t" ; 0 >) .
eq mmc = 0.01 |-> [1 . L(0); 1 . L(1) + 1 . s-t ; 1 . s-t] ;
0.02 |-> [1 . L(1); 1 . L(0) + 1 . s-t ; 1 . s-t] ;
0.02 |-> [1 . L(2); 1 . L(1) + 1 . s-t ; 1 . s-t] ; 0.01 |-> [1 . L(1); 1 . L(2) + 1 . s-t ; 1 . s-t] ;
0.02 |-> [1 . L(2); 1 . L(3) + 1 . s-t ; 1 . s-t] ; 0.01 |-> [1 . L(3); 1 . L(2) + 1 . s-t ; 1 . s-t] .
op N : -> NzNat [memo] . eq N = 4 . *** model's parameter (place capacity)
ops emptyp pin cpuIn cpuOut gpuIn gpuOut netOut a b : NzNat -> Place [memo] .
eq emptyp(k) = p(emptyLab, k) . *** bounded place with empty lab
eq pin(k) = replica(emptyp(k), 1, "in") . *** bounded input place (symmetric style)
eq cpuIn(k) = replica(emptyp(k), 1, "cpu") .
```

```
eq cpuOut(k) = replica(emptyp(k), 1, "cpu-Out") .
...
var I : Nat . vars K K' K'' : NzNat . vars P P' : Place . var R : Float .
var MC : System . var Q : Q . vars S B : Pbag . var Nw : Network .
op buffer : -> Net [memo] .
eq buffer = newTin(pin(N),2.0) ; t("collect",1.0,1) |-> [N . pin(N), N . p(< "b" ; 0 >, N)] ;
      t("timeout", 0.2, 1) |-> [nilP, nilP] f: pin(N) >> p(< "b" ; 0 >, N) ;
      t("continue", 1.0, 1) |-> [nilP, nilP] f: p(< "b" ; 0 >, N) >> cpuIn(N) .
ops A B : Place NzNat -> ElQueue [memo] . *** parametric in the input and in the capacity
eq A(P, K) = newElQueue(P, netOut(K), 2.0) .
eq B(P, K) = newElQueue(P, netOut(K), 4.0) .
ops cpu gpu : NzNat NzNat -> Network [memo] . *** 1st par: replica, 2nd: capacity
eq cpu(K, K') = pool(newElQueue(emptyp(N), cpuOut(K'), 5.0), K, "cpu") .
op network : Nat -> [Network] [memo] .
eq network(0) = buffer, cpu(1, N), A(cpuOut(N), N), newTout(netOut(N), 1.0) .
eq network(1) = tin, connect&link(newJsq(pin(1)), cpu(3, N)), B(cpuOut(N), N),
  newTout(netOut(N), 1.0) .
crl [S0>S1] : MM: (Q S) N: (Nw : B) => MM: (Q 1 . L(1)) N: normalize((
    network(1) : B)) if S == 1 . L(1) + 1 . s-t /\ in(Nw, buffer) .
crl [S1>S0] : MM: (Q S) N: (Nw : B) => MM: (Q 1 . L(0)) N:
   normalize((network(0) : set(B, pin(cpu(1, N)), I)))
  if S == 1 . L(0) + 1 . s-t /\ in(Nw, cpu(3, N)) /\
   I := allClients(cpu(3, N), B) /\ not(I <= cap(pin(cpu(1, N)))) .
...
op MMsys0 : -> MMnetwork . *** initial setup
eq MMsys0 = MM: (mmc 1 . L(0)) N: (network(0) : nilP) .
```

We use aliasing to improve readability (e.g., operators `buffer`, `A`, `B`, `cpu`, `gpu` map to components in Fig. 3, and `mmc` denotes the modulating CTMC in Fig. 2). Specifically, `cpu(K, K')` denotes a CPU pool defined by compositional symmetric operators, and `network(I)` represents the system configuration in state L_I. We also use the symmetric operator `replica` to build correctly labeled places from an initially empty one since the places (module `PSET-MOD`) form a monoid. The rewriting mechanism is activated exactly when the CTMC place `s-t` holds one token (indicating a CTMC state transition) and is deactivated as soon as this token is removed from `s-t`. During the transition from L_1 to L_0, the number of CPUs decreases from three to one. In this phase, the policy may delay job migration based on the processing capacity of the remaining core, slightly violating the assumption of instantaneous reconfiguration. Two additional pairs of rules, defined analogously, formalize the transitions $L_1 \leftrightarrow L_2$ and $L_2 \leftrightarrow L_3$ (see Figure 2).

To construct the TS quotient, we wrap the `NetSys` subterm of `MMnetwork` on the right-hand side of the rules in the `normalize` operator. Table 2 reports the performance of the Maude command `search mcStateTran(MMsys0) =>! M:McStateTran`, which identifies final states in the fully reconfigurable edge-server model (`MMsys0` is the initial setup). A single-parameter model is used, with N as the capacity of most places; the searches show the system is deadlock-free. Without symmetries, analysis is feasible only for small setups ($N \leq 2$); normalization adds acceptable overhead, and for $N = 16$ the TS is estimated to contain tens of billions of states. The model still generates large state spaces as N grows, because only CPUs (and to a lesser extent GPUs) are

significantly symmetric, yielding only partial symmetry. Thus, additional techniques such as abstraction and structural reductions must complement symmetry-based reductions.

We use a parameter $0 < \alpha < 1$ for the modulation process. With base switching time $T_{base} = 6$ hours, we set $q_{i,i+1} = 1/(\alpha T_{base})$ and $q_{i,i-1} = 1/((1-\alpha)T_{base})$. The solution is obtained through discrete-event simulation using the `R` solver [32] on the lumped CTMC generator produced by preprocessing in [10]. Simulation runs are on average an order of magnitude faster than the corresponding build times in Table 2. Fig. 4 shows the response time and monthly cost (secondary axis) using the arrival rates of [23]. The former is acceptable unless the system is too long in L_0, where the buffer is used; the monthly cost increases by nearly an order of magnitude when the system is longer in L_3.

	TS		Quotient TS	
N	states(final)	time (sec)	states(final)	time (sec)
2	5,789,456(0)	2445	37,344(0)	40
4	-	†	303,534(0)	367
8	-	†	2,990,842(0)	4200
16	-	†	18,189,496(0)	27070

Table 2: Build time of conventional TS vs normalized TS as the capacity varies)

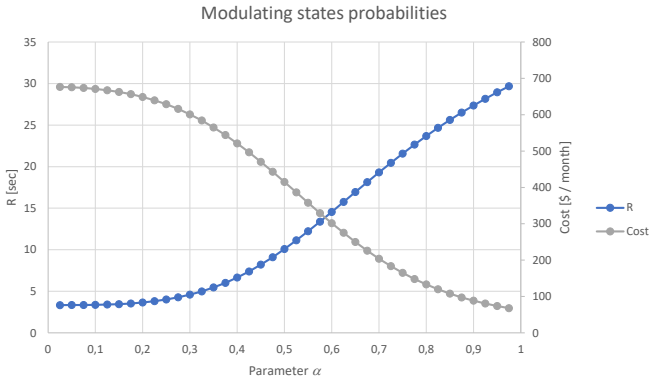


Fig. 4: Response times and costs for different values of the parameter α.

5 Conclusions

We present a unified technique for analyzing adaptive systems that combines two approaches. The first derives a lumped Markov process for RwSPNs using compositional operators that exploit symmetries via a structured node-labeling scheme reflecting model modularity. The second uses Maude as a multiformalism framework for models combining several formalisms. We extend compositional RwSPNs to this setting and integrate them into the preprocessing-based approach of [10] to derive a lumped CTMC, reporting promising initial results in an edge-computing case study. We plan to extend our methodology to encompass a larger variety of nodes—interpreted as fundamental building blocks of network components—as well as a broader spectrum of state concepts, including components with both private (hidden) and public (shared) state.

References

1. Agha, G., Meseguer, J., Sen, K.: Pmaude: Rewrite-based specification language for probabilistic object systems. Electronic Notes in Theoretical Computer Science **153**(2), 213–239 (2006). https://doi.org/https://doi.org/10.1016/j.entcs.2005.10.040, proceedings of the Third Workshop on Quantitative Aspects of Programming Languages (QAPL 2005)
2. Alshekhly, I.: Smart cities: Survey. J. Adv. Comput. Sci. Technol. Res **2**, 79–90 (2012)
3. Barbierato, E., Capra, L., Gribaudo, M., Iacono, M.: Enhancing multiformalism models with rewrite engines. Proceedings of EPEW 2025, June 26-27, 2025 Catania, Italy (2025)
4. Barbierato, E., Gribaudo, M., Iacono, M.: Modeling hybrid systems in SIMTHESys. Electronic Notes in Theoretical Computer Science **327**, 5 – 25 (2016). https://doi.org/10.1016/j.entcs.2016.09.021
5. Bause, F., Buchholz, P., Kemper, P.: A toolbox for functional and quantitative analysis of DEDS. In: Proc. of 10th Int. Conf. on Comp. Perf Ev.: Modelling Techniques and Tools. pp. 356–359. TOOLS '98, Springer-Verlag, UK (1998)
6. Bruni, R., Meseguer, J.: Generalized rewrite theories. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) Automata, Languages and Programming. pp. 252–266. Springer-Verlag, Berlin, Heidelberg (2003). https://doi.org/10.1007/3-540-45061-0_22
7. Buchholz, P.: Exact and ordinary lumpability in finite markov chains. Journal of Applied Probability **31**(1), 59–75 (1994). https://doi.org/https://doi.org/10.2307/3215235
8. Campanile, L., Iacono, M., Mastroianni, M., Riccio, C.: Performance evaluation of an edge-blockchain architecture for smart city. vol. 2025-June, p. 620 – 627 (2025). https://doi.org/10.7148/2025-0620
9. Capra, L.: A Maude implementation of rewritable Petri nets: a feasible model for dynamically reconfigurable systems. In: Gleirscher, M., Pol, J.v.d., Woodcock, J. (eds.) Proceedings of the First Workshop on Applicable Formal Methods, 2021. Electronic Proceedings in Theoretical Computer Science, vol. 349, pp. 31–49. Open Publishing Association (2021). https://doi.org/10.4204/EPTCS.349.3
10. Capra, L.: Associating a Markov process with Maude executable modules. In: Proceedings of the 15th International Conference on Simulation and Modeling Methodologies, Technologies and Applications. pp. 106–116. SciTePress (2025)
11. Capra, L., Gribaudo, M.: A lumped CTMC for modular rewritable PN. In: Doncel, J., Remke, A., Pompeo, D.D. (eds.) Computer Performance Engineering - 20th European Workshop, EPEW 2024, Venice, Italy, June 14, 2024. Lecture Notes in Computer Science, vol. 15454, pp. 106–120. Springer (2024)
12. Capra, L., Gribaudo, M., Iacono, M., Köhler-Bußmeier, M.: Using rewriting systems for performance analysis. QualITA 2025: The Fourth Conference on System and Service Quality, June 25 and 27, 2025, Catania, Italy, Proceedings (2025)
13. Capra, L., Köhler-Bußmeier, M.: Modular rewritable Petri nets: An efficient model for dynamic distributed systems. Theoretical Computer Science **990**, 114397 (2024). https://doi.org/10.1016/j.tcs.2024.114397
14. Ciardo, G., Miner, A.S.: SMART: The stochastic model checking analyzer for reliability and timing. In: Quantitative Evaluation of Systems, Int. Conf. on. pp. 338–339. IEEE, Los Alamitos, CA, USA (2004)
15. Clavel, M., Durán, F., Eker, S., Lincoln, P., Oliet, N.M., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic. Lecture Notes in Computer Science, Springer (July 2007). https://doi.org/10.1007/978-3-540-71999-1
16. Far, A.Z., Far, M.Z., Gharibzadeh, S., Zangeneh, S., Amini, L., Rahimi, M.: Artificial intelligence for secured information systems in smart cities: Collaborative iot computing with deep reinforcement learning and blockchain. arXiv preprint arXiv:2409.16444 (2024)

17. Franceschinis, G., Gribaudo, M., Iacono, M., Mazzocca, N., Vittorini, V.: Drawnet++: Model objects to support performance analysis and simulation of systems. Lecture Notes in Computer Science **2324**, 233 – 238 (2002)
18. Gribaudo, M., Iacono, M.: An introduction to multiformalism modeling (2013). https://doi.org/10.4018/978-1-4666-4659-9.ch001
19. Guo, S., Qi, Y., Yu, P., Shao, S., Qiu, X.: Edge network resource synergy for mobile blockchain in smart city. In: 2020 International Wireless Communications and Mobile Computing (IWCMC). pp. 1272–1277. IEEE (2020)
20. Iacono, M., Gribaudo, M.: Element based semantics in multi formalism performance models. p. 413 – 416 (2010). https://doi.org/10.1109/MASCOTS.2010.54
21. de Lara, J., Vangheluwe, H.: Atom3: A tool for multi-formalism and meta-modelling. In: FASE. LNCS, vol. 2306, pp. 174–188. Springer (2002)
22. Lluch Lafuente, A., Meseguer, J., Vandin, A.: State space c-reductions of concurrent systems in rewriting logic. In: Aoki, T., Taguchi, K. (eds.) Formal Methods and Software Engineering. pp. 430–446. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
23. Lorenzo, C., Marco, G., Mauro, I.: Reconfigurable stochastic multi-formalism models: an approach based on maude. In: Proença, J., Umberto, R. (eds.) Software Engineering and Formal Methods. SEFM 2025 Collocated Workshops (to appear in). Springer Nature Switzerland, Cham (2026)
24. Meseguer, J.: Twenty years of rewriting logic. The Journal of Logic and Algebraic Programming **81**(7), 721–781 (2012). https://doi.org/https://doi.org/10.1016/j.jlap.2012.06.003, rewriting Logic and its Applications
25. Moscato, F., Flammini, F., Di Lorenzo, G., Vittorini, V., Marrone, S., Iacono, M.: The software architecture of the OsMoSys Multisolution Framework (2007)
26. Padberg, J., Kahloul, L.: Overview of reconfigurable petri nets. In: Heckel, R., Taentzer, G. (eds.) Graph Transformation, Specifications, and Nets: In Memory of Hartmut Ehrig, pp. 201–222. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75396-6_11
27. Padberg, J., Schulz, A.: Model checking reconfigurable petri nets with maude. In: Echahed, R., Minas, M. (eds.) Graph Transformation. pp. 54–70. Springer (2016). https://doi.org/10.1007/978-3-319-40530-8_4
28. Reisig, W.: Petri Nets: An Introduction. Springer-Verlag New York, Inc., New York, NY, USA (1985). https://doi.org/10.1007/978-3-642-69968-9
29. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: Qmaude: Quantitative specification and verification in rewriting logic. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) Formal Methods. pp. 240–259. Springer International Publishing, Cham (2023)
30. Sanders, W.: Integrated frameworks for multi-level and multi-formalism modeling. In: Petri Nets and Performance Models, 1999. Proc. 8th Int. Work. on. pp. 2–9 (1999)
31. Trivedi, K.S.: Sharpe 2002: Symbolic hierarchical automated reliability and performance evaluator. In: DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks. p. 544. IEEE, Washington, DC, USA (2002)
32. Ucar, I., Smeets, B., Azcorra, A.: simmer: Discrete-event simulation for r. Journal of Statistical Software **90**(2), 1–30 (2019). https://doi.org/10.18637/jss.v090.i02
33. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theoretical Computer Science **285**(2), 359–405 (2002). https://doi.org/https://doi.org/10.1016/S0304-3975(01)00363-2, rewriting Logic and its Applications

Author Index