Santiago Escobar, Narciso Martí-Oliet (Eds.)

Rewriting Logic and its Applications

13th International Workshop, WRLA 2020

part of the European Joint Conferences on Theory & Practice of Software (ETAPS 2020)

Dublin, Ireland, April 25th & 26th, 2020.

Informal Proceedings

Preface

This volume contains the preliminary proceedings of the 13th International Workshop on Rewriting Logic and its Applications (WRLA 2020).

Rewriting logic is a natural model of computation and an expressive semantic framework for concurrency, parallelism, communication, and interaction. It can be used for specifying a wide range of systems and languages in various application fields. It also has good properties as a metalogical framework for representing logics. Over the years, several languages based on rewriting logic have been designed and implemented. The aim of the workshop is to bring together researchers with a common interest in rewriting logic and its applications, and to give them the opportunity to present their recent works, discuss future research directions, and exchange ideas. The previous meetings were held in Asilomar (USA) 1996, Pont-à-Mousson (France) 1998, Kanazawa (Japan) 2000, Pisa (Italy) 2002, Barcelona (Spain) 2004, Vienna (Austria) 2006, Budapest (Hungary) 2008, Paphos (Cyprus) 2010, Tallinn (Estonia) 2012, Grenoble (France) 2014, Eindhoven (Netherlands) 2016, and Thessaloniki (Greece) 2018.

WRLA 2020 should have been held on April 25-26, 2020, in Dublin, Ireland, as a satellite event of the European Joint Conferences on Theory & Practice of Software (ETAPS). Unfortunately, the COVID-19 pandemic reached us forcing the organizers first to postpone ETAPS 2020 and later to cancel it, similarly to other events around the world. Due to this, our preparations for having invited speakers and tutorials at the workshop were also cancelled. At the time of writing we do not know yet whether an alternative celebration of this workshop will take place or not. This will be discussed with the steering and the program committees and announced in due time.

We received 16 submissions; each one of them was reviewed by at least three program committee members. After an extensive discussion, the program committee decided to accept 15 papers for presentation at the workshop.

A selection of the papers accepted for presentation will appear in the proceedings that will be published in the Springer LNCS series, following the tradition of previous meetings in this series.

We sincerely thank all the authors of papers submitted to the workshop, as well as the members of the program committee and the referees for their careful work in the review process. This time we cannot thank the ETAPS organizers because we did not reach the celebration stage. Hopefully, the COVID-19 pandemic will soon be over and then we will be able to recover our meetings on rewriting logic and everything else.

June, 2020

Santiago Escobar
Narciso Martí-Oliet

Table of Contents

Program Committee

Erika Abraham	RWTH Aachen University
María Alpuente	Universitat Politècnica de València
Irina Mariuca Asavoae	Trusted Labs, Thales Group
Kyungmin Bae	Pohang University of Science and Technology
Clara Bertolissi	Université Aix-Marseille
Artur Boronat	University of Leicester
Roberto Bruni	Università di Pisa
Francisco Durán	University of Málaga
Santiago Escobar	Universitat Politècnica de València
Maribel Fernandez	KCL
Thomas Genet	IRISA - Rennes
Raúl Gutiérrez	Universitat Politècnica de València
Nao Hirokawa	JAIST
Alexander Knapp	Universität Augsburg
Alberto Lluch Lafuente	Technical University of Denmark
Dorel Lucanu	Alexandru Ioan Cuza University
Narciso Martí-Oliet	Universidad Complutense de Madrid
Aart Middeldorp	University of Innsbruck
Cesar Munoz	NASA
Vivek Nigam	fortiss GmbH
Kazuhiro Ogata	JAIST
Etienne Payet	LIM, Université de La Réunion
Adrian Riesco	Universidad Complutense de Madrid
Christophe Ringeissen	INRIA
Camilo Rocha	Pontificia Universidad Javeriana Cali
Grigore Roșu	University of Illinois at Urbana-Champaign
Vlad Rusu	INRIA
Ralf Sasse	ETH Zurich
Traian Florin Serbanuta	University of Bucharest
Carolyn Talcott	SRI International

Additional Reviewers

A

Abd Alrahman, Yehia
K

Kremer, Gereon
M

Marshall, Andrew M.
V

Van Oostrom, Vincent
W

Winkler, Sarah

Combining Parallel Graph Rewriting and Quotient Graphs

Thierry Boy de la Tour and Rachid Echahed

CNRS and University Grenoble Alpes, LIG Lab. Grenoble, France
thierry.boy-de-la-tour@imag.fr rachid.echahed@imag.fr

Abstract. We define two graph transformations, one by parallelizing graph rewrite rules, the other by taking quotients of graphs. The former consists in the exhaustive application of local transformations defined by graph rewrite rules expressed in a set-theoretic framework. Compared with other approaches to parallel rewriting, we allow a substantial amount of overlapping only restricted by a condition called the *effective deletion property*. This transformation can be reduced by factoring out possibly many equivalent matchings by the automorphism groups of the rules. The second transformation is based on the use of equivalence relations over graph items and offers a new way of performing simultaneous merging operations. The relevance of combining the two transformations is illustrated on a running example.

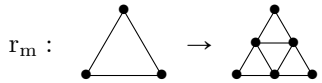
1 Introduction

Graph structures play an important role in the modeling and construction of complex systems in various disciplines including computer science, biology, chemistry or physics, as they provide natural and concise representation of many data structures. Computing with graphs as first-class citizens requires the use of advanced graph-based computational models. In contrast to term rewriting [2], there are different ways, in the literature, to define graphs (e.g., simple or multiple graphs, hyper-graphs, attributed graphs, etc.) as well as their transformations, see, e.g., [12, 16, 4].

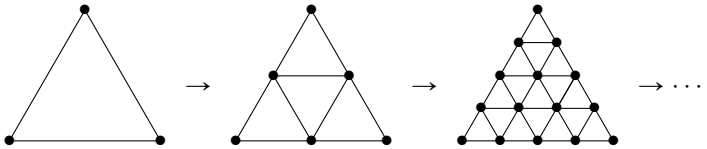
In this paper, we are interested in parallel transformations of graphs which yield deterministic computations. There are many situations where simultaneous graph transformations occur in practice such as social networks dynamics, cell-phones connections, cellular automata or biological processes such as plant growth. The need of using parallel graph transformations has been pointed out since the 70's, see e.g., [13, 18]. The main novelty of our proposal is twofold: first, overlapping transformations can be handled in parallel under a suitable condition called the *effective deletion property*. This new condition makes it possible to fire simultaneously two (or more) rules with a mild form of disagreement in the sense that one rule can delete an item (i.e., node, arrow or attribute) while this is not required by another rule. The second novelty of the paper consists in proposing graph equivalences to formally describe parallel merging of graph

items or attributes' expression evaluations. Furthermore, we introduce the notion of automorphism groups associated to graph rewrite rules which induce a substantial improvement of simultaneous graph transformations.

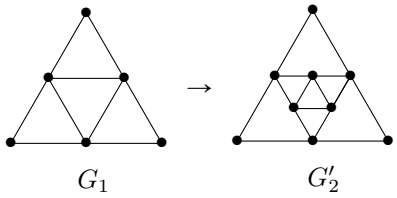
To motivate our purpose and to illustrate the introduced notions, we consider a running example borrowed from the rules defining mesh refinements [1]. In the following rule $\mathrm{r_m}$, a triangle is refined into four smaller triangles. Notice that this example does not show all the expressiveness of the rules we consider but illustrates sufficiently the investigated concepts.



This geometric rule specifies a sequence of mesh refinements as depicted below :



In this paper, we wish to view mesh refinement as a transformation of graphs, and to propose methods that could achieve this purpose. One obvious point is that $\mathrm{r_m}$ has to be applied simultaneously to every triangle subgraph of G_i to obtain G_{i+1}. However, if rule $\mathrm{r_m}$ is applied sequentially, say, at the center triangle subgraph of G_1, we get :



then it is no longer possible to obtain G_2 from G'_2 since the side subgraph triangles of G_1 have been modified and cannot be matched anymore by the left-hand side of the rule $\mathrm{r_m}$. In other words, the matchings of $\mathrm{r_m}$ in G_1 are not parallel independent. We therefore need to define a general parallel transformation that may yield a result unreachable by sequential rewriting. This will be achieved in Sections 5 and 6.

Before that, we start by introducing the considered definition of attributed graphs in Section 2, together with convenient notations. A set-theoretic framework is developed starting from Section 3 that will enable us to define graph transformations by an algebraic expression (Definition 5). Section 4 is dedicated to defining the notion of rewrite rules, together with their matchings. In Section 5, the general parallel graph transformations are defined and a central notion of *effective deletion property* of sets of matchings is exposed, which guarantees that graph objects are consistently deleted during the transformation. Section 6 is dedicated to a particular parallel rewrite relation where parallel matchings are

considered up to automorphisms, based on a notion of automorphism groups of the considered rules. This section uses notions borrowed from group theory. In Section 7, we introduce the notion of graph transformation as quotient graphs. This is not rule-based but allows one to write fancy definitions of merge actions over graphs that cannot be expressed with the rules above. Finally, related work and concluding remarks are given in Section 8. These transformations are all illustrated on the example of rule $\mathrm{r_m}$ in quite some detail. The missing proofs can be found in [5].

2 Preliminaries

We consider a fixed many-sorted signature Σ (see [15]). A *graph* G is a tuple $(V, A, s, t, \mathcal{A}, l)$ where V, A are sets, $\mathcal{A}$ is a Σ-algebra, s, t are the *source* and *target* functions from A to V and l is an *attribution of* G, i.e., a function from $V \cup A$ to $\mathscr{P}(\lfloor\mathcal{A}\rfloor)$ (the carrier set $\lfloor\mathcal{A}\rfloor$ of $\mathcal{A}$ is the disjoint union of the carrier sets of the sorts in $\mathcal{A}$). We assume that V, A and $\lfloor\mathcal{A}\rfloor$ are mutually disjoint, their elements are respectively called *vertices*, *arrows* and *attributes*. Hence vertices and arrows are attributed by *sets* of elements of a Σ-algebra. G is *finite* if the sets V, A and $l(x)$ are finite for all $x \in V \cup A$. The *carrier* of G is the set $\lfloor G\rfloor \stackrel{\text{def}}{=} V \cup A \cup \lfloor\mathcal{A}\rfloor$. When we speak of a graph G without specifying its components, these will be referred to as in $G = (\dot{G}, \vec{G}, \acute{G}, \grave{G}, \mathscr{A}_G, \mathring{G})$.

A graph H is a *subgraph* of G, written $H \lhd G$, if the *underlying graph* $(\dot{H}, \vec{H}, \acute{H}, \grave{H})$ of H is a subgraph of G's underlying graph (in the usual sense), $\mathscr{A}_H = \mathscr{A}_G$ and $\forall x \in \dot{H} \cup \vec{H}, \mathring{H}(x) \subseteq \mathring{G}(x)$.

A morphism α from graph H to graph G is a function from $\lfloor H\rfloor$ to $\lfloor G\rfloor$ such that the restriction of α to $\dot{H} \cup \vec{H}$ is a morphism from H's to G's underlying graphs (that is, $\acute{G} \circ \alpha = \alpha \circ \acute{H}$ and $\grave{G} \circ \alpha = \alpha \circ \grave{H}$, this restriction of α is called the *underlying graph morphism of* α), the restriction of α to $\lfloor\mathscr{A}_H\rfloor$ is a Σ-homomorphism from $\mathscr{A}_H$ to $\mathscr{A}_G$, denoted $\mathring{\alpha}$, and $\forall x \in \dot{H} \cup \vec{H}$, $\mathring{\alpha} \circ \mathring{H}(x) \subseteq \mathring{G} \circ \alpha(x)$. This means that α is an isomorphism if and only if α is a bijective morphism *and* α^{-1} is a morphism, hence if and only if the underlying graph morphism of α is an isomorphism, $\mathring{\alpha}$ is a Σ-isomorphism *and* $\mathring{\alpha} \circ \mathring{H} = \mathring{G} \circ \alpha$. For all $F \lhd H$, the *image* $\alpha(F)$ is the smallest subgraph of G w.r.t. the order $\lhd$ such that $\alpha|_{\lfloor F\rfloor}$ is a morphism from F to $\alpha(F)$.

If the underlying graph morphism of α is injective then α is called a *matching*. Note that the Σ-homomorphism $\mathring{\alpha}$ need not be injective.

Given two attributions l and l' of G we define $l \backslash l'$ (resp. $l \cap l'$, $l \cup l'$) as the attribution of G that maps any x to $l(x) \backslash l'(x)$ (resp. $l(x) \cap l'(x)$, $l(x) \cup l'(x)$). If l is an attribution of a subgraph $H \lhd G$, we extend it implicitely to the attribution of G that is identical to l on $\dot{H} \cup \vec{H}$ and maps any other x to $\varnothing$.

For any sets V, A and attribution l, we say that G *is disjoint from* V, A, l if $\dot{G} \cap V = \varnothing$, $\vec{G} \cap A = \varnothing$ and $\mathring{G}(x) \cap l(x) = \varnothing$ for all $x \in \dot{G} \cup \vec{G}$. We write $G \backslash [V, A, l]$ for the largest subgraph of G (w.r.t. $\lhd$) that is disjoint from G. It is easy to see that this subgraph always exists.

3 Joinable Graphs

In order to define parallel rewrite relations on graphs, it is convenient to join possibly many different graphs that have a common part, i.e., that are joinable. As a matter of fact, this notion also allows a simple definition of graph rewrite rules, and is crucial in defining the automorphism groups of these rules. We start with a simpler notion of joinable functions.

Definition 1 (joinable functions). *Two functions $f : D \to C$ and $g : D' \to C'$ are* joinable *if $\forall x \in D \cap D'$, $f(x) = g(x)$. Then, the* meet *of f and g is the function $f \curlywedge g : D \cap D' \to C \cap C'$ that is the restriction of f (or g) to $D \cap D'$. The* join *$f \curlyvee g$ is the unique function from $D \cup D'$ to $C \cup C'$ such that $f = (f \curlyvee g)|_D$ and $g = (f \curlyvee g)|_{D'}$.*

For any set I and any I-indexed family $(f_i : D_i \to C_i)_{i \in I}$ of pairwise joinable functions, let $\bigvee_{i \in I} f_i$ be the only function from $\bigcup_{i \in I} D_i$ to $\bigcup_{i \in I} C_i$ such that $f_i = \left(\bigvee_{i \in I} f_i\right)|_{D_i}$ for all $i \in I$.

If S and T are sets of functions, let $S \curlyvee T \stackrel{\text{def}}{=} \{f \curlyvee g \mid f \in S, g \in T\}$ and $S \circ T \stackrel{\text{def}}{=} \{f \circ g \mid f \in S, g \in T\}$, provided these operations can be applied. If f is a function, let $f \circ T \stackrel{\text{def}}{=} \{f\} \circ T$.

In particular, functions with disjoint domains are joinable (e.g. $\dot{\alpha}$ and $\vec{\alpha}$), and every function is joinable with itself: $f \curlyvee f = f \curlywedge f = f$. More generally, any two restrictions $f|_A$ and $f|_B$ of the same function f are joinable, $f|_A \curlywedge f|_B = f|_{A \cap B}$ and $f|_A \curlyvee f|_B = f|_{A \cup B}$.

It is obvious that these operations are commutative. On triples of pairwise joinable functions, they are also associative and distributive over each other.

Definition 2 (joinable graphs). *Two graphs H and G are* joinable *if $\mathscr{A}_H = \mathscr{A}_G$, $\dot{H} \cap \vec{G} = \vec{H} \cap \dot{G} = \varnothing$, and the functions $\acute{H}$ and $\acute{G}$ (and similarly $\grave{H}$ and $\grave{G}$) are joinable. We can then define the graphs*

$$H \sqcap G \stackrel{\text{def}}{=} (\ \dot{H} \cap \dot{G},\ \vec{H} \cap \vec{G},\ \acute{H} \curlywedge \acute{G},\ \grave{H} \curlywedge \grave{G},\ \mathscr{A}_H,\ \mathring{H} \cap \mathring{G}\ ),$$
$$H \sqcup G \stackrel{\text{def}}{=} (\ \dot{H} \cup \dot{G},\ \vec{H} \cup \vec{G},\ \acute{H} \curlyvee \acute{G},\ \grave{H} \curlyvee \grave{G},\ \mathscr{A}_H,\ \mathring{H} \cup \mathring{G}\ ).$$

Similarly, if $(G_i)_{i \in I}$ is an I-indexed family of graphs (where $I \neq \varnothing$) that are pairwise joinable, hence have the same algebra $\mathcal{A}$ of attributes, then let

$$\bigsqcup_{i \in I} G_i \stackrel{\text{def}}{=} (\ \bigcup_{i \in I} \dot{G}_i,\ \bigcup_{i \in I} \vec{G}_i,\ \bigvee_{i \in I} \acute{G}_i,\ \bigvee_{i \in I} \grave{G}_i,\ \mathcal{A},\ \bigcup_{i \in I} \mathring{G}_i\ ).$$

It is easy to see that these structures are graphs: the sets of vertices and arrows are disjoint and the adjacency functions have the correct domains and codomains. Note that if H and G are joinable then $H \sqcap G = G \sqcap H \lhd H \lhd H \sqcup G = G \sqcup H$. Similarly, if the G_i's are pairwise joinable then $\forall j \in I$, $G_j \lhd \bigsqcup_{i \in I} G_i$. We also see that any two subgraphs of G are joinable, and that $H \lhd G$ iff $H \sqcap G = H$ iff $H \sqcup G = G$. As above, on triples of pairwise joinable $\mathcal{A}$-graphs, these operations are associative and distributive over each other.

4 Rules

We consider rules with three joinable graphs L, K and R as depicted below.

The semantics of such rules is defined in Section 5. Informally, L shall be matched in the input graph G, the region $L\backslash K$ (the items matched by L but not by K) shall be removed from G, and the region $R\backslash L$ shall be added in order to obtain an image of R in the output graph.

In the following, we assume a set $\mathcal{V}$ disjoint from Σ, whose elements are called *variables*. For any finite $X \subseteq \mathcal{V}$, we call (Σ, X)-*graph* a finite graph G such that $\mathscr{A}_G = \mathscr{T}(\Sigma, X)$ (the Σ-term algebra, see e.g. [2, p. 49]). A Σ-*graph* is a $(\Sigma, \varnothing)$-graph. We define the set of variables occurring in a (Σ, X)-graph G as $\mathrm{Var}(G) \stackrel{\text{def}}{=} \bigcup_{x \in \dot{G} \cup \vec{G}} \left(\bigcup_{t \in \mathring{G}(x)} \mathrm{Var}(t) \right)$, where $\mathrm{Var}(t)$ is the set of variables occurring in t.

Definition 3 (rules, matchings).

A rule *r is a triple (L, K, R) of (Σ, X)-graphs such that L and R are joinable, $L \sqcap R \lhd K \lhd L$ and $\mathrm{Var}(L) = X$. Note that this implies that $\mathrm{Var}(R) \subseteq \mathrm{Var}(L)$, R and K are joinable and $R \sqcap K = L \sqcap R$. The rule r is* standard *if $L \sqcap R = K$.*

A matching *μ of r in a graph G is a matching from L to G such that*

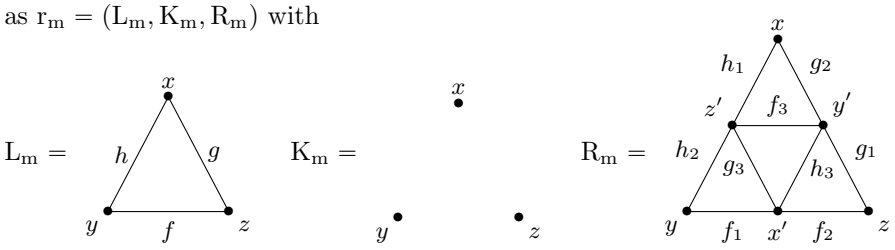
$$\mathring{\mu}(\mathring{L}(x) \backslash \mathring{K}(x)) \cap \mathring{\mu}(\mathring{K}(x)) = \varnothing$$

(or equivalently $\mathring{\mu}(\mathring{L}(x)\backslash\mathring{K}(x)) = \mathring{\mu}(\mathring{L}(x))\backslash\mathring{\mu}(\mathring{K}(x))$) for all $x \in \dot{K} \cup \vec{K}$. We denote $\mathscr{M}(r, G)$ the set of all matchings of r in G (they all have domain $\lfloor L \rfloor$).

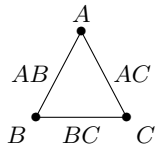
We consider finite sets $\mathcal{R}$ of rules such that $\forall r, r' \in \mathcal{R}$, if $(L, K, R) = r \neq r' = (L', K', R')$ then $\dot{L} \cup \vec{L} \neq \dot{L}' \cup \vec{L}'$, so that $\lfloor L \rfloor \neq \lfloor L' \rfloor$ hence $\mathscr{M}(r, G) \cap \mathscr{M}(r', G) = \varnothing$ for any graph G; we then write $\mathscr{M}(\mathcal{R}, G)$ for $\biguplus_{r \in \mathcal{R}} \mathscr{M}(r, G)$. For any $\mu \in \mathscr{M}(\mathcal{R}, G)$ there is a unique rule $\mathrm{r}_\mu \in \mathcal{R}$ such that $\mu \in \mathscr{M}(\mathrm{r}_\mu, G)$, and its components are denoted $\mathrm{r}_\mu = (\mathrm{L}_\mu, \mathrm{K}_\mu, \mathrm{R}_\mu)$.

Note that if X were allowed to contain a variable v not occurring in L, then v would freely match any element of $\mathscr{A}_G$ and the set $\mathscr{M}(r, G)$ would contain as many matchings with essentially the same effect.

Example 1. Let us consider the rule given in the introduction. It could be defined as $\mathrm{r_m} = (\mathrm{L_m}, \mathrm{K_m}, \mathrm{R_m})$ with



We assume, for the rule above, that all attributes are empty, hence $\mathrm{L_m}$, $\mathrm{K_m}$ and $\mathrm{R_m}$ are Σ-graphs. Each edge f, g, h represents a pair of opposite arrows; for sake of simplicity they will be treated as single objects. Note that $\mathrm{r_m}$ is a standard rule. A matching μ of $\mathrm{r_m}$ in the graph

$G_0 =$ 

is given by $\mu = \{(x, A), (y, B), (z, C), (f, BC), (g, AC), (h, AB)\}$. The Σ-algebra $\mathscr{A}_{G_0}$ and the Σ-morphism $\mathring{\mu}$ are not relevant here, we can choose $\mathscr{T}(\Sigma, \varnothing)$ and its identity morphism (though other algebras and attributes will be adopted later).

A rewrite step may involve the creation of new vertices in a graph, corresponding to the vertices of a rule that have no match in the input graph, i.e., those in $\dot{R}\backslash\dot{L}$ (or similarly may create new arrows). These vertices should really be new, not only different from the vertices of the original graph but also different from the vertices created by other transformations (corresponding to other matchings in the graph). This is computationally easy to do but not that easy to formalize in an abstract way. We simply reuse the vertices x from $\dot{R}\backslash\dot{L}$ by *indexing* them with any relevant matching μ, each time yielding a new vertex (x, μ) which is obviously different from any new vertex (x, ν) for any other matching $\nu \neq \mu$, and also from any vertex of G since μ depends[1] on G.

Definition 4 (graph $G{\uparrow_\mu}$ and matching $\mu{\uparrow}$). *For any rule $r = (L, K, R)$, graph G and $\mu \in \mathscr{M}(r, G)$ we define a graph $G{\uparrow_\mu}$ together with a matching $\mu{\uparrow}$ of R in $G{\uparrow_\mu}$. We first define the sets*

$$\dot{G}{\uparrow_\mu} \stackrel{\text{def}}{=} \mu(\dot{R} \cap \dot{K}) \uplus ((\dot{R}\backslash\dot{K}) \times \{\mu\}) \text{ and } \vec{G}{\uparrow_\mu} \stackrel{\text{def}}{=} \mu(\vec{R} \cap \vec{K}) \uplus ((\vec{R}\backslash\vec{K}) \times \{\mu\}).$$

Next we define $\mu{\uparrow}$ by: $\mathring{\mu}{\uparrow} \stackrel{\text{def}}{=} \mathring{\mu}$ and for all $x \in \dot{R} \cup \vec{R}$, if $x \in \dot{K} \cup \vec{K}$ then $\mu{\uparrow}(x) \stackrel{\text{def}}{=} \mu(x)$ else $\mu{\uparrow}(x) \stackrel{\text{def}}{=} (x, \mu)$. Since the restriction of $\mu{\uparrow}$ to $\dot{R} \cup \vec{R}$ is bijective, then $\mu{\uparrow}$ is a matching from R to the graph

$$G{\uparrow_\mu} \stackrel{\text{def}}{=} (\dot{G}{\uparrow_\mu}, \vec{G}{\uparrow_\mu}, \mathscr{A}_G, \mu{\uparrow} \circ \acute{R} \circ \mu{\uparrow}^{-1}, \mu{\uparrow} \circ \grave{R} \circ \mu{\uparrow}^{-1}, \mathring{\mu}{\uparrow} \circ \mathring{R} \circ \mu{\uparrow}^{-1}).$$

Example 2. Following Example 1, we get $\dot{G_0}{\uparrow_\mu} = \{A, B, C, (x', \mu), (y', \mu), (z', \mu)\}$, $\vec{G_0}{\uparrow_\mu} = \{(f_1, \mu), \ldots, (h_3, \mu)\}$, $\mu{\uparrow} = \{(x, A), (y, B), (z, C), (x', (x', \mu)), (y', (y', \mu)), (z', (z', \mu)), (f_1, (f_1, \mu)), \ldots, (h_3, (h_3, \mu))\}$. The graph $G_0{\uparrow_\mu}$ is obtained as $\mu{\uparrow}(\mathrm{R_m})$.

By construction μ and $\mu{\uparrow}$ are joinable and $\mu \curlywedge \mu{\uparrow}$ is a matching from $R \sqcap K$ to $\mu(R \sqcap K)$. It is easy to see that the graph G and the graphs $G{\uparrow_\mu}$ are pairwise joinable.

[1] $\lfloor G \rfloor$ is the codomain of μ, hence $(x, \mu) \notin \lfloor G \rfloor$ by the axiom of regularity from set theory.

5 Parallel Rewriting

For any set $M \subseteq \mathscr{M}(\mathcal{R}, G)$ of matchings in a graph G we define below how to rewrite G by applying simultaneously the rules associated with matches in M.

Definition 5 (graph $G_{\|M}$). *For any graph G and $M \subseteq \mathscr{M}(\mathcal{R}, G)$, let*

$$G_{\|M} \stackrel{\text{def}}{=} G\backslash[V, A, l] \sqcup \bigsqcup_{\mu \in M} G\!\uparrow_{\mu}, \text{ where}$$

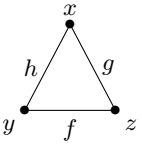
$$V \stackrel{\text{def}}{=} \bigcup_{\mu \in M} \mu(\dot{\mathrm{L}}_\mu \backslash \dot{\mathrm{K}}_\mu),\ A \stackrel{\text{def}}{=} \bigcup_{\mu \in M} \mu(\vec{\mathrm{L}}_\mu \backslash \vec{\mathrm{K}}_\mu) \text{ and } l \stackrel{\text{def}}{=} \bigcup_{\mu \in M} \mathring{\mu} \circ (\mathring{\mathrm{L}}_\mu \backslash \mathring{\mathrm{K}}_\mu) \circ \mu^{-1}.$$

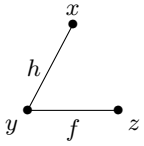
Note that l is only defined on $\bigsqcup_{\mu \in M} \mu(\mathrm{K}_\mu)$; so l is implicitly extended to the suitable domain by mapping other vertices and arrows to $\varnothing$. $G_{\|M}$ is guaranteed to be a graph since the $\sqcup$ operation is only applied on joinable graphs.

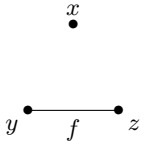
The definition of $G_{\|M}$ bears some similarity with the double pushout diagram (see [12]), with $G\backslash[V, A, l]$ as a pushout complement of G and $\bigsqcup_{\mu \in M} G\!\uparrow_{\mu}$ as a pushout. But we are not restricted by the gluing condition, and since we use a set of matchings the pushout is actually a colimit. The case where M is a singleton defines the classical semantics of one sequential rewrite step.

It is obvious that $G_{\|M}$ contains images of the right hand sides of all the rules involved (through the elements M). However it may be the case that some elements of V, A or l occur in $\bigsqcup_{\mu \in M} G\!\uparrow_{\mu}$, hence also in the result $G_{\|M}$, since any two matchings may conflict as one retains what another removes as illustrated in the following example.

Example 3. Let us consider the (non standard) rule $r_{\mathrm{c}} = (L_{\mathrm{c}}, K_{\mathrm{c}}, R_{\mathrm{c}})$ with

$L_{\mathrm{c}} =$ 

$K_{\mathrm{c}} =$ 

$R_{\mathrm{c}} =$ 

This rule removes g since g is in L_{c} but not in K_{c} , retains f (and all vertices) since f is in $K_{\mathrm{c}} \sqcap R_{\mathrm{c}}$, and does not care about h since h is in K_{c} but not in R_{c}. Nothing is added since $K_{\mathrm{c}} \sqcap R_{\mathrm{c}} = R_{\mathrm{c}}$. We consider the same graph G_0 and matching μ as in Example 1. Let $\nu = \mu \circ (x\ y)(f\ g)$ (in cyclic notation), this is obviously a matching of r_{c} in G_0. By μ we must remove AC and retain BC, while ν asks exactly the opposite which means there is a conflict between the application of the two matches. We easily see that the graph $G_0\!\uparrow_{\mu}$ contains BC and that $G_0\!\uparrow_{\nu}$ contains AC, so that $G_{0\|\{\mu,\nu\}} = G_0$, hence the instructions of removing AC and BC have not been fulfilled. The reader may check that no such conflict occurs between μ and $\mu \circ (y\ z)(g\ h)$; they remove AC and AB.

Since the semantics of individual rules have to be preserved under parallelization, we must avoid such conflicts by stating that any item deleted by any rule should not occur in the result. We can however allow attributes to be removed and yet restored: this is a situation similar to an assignment $v := 1$, where the former value of v is deleted unless it is 1.

Definition 6 (effective deletion property). *For any graph G and set $M \subseteq \mathscr{M}(\mathcal{R}, G)$, let V, A, l as in Definition 5 and $l' \stackrel{\text{def}}{=} \bigcup_{\mu \in M} \mathring{\mu} \circ (\mathring{\mathrm{R}}_\mu \backslash \mathring{\mathrm{K}}_\mu) \circ \mu^{-1}$ then M has the* effective deletion property *if $G_{\|M}$ is disjoint from $V, A, l \backslash l'$.*

We may ask wether it is possible to ensure from particular properties of $\mathcal{R}$ that effective deletion holds for all G and M. It is however easy to see that, given a rule that removes an object, say a vertex, and another (or the same) rule that retains some vertex, there always exists a graph G and two matchings in G that do not have the effective deletion property, as they conflict on the same vertex. Hence effective deletion should be checked for each G and M.

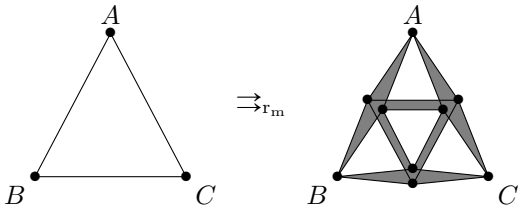
This naturally leads to the following definition.

Definition 7 (full parallel rewriting). *For any finite set of rules $\mathcal{R}$, we define the relation $\rightrightarrows_{\mathcal{R}}$ of* full parallel rewriting *between graphs by stating that, for all G such that $\mathscr{M}(\mathcal{R}, G)$ has the effective deletion property, $G \rightrightarrows_{\mathcal{R}} G_{\|\mathscr{M}(\mathcal{R},G)}$.*

6 Parallel Rewriting modulo Automorphisms

Using the full set of matchings exceeds the needs of mesh refinement, see below.

Example 4. Following Example 1, we see that there are 6 matchings of $\mathrm{r_m}$ in G_0, hence the relation $\rightrightarrows_{\mathrm{r_m}}$ does not create 3 but 18 new vertices, not 9 but 54 new edges, which is illustrated as gray areas below.



We therefore wish to select a subset M of $\mathscr{M}(\mathcal{R}, G)$ for defining a rewrite relation that yields more natural and concise graphs.

In Example 4, the similarities between the 6 matchings clearly come from the symmetries of rule $\mathrm{r_m}$. These depend on the automorphisms of the graphs $\mathrm{L_m}$, $\mathrm{K_m}$ and $\mathrm{R_m}$, i.e., on the groups commonly denoted $\mathrm{Aut}(\mathrm{L_m})$, $\mathrm{Aut}(\mathrm{K_m})$ and $\mathrm{Aut}(\mathrm{R_m})$. We need to build a notion of rule automorphisms that properly accounts for the interactions between these 3 groups. We first extend the notion of automorphism groups of graphs to their subgraphs.

Definition 8 (groups $\mathrm{Aut}_G(H_1, \ldots, H_n)$ and $\mathcal{S}|_H$). *For all $n \geqslant 1$, graph G and subgraphs $H, H_1, \ldots, H_n \lhd G$, let*

$$\mathrm{Aut}_G(H) \stackrel{\text{def}}{=} \{\alpha \in \mathrm{Sym}(\dot{G}) \curlyvee \mathrm{Sym}(\vec{G}) \curlyvee \mathrm{Aut}(\mathscr{A}_G) \mid \alpha(H) = H\},$$

$$\mathrm{Aut}_G(H_1, \ldots, H_n) \stackrel{\text{def}}{=} \bigcap_{i=1}^{n} \mathrm{Aut}_G(H_i).$$

For any $\alpha \in \mathrm{Aut}_G(H)$*, we write* $\alpha|_H$ *for* $\alpha|_{\lfloor H \rfloor}$*, and for any subgroup* $\mathcal{S}$ *of* $\mathrm{Aut}_G(H)$*, let* $\mathcal{S}|_H = \{\alpha|_H \mid \alpha \in \mathcal{S}\}$*; this is a subgroup of* $\mathrm{Aut}(H)$*.*

It is obvious that $\mathrm{Aut}_G(G) = \mathrm{Aut}(G)$. We see that $\mathrm{Aut}_G(H)$ is a permutation group on $\lfloor G \rfloor$, but only the graph structure of H is involved in the constraint $\alpha(H) = H$, not the structure of G.

Example 5. Take for instance $H = x \underset{g}{\overset{f}{\rightleftarrows}} y$ and $G = x \underset{g}{\overset{f}{\rightleftarrows}} y \underset{k}{\overset{h}{\rightleftarrows}} z$ with empty attributes. We have $\mathrm{Aut}(H) = \{1_H, (x)(y)(f\ g)\}$ and $\mathrm{Aut}(G) = \{1_G, (x)$
$(y)(z)(f\ g)(h)(k)\}$. We write non permuted points such as (x) in order to make the domains explicit. However, in $\mathrm{Aut}_G(H)$ the permutations of objects that do not belong to H are free, hence

$$\begin{aligned}\mathrm{Aut}_G(H) &= \{1_G, (x)(y)(z)(f\ g)(h)(k), (x)(y)(z)(f)(g)(h\ k),\\ &\qquad\qquad (x)(y)(z)(f\ g)(h\ k)\}\\ &= \mathrm{Aut}(H) \curlyvee \{(z)(h)(k), (z)(h\ k)\}\\ &= \mathrm{Aut}(H) \curlyvee \{(z)\} \curlyvee \{(h)(k), (h\ k)\}\\ &= \mathrm{Aut}(H) \curlyvee \mathrm{Sym}\{z\} \curlyvee \mathrm{Sym}\{h, k\}.\end{aligned}$$

Since $\mathscr{A}_H = \mathscr{A}_G$, it is easy to see that $\mathrm{Aut}_G(H) = \mathrm{Aut}(H) \curlyvee \mathrm{Sym}(\dot{G}\backslash\dot{H}) \curlyvee \mathrm{Sym}(\vec{G}\backslash\vec{H})$ always holds and thus $\mathrm{Aut}_G(H)|_H = \mathrm{Aut}(H)$. This means that, compared to the elements of $\mathrm{Aut}(H)$ which are only permutations of $\lfloor H \rfloor$, the elements of $\mathrm{Aut}_G(H)$ are *all* possible extensions of the elements of $\mathrm{Aut}(H)$ to permutations of $\lfloor G \rfloor$. This allows us to conveniently intersect the automorphism groups of joinable graphs.

Definition 9 (group $\mathrm{Aut}(r)$**, relation** $\approx$**).** *For any rule* $r = (L, K, R)$*, the* automorphism group *of* r *is* $\mathrm{Aut}(r) \stackrel{\text{def}}{=} \mathrm{Aut}_{L \sqcup R}(L, K, R)|_L$*. For any graph* G*, let* $\approx$ *be the equivalence relation on* $\mathscr{M}(\mathcal{R}, G)$ *defined by* $\mu \approx \nu$ *iff* $\mu \circ \mathrm{Aut}(\mathrm{r}_\mu) = \nu \circ \mathrm{Aut}(\mathrm{r}_\nu)$*. The equivalence class of* μ *is denoted* $\bar{\mu}$*. For any subset* $M \subseteq \mathscr{M}(\mathcal{R}, G)$ *we write* $\bar{M}$ *for the set* $\bigcup_{\mu \in M} \bar{\mu}$*.*

Lemma 1. $\forall \mu \in \mathscr{M}(\mathcal{R}, G)$*,* $\bar{\mu} = \mu \circ \mathrm{Aut}(\mathrm{r}_\mu)$*.*

Note that $|\bar{\mu}| \leqslant |\mathrm{Aut}(\mathrm{r}_\mu)|$ and that the equality holds if μ is injective. The more symmetric a rule is, the more matchings are likely to occur in the equivalence classes of matchings of this rule. The definition of the automorphism groups of rules has been crafted so that the isomorphism classes of the output graphs do not depend on the choice of elements in the equivalence classes of matchings.

Theorem 1. *For any graph* G*, any* $\mathcal{M} \subseteq \mathscr{M}(\mathcal{R}, G)$ *and any minimal sets* M, N *such that* $\mathcal{M} = \bar{M} = \bar{N}$*, the graphs* $G_{\|M}$ *and* $G_{\|N}$ *are isomorphic.*

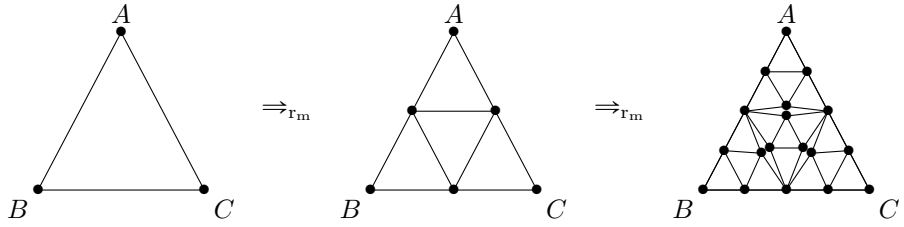
This means that the following graph rewrite relation is deterministic up to isomorphism.

Definition 10 (parallel rewriting modulo automorphisms). *For any finite set of rules $\mathcal{R}$, we define the relation $\Rightarrow_{\mathcal{R}}$ of* parallel rewriting modulo automorphisms *between graphs by stating that, for all G and minimal set M such that $\bar{M} = \mathscr{M}(\mathcal{R}, G)$ and M has the effective deletion property, $G \Rightarrow_{\mathcal{R}} G_{\|M}$.*

Example 6. Following Example 1, we see that the group $\mathrm{Aut}(\mathrm{K_m})$ is generated by $\{(x\ y), (x\ z)\}$ (this is $\mathrm{Sym}\{x, y, z\}$), the group $\mathrm{Aut}(\mathrm{L_m})$ is generated by $\{(x\ y)(f\ g), (x\ z)(f\ h)\}$, and $\mathrm{Aut}(\mathrm{R_m})$ is generated by $\{\rho_1, \rho_2\}$ where $\rho_1 = (x\ y)(x'\ y')(h_1\ h_2)(f_1\ g_2)(f_2\ g_1)(f_3\ g_3)$ and $\rho_2 = (x\ z)(x'\ z')(g_1\ g_2)(f_1\ h_2)(f_2\ h_1)$ $(f_3\ h_3)$. We then use the facts that

$$\mathrm{Aut}_{\mathrm{L_m} \sqcup \mathrm{R_m}}(\mathrm{L_m}) = \mathrm{Aut}(\mathrm{L_m}) \curlyvee \mathrm{Sym}\{x', y', z'\} \curlyvee \mathrm{Sym}\{f_1, g_1, h_1, f_2, g_2, h_2, f_3, g_3, h_3\}$$
$$\mathrm{Aut}_{\mathrm{L_m} \sqcup \mathrm{R_m}}(\mathrm{K_m}) = \mathrm{Aut}(\mathrm{K_m}) \curlyvee \mathrm{Sym}\{x', y', z'\} \curlyvee \mathrm{Sym}\{f, g, h, f_1, \ldots, h_3\}$$
$$\mathrm{Aut}_{\mathrm{L_m} \sqcup \mathrm{R_m}}(\mathrm{R_m}) = \mathrm{Aut}(\mathrm{R_m}) \curlyvee \mathrm{Sym}\{f, g, h\}$$

to see that $\mathrm{Aut}_{\mathrm{L_m} \sqcup \mathrm{R_m}}(\mathrm{L_m})$ is a subgroup of $\mathrm{Aut}_{\mathrm{L_m} \sqcup \mathrm{R_m}}(\mathrm{K_m})$, and then we easily see that $\mathrm{Aut}_{\mathrm{L_m} \sqcup \mathrm{R_m}}(\mathrm{L_m}, \mathrm{K_m}, \mathrm{R_m}) = \mathrm{Aut}_{\mathrm{L_m} \sqcup \mathrm{R_m}}(\mathrm{L_m}, \mathrm{R_m})$ is generated by $\{\rho_1 \curlyvee (f\ g), \rho_2 \curlyvee (f\ h)\}$. We thus obtain that $\mathrm{Aut}(\mathrm{r_m})$ is the group generated by $\{(\rho_1 \curlyvee (f\ g))|_{\mathrm{L_m}}, (\rho_2 \curlyvee (f\ h))|_{\mathrm{L_m}}\} = \{(x\ y)(f\ g), (x\ z)(f\ h)\}$, i.e., $\mathrm{Aut}(\mathrm{r_m}) = \mathrm{Aut}(\mathrm{L_m})$ and this group has 6 elements. The 6 matchings of $\mathrm{L_m}$ in G_0 are therefore all equivalent by $\approx$. Similarly, the 24 matchings of $\mathrm{L_m}$ in G_1 form 4 equivalence classes modulo $\approx$. This yields the following parallel rewrite steps modulo automorphisms.



7 Graph Transformations as Quotients

The last example shows that we still need to be able to merge graph items. In this section, we propose to use equivalence relations to merge graph vertices or edges as well as their possible attributes.

Definition 11 (Congruence on a Graph, quotient graph). *For any graph G, a* congruence $\mathcal{C}$ on G *is a tuple $(\sim, \simeq, \cong)$ where $\cong$ is a congruence on $\mathscr{A}_G$ (see [2, p. 45]) and $\sim$, $\simeq$ are equivalence relations on $\dot{G}$, $\vec{G}$ respectively, such that*

$$\forall f, g \in \vec{G},\ \textit{if } f \simeq g \textit{ then } \acute{G}(f) \sim \acute{G}(g) \textit{ and } \grave{G}(f) \sim \grave{G}(g). \tag{1}$$

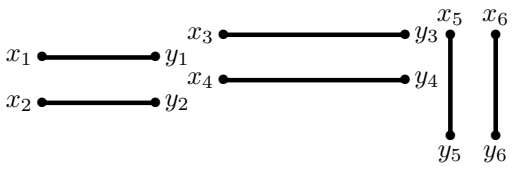
$\mathcal{C}$ is neutral *if $\cong$ is the identity relation on $\mathscr{A}_G$.*

The quotient of G by $\mathcal{C}$ *is the graph $G/\mathcal{C} = (\dot{G}/\sim, \vec{G}/\simeq, s, t, \mathscr{A}_G/\cong, l)$ where $\dot{G}/\sim$ and $\vec{G}/\simeq$ are the standard quotients (the sets of equivalence classes), $\mathscr{A}_G/\cong$ is the quotient algebra (see [2, p. 45]), and*

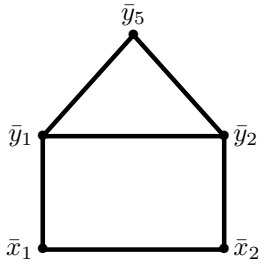
– *for any* $F \in \vec{G}/\simeq$ *and any* $f \in F$, $s(F)$ *(resp.* $t(F)$*) is the class of* $\acute{G}(f)$ *(resp.* $\grave{G}(f)$*) modulo* $\sim$ *(which by (1) depends only on the class* F *of* f *modulo* $\simeq$*),*
– *for any* $C \in (\dot{G}/\sim) \cup (\vec{G}/\simeq)$, $l(C) \stackrel{\text{def}}{=} \bigcup_{x \in C} \{\bar{t} \mid t \in \mathring{G}(x)\}$, *where* $\bar{t}$ *is the equivalence class of* $t \in \lfloor \mathscr{A}_G \rfloor$ *modulo* $\cong$*.*

Note that if $\mathcal{C}$ is neutral then $\mathscr{A}_{G/\mathcal{C}}$ is identified with $\mathscr{A}_G$.

Example 7. Let us consider the following graph Z consisting of six sticks. These sticks could be thought, for example, as a furniture kit. Every stick can be considered as a pair of two opposite arrows attributed by its length (not depicted), with $\lfloor \mathscr{A}_Z \rfloor = \mathbb{R}$. Each end of a stick is a vertex whose attribute is $\varnothing$.



The instructions for assembling the kit can be given as the following equivalence relation on vertices: $x_1 \sim x_3$, $x_2 \sim y_3$, $y_1 \sim x_4 \sim x_5$, $y_2 \sim y_4 \sim x_6$ and $y_5 \sim y_6$. We now consider the neutral congruence $\mathcal{Z} = (\sim, =_{\vec{Z}}, =_{\mathscr{A}_G})$, where $=_{\vec{Z}}$ is the identity relation on $\vec{G}$ (so that condition (1) is obviously satisfied). The reader can easily check that the quotient graph $Z/\mathcal{Z}$ is the following one.



By definition, a congruence is specific to a particular graph, hence we need a way to define a congruence from a graph in order to define a universal transformation by taking quotients of graphs. There are many ways this could be done, and we only propose a solution relevant to mesh refinement.

Definition 12 (The localizing congruence). *For any graph* G*, the* localizing congruence $\mathscr{L}_G$ *on* G *is the tuple* $(\sim, \simeq, =_{\mathscr{A}_G})$ *where*

– *for all* $x, y \in \dot{G}$, $x \sim y$ *iff* $\mathring{G}(x) = \mathring{G}(y)$,
– *for all* $f, g \in \vec{G}$, $f \simeq g$ *iff* $\acute{G}(f) \sim \acute{G}(g)$ *and* $\grave{G}(f) \sim \grave{G}(g)$,
– $=_{\mathscr{A}_G}$ *is the identity relation on* $\mathscr{A}_G$*.*

We write $\gtrsim_{\mathscr{L}}$ *for the binary relation on graphs defined by* $G \gtrsim_{\mathscr{L}} (G/\mathscr{L}_G)$ *for all graphs* G*.*

In this transformation, the attributes of the vertices act as coordinates in the sense that there can be only one vertex (or point) at each coordinate, and only one arrow from one point to another. Note that, since $\mathscr{L}_G$ is neutral, then obviously $\mathscr{A}_G = \mathscr{A}_H$ whenever $G \gtrsim_{\mathscr{L}} H$.

Example 8. In Example 1, it was not necessary to define precisely which algebra was considered since all attributes were empty. We now give more substance to the mesh graphs by assuming that every vertex is attributed with its coordinates in the affine plane, i.e., by a singleton containing an element of $\mathbb{R}^2$. The only operation we need is the function that returns the coordinates of the middle of two points, hence we take $\Sigma = \{\texttt{mid}\}$ where $\texttt{mid}$ has arity 2, and we consider the Σ-algebra $\mathcal{P}$ with carrier set $\mathbb{R}^2$ where $\texttt{mid}$ is interpreted as the function that to any $(x, y) \in \mathbb{R}^2$ and $(x', y') \in \mathbb{R}^2$ maps $(\frac{x+x'}{2}, \frac{y+y'}{2}) \in \mathbb{R}^2$. We start with the graph G_0 as in Example 1, but with $\mathscr{A}_{G_0} = \mathcal{P}$, $\mathring{G}_0(A)$ is a singleton that contains the coordinates of A (an element of $\mathbb{R}^2$), and similarly for vertices B and C.

In order to match these coordinates we also need to use variables in the rule $\mathrm{r_m}$, hence we consider 3 distinct variables $u, v, w \in \mathscr{V}$ and let $X = \{u, v, w\}$. We consider the graphs $\mathrm{L_m}$, $\mathrm{K_m}$, $\mathrm{R_m}$ of Example 1 but with the algebra $\mathscr{T}(\Sigma, X)$ and with the following attributes on vertices:

- $\mathring{\mathrm{L}}_\mathrm{m}(x) = \mathring{\mathrm{K}}_\mathrm{m}(x) = \mathring{\mathrm{R}}_\mathrm{m}(x) = \{u\}$, $\mathring{\mathrm{L}}_\mathrm{m}(y) = \mathring{\mathrm{K}}_\mathrm{m}(y) = \mathring{\mathrm{R}}_\mathrm{m}(y) = \{v\}$ and $\mathring{\mathrm{L}}_\mathrm{m}(z) = \mathring{\mathrm{K}}_\mathrm{m}(z) = \mathring{\mathrm{R}}_\mathrm{m}(z) = \{w\}$. These attributes are therefore not modified by the rule.
- We must also compute the coordinates of the new vertices x', y', z' created by $\mathrm{R_m}$. One difficulty is that in the algebra of terms $\texttt{mid}(u, v)$ is different from $\texttt{mid}(v, u)$, and if we choose one then we necessarily loose some automorphisms of the rule. The solution is to take both, that is

$$\begin{aligned}
\mathring{\mathrm{R}}_\mathrm{m}(x') &= \{\texttt{mid}(v, w), \texttt{mid}(w, v)\},\\
\mathring{\mathrm{R}}_\mathrm{m}(y') &= \{\texttt{mid}(u, w), \texttt{mid}(w, u)\},\\
\mathring{\mathrm{R}}_\mathrm{m}(z') &= \{\texttt{mid}(v, u), \texttt{mid}(u, v)\}.
\end{aligned}$$

With these attributes it is easy to see that $\mathrm{Aut}(\mathrm{r_m})$ is generated by the two permutations $(x\ y)(f\ g)(u\ v)$ and $(x\ z)(f\ h)(u\ w)$, and has 6 elements. The 6 matchings of $\mathrm{L_m}$ in G_0 are all $\approx$-equivalent to the matching $\mu \in \mathscr{M}(\mathrm{r_m}, G_0)$ with the same images of vertices and arrows as in Example 1, and where $\mathring{\mu}(x)$ is the coordinate of A, and similarly for y and z. Hence we have $G_0 \Rightarrow_{\mathrm{r_m}} G_1$, where to every vertex is attributed the coordinate of the corresponding point of $\mathcal{P}$. Applying $\Rightarrow_{\mathrm{r_m}}$ still yields a graph with too many vertices and edges, though with correct coordinates, hence quotienting this graph with its localizing congruence yields the graph G_2. We thus see that

$$G_0 \ \ (\Rightarrow_{\mathrm{r_m}} \circ \gtrsim_{\mathscr{L}}) \ \ G_1 \ \ (\Rightarrow_{\mathrm{r_m}} \circ \gtrsim_{\mathscr{L}}) \ \ G_2 \ \ (\Rightarrow_{\mathrm{r_m}} \circ \gtrsim_{\mathscr{L}}) \ \ \ldots$$

ad infinitum. Note that $\rightrightarrows_{\mathrm{r_m}} \circ \gtrsim_{\mathscr{L}}$ yields the same result as $\Rightarrow_{\mathrm{r_m}} \circ \gtrsim_{\mathscr{L}}$, though in a less efficient way since $\mathrm{Aut}(\mathrm{r_m})$ needs only be computed once.

8 Related Work and Concluding Remarks

Parallel graph rewriting has already been considered in the literature. In the mid-seventies, H. Ehrig and H.-J. Kreowski [13] tackled the problem of parallel graph transformations and proposed conditions under which parallel graph transformations could be sequentialized and how sequential independent graph transformations could be parallelized. This pioneering work has been considered for several algebraic graph transformation approaches, see, e.g., the most recent contributions [9, 23, 22] or Volume 3 of the Handbook of Graph Grammars and Computing by Graph Transformation [14]. However, this stream of work departs drastically from our goal where parallel graph transformations are not aimed to be sequentialized.

Non independent parallel graph transformations has been considered in the Double-Pushout setting, see e.g. [25] where rules can be amalgamated by agreeing on common deletions and preservations. However, the amalgamation technique does not allow the amount of overlaps achieved in the present framework. Indeed, the effective deletion property makes it possible for one rule to delete an item that is matched but not deleted by another (non standard) rule. This is an essential feature for instance in cellular automata where the state of a cell can be modified by one rule and only consulted by others (see [5]).

In [20], a framework based on the algebraic Single-Pushout approach has been proposed and where parallel transformations consider only matchings provided by a control flow mapping. The users can solve the possible conflicts between the rules by providing the right control flow. More recently, a parallel graph rewrite relation has been defined in [11] for a special kind of graphs called port-graphs. Unfortunately, such graphs are not closed under parallel graph transformation, in the sense that a port-graph can be rewritten in a structure which is not a port-graph. In addition, conditions for avoiding conflicts in parallel transformations have been defined over the considered rewrite rules, which limits drastically the class of the considered systems. The present framework provides more abstract and more general conditions over matchings that ensure a correct definition of parallel graph transformations for a large class of systems.

In [24, chapter 14], parallel graph transformations have been studied in order to improve the operational semantics of the functional programming language CLEAN [17]. In that contribution, the authors do not deal with true parallelism but rather have an interleaving semantics. This particularly entails that their parallel rewrite steps can be simulated by sequential ones. This is also the case for other frameworks where massive parallel graph transformations is defined so that it can be simulated by sequential rewriting e.g., [10, 22, 21].

Graph equivalence has already been used to encode vertex merging as in [3] where the notion of e-graphs has been proposed. An e-graph is a pair $(G, \sim)$ of a hypergraph and an equivalence over vertices. Contrary to our framework, quotient graphs are not used per se as objects to be transformed. Furthermore, our notion of equivalence over graphs is more general since it can be defined either on vertices, arrows or even attributes.

Transforming a graph by using simultaneously several rules in parallel is not an easy task. As mentioned above, most of the proposals in the literature consider parallel transformations that can be sequentialized. In this paper, we have developed a new framework where true parallel graph transformations are defined following an algorithmic approach. We proposed deterministic parallel rewrite relations, particularly one based on the notion of automorphism groups of rules. Furthermore, we defined the notion of *effective deletion property* of matchings which ensures that these relations are well-behaved, even when the overlappings of matches forbids sequentialization, as illustrated by the mesh refinement rule $\mathrm{r_m}$. The proposed rewrite relations may be used in several contexts such as extensions of L-systems to dynamic graph structures (see, e.g., [26, 18, 19]). For the sake of simplicity we have not addressed here the problem of the finiteness of the graphs obtained by parallel rewriting, see [5] for a discussion and results on this subject.

The considered rewrite systems could be enriched by means of new features such as vertex and edge cloning as proposed in [7, 8]. This is possible in an algebraic framework, see [6]. Future work also includes implementation issues, particularly for the parallel rewrite relation up to automorphisms. The present framework has been designed so that the automorphism groups of rules are finite permutation groups, thus paving the way to efficient implementations through the methods of Algorithmic Group Theory.

References

1. R. B. Andrew, A. H. Sherman, and A. Weiser. Some refinement algorithms and data structures for regular local mesh refinement, 1983.
2. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
3. P. Baldan, F. Gadducci, and U. Montanari. Concurrent rewriting for graphs with equivalences. In *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2006.
4. A. Boronat, R. Heckel, and J. Meseguer. Rewriting logic semantics and verification of model transformations. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering*, pages 18–33. Springer, 2009.
5. T. Boy de la Tour and R. Echahed. A set-theoretic framework for parallel graph rewriting. *CoRR*, (abs/1808.03161), 2018.
6. T. Boy de la Tour and R. Echahed. True parallel graph transformations: an algebraic approach based on weak spans. *CoRR*, (abs/1904.08850), 2019.
7. J. H. Brenas, R. Echahed, and M. Strecker. Verifying graph transformation systems with description logics. In *11th ICGT*, volume 10887 of *LNCS*, pages 155–170. Springer, 2018.
8. A. Corradini, D. Duval, R. Echahed, F. Prost, and L. Ribeiro. AGREE - algebraic graph rewriting with controlled embedding. In *8th ICGT*, volume 9151 of *LNCS*, pages 35–51. Springer, 2015.
9. A. Corradini, D. Duval, M. Löwe, L. Ribeiro, R. Machado, A. Costa, G. G. Azzi, J. S. Bezerra, and L. M. Rodrigues. On the essence of parallel independence for

the double-pushout and sesqui-pushout approaches. In *Graph Transformation, Specifications, and Nets - In Memory of Hartmut Ehrig*, volume 10800 of *LNCS*, pages 1–18. Springer, 2018.
10. R. Echahed and J. Janodet. Parallel admissible graph rewriting. In *Recent Trends in Algebraic Development Techniques, 13th International Workshop WADT'98, Selected Papers*, volume 1589 of *LNCS*, pages 122–137. Springer, 1999.
11. R. Echahed and A. Maignan. Parallel graph rewriting with overlapping rules. *CoRR*, (abs/1701.06790), 2017.
12. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.
13. H. Ehrig and H. Kreowski. Parallelism of manipulations in multidimensional information structures. In *Mathematical Foundations of Computer Science*, volume 45 of *LNCS*, pages 284–293. Springer, 1976.
14. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 3: Concurrency, Parallelism and Distribution*. World Scientific, 1999.
15. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.
16. J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 1–94, 1997.
17. S. T. R. Group. *The Clean Home Page*. Radboud University, Nijmegen.
18. K. C. II and A. Lindenmayer. Parallel graph generating and recurrence systems for multicellular development. *International Journal of General Systems*, 3(1):53–66, 1976.
19. D. Janssens, G. Rozenberg, and R. Verraedt. On sequential and parallel node-rewriting graph grammars. *Computer Graphics and Image Processing*, 18(3):279–304, 1982.
20. O. Kniemeyer, G. Barczik, R. Hemmerling, and W. Kurth. Relational growth grammars - A parallel graph transformation approach with applications in biology and architecture. In *Third International Symposium AGTIVE, Revised Selected and Invited Papers*, pages 152–167, 2007.
21. H. Kreowski and S. Kuske. Graph multiset transformation: a new framework for massively parallel computation inspired by DNA computing. *Natural Computing*, 10(2):961–986, 2011.
22. H. Kreowski, S. Kuske, and A. Lye. A simple notion of parallel graph transformation and its perspectives. In *Graph Transformation, Specifications, and Nets - In Memory of Hartmut Ehrig*, volume 10800 of *LNCS*, pages 61–82. Springer, 2018.
23. M. Löwe. Characterisation of parallel independence in AGREE-rewriting. In *11th ICGT*, volume 10887 of *LNCS*, pages 118–133. Springer, 2018.
24. R. Plasmeijer and M. V. Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1993.
25. G. Taentzer. Parallel high-level replacement systems. *TCS: Theoretical Computer Science*, 186:43–81, 1997.
26. S. Wolfram. *A new kind of science*. Wolfram-Media, 2002.

Connecting Constrained Constructor Patterns and Matching Logic

Xiaohong Chen[1], Dorel Lucanu[2], and Grigore Roșu[1]

[1] University of Illinois at Urbana-Champaign, USA
[2] Alexandru Ioan Cuza University of Iași, Romania
{xc3,grosu}@illinois.edu, dlucanu@info.uaic.ro

Abstract. Constrained constructor patterns are built from pairs of constructor term patterns and a constraint expressed by a quantifier-free first order logic formula, using the conjunction and disjunction connector. They are used to express state predicates for reachability logic defined over rewrite theories. Matching logic has been recently proposed as a unifying foundation for programming languages, specification and verification. It has been shown to capture several logics important for programming languages, including first-order logic with fixpoints and order-sorted algebra. In this paper we investigate the relationship between the language of the constrained constructor patterns and matching logic. The results we obtained from this comparison bring a mutual benefit for the two approaches. The matching logic can borrow computationally efficient proofs for some equivalences, and the language of the constrained constructor patterns can get a more logical flavor and more expressivity.

1 Introduction

The subject of this paper is inspired by a comment given by José Meseguer in a private message: "I strongly conjecture that there is a deep connection between matching logic and the constrained constructor patterns. It would be great to better understand the details of such a connection."

Constrained constructor patterns are the bricks of the *rewrite-theory-generic* reachability logic framework [11], by which we mean that the reachability logic framework as considered in [11] is parametric in the underlying rewriting theory. The order-sorted specifications $(\Sigma, E \cup B)$, used as support for rewrite theories, consist of an order-sorted signature Σ, a set of particular equations B used to reason modulo B, and a set of equations E that can be turned into a set of rewrite rules $\overrightarrow{E}$ convergent modulo B, assuming that the theory $(\Sigma, E \cup B)$ is sufficiently complete [9]. In this paper, we work under the assumptions that ensure all the properties mentioned below (now we implicitly assume them). The definition of constrained constructor patterns is based on the strong relationship between the initial $(\Sigma, E \cup B)$-algebra $T_{\Sigma/E\cup B}$ and its canonical constructor $(\Omega, E_\Omega \cup B_\Omega)$-algebra $C_{\Omega/E_\Omega,B_\Omega}$. This relationship is briefly explained as follows: 1. $T_{\Sigma/E\cup B}$ is isomorphic to the canonical term-algebra $C_{\Sigma/E,B}$, consisting

of B-equivalence classes of $\overrightarrow{E}$-irreducible-modulo-B Σ-terms; 2. $\Omega \subseteq \Sigma$ is the subsignature of constructors; 3. $C_{\Sigma/E,B}|_\Omega = C_{\Omega/E_\Omega,B_\Omega}$. A constrained constructor pattern predicate is a pair $u|\varphi$, where u is a constructor term pattern and φ is a quantifier-free first-order logic (FOL) formula. The set of constrained constructor patterns includes the constrained constructor pattern predicates and is closed under conjunction and disjunction. The semantics defined by $u|\varphi$ is given by the subset of states $[\![u|\varphi]\!] \subseteq C_{\Omega/E_\Omega,B_\Omega}$ matching u, i.e., for each $a \in [\![u|\varphi]\!]$ there is a valuation ρ such that φ holds (written $\rho \vDash \varphi$) and $a = u\rho$.

There are several additional operations over constrained constructor patterns required to express reachability properties and to support their verification in a computational efficient way. These include (parameterized) subsumption, over-approximation of the complements, and parameterized intersections. The definitions of these operations exploits the cases when the matching and unification modulo $E \cup B$ can be efficiently solved, using, e.g., the theory of variants [4, 7].

Matching Logic (ML) [10, 3, 2] is a variant of first-order logic (FOL) with fixpoints that makes no distinction between functions and predicates. It uses instead symbols and application to uniformly build patterns that can represent static structures and logical constraints at the same time. Semantically, ML patterns are interpreted as the sets of elements that match them. The functional interpretation is obtained by adding axioms like $\exists y.\mathtt{s}\, x = y$ that forces the pattern $\mathtt{s}\, x$ to be evaluated to a singleton. The conjunction and disjunction are interpreted as the intersection, respectively union. For instance, the ML pattern $\exists x{:}Nat.\, \mathtt{s}\, x \wedge (x = 2 \vee x = 5)$, when interpreted over the natural numbers, denotes the set $\{3, 6\}$ since $\mathtt{s}\, x$ is matched by the successor of x, constants 2 and 5 are matched by the numbers 2 and 5, respectively, and $x = n$ is a "predicate": it matches either the entire carrier set when x and n are matched by the same elements, or otherwise the empty set.

The main **contribution** of the paper is an insightful comparison of constrained constructor patterns and matching logic. Since order-sorted algebras can be captured in matching logic [2], we were tempted to think that this comparison is a natural one, because a constrained constructor pattern $u|\varphi$ can be seen as a special ML pattern $u \wedge \varphi$. When we started to formalize this intuition, we realized a few interesting challenges that we need to address:

- How to capture the logical reasoning modulo equations in B in ML?
- How to formalize the canonical model containing only constructor terms?
- What properties does the ML model corresponding to an OSA canonical model have?
- Which are the most suitable ML patterns that capture constrained constructor pattern operations?
- How to express the equivalence between a constrained constructor pattern and its ML encoding?

In order to better understand the relationship between the two approaches, we consider a running example, the QLOCK mutual exclusion protocol [5, 11], and show how to define it in ML. This example gives us a better view of the specificity of ML axioms and how the OSA canonical model is reflected in ML. In this paper,

we only consider the static structure of QLOCK. Since the ML axiomatization includes the complete specifications of natural numbers, (finite) list and (finite) multisets, and it specifies their carrier sets using least fixpoints, we can derive from the specifications an induction proof principle for them.

Structure of the paper. We define constrained constructor patterns and introduce the QLOCK example in Section 2. In Section 3, we introduce matching logic (ML) in details, as it was recently proposed. In Section 4 we discuss the axiomatization of free constructors and the encoding of OSA in ML, and a complete specification of the QLOCK configurations. In Section 5, we show the ML encoding of the constrained constructor patterns and their operations, which is our main contribution. We conclude in Section 6.

2 Constrained Constructor Patterns

We assume the readers are familiar with order-sorted equational and first-order logics (see, e.g., [8]). Here we briefly recall the definitions of constructor pattern predicates [11].

Definition 1. *An* order-sorted signature $\Sigma = (S, \leq, F)$ *contains a sort set* S*, a partial ordering* $\leq \subseteq S \times S$ *called* subsorting*, and a function (family) set* $F = \{F_{s_1 \ldots s_n, s}\}_{s_1,\ldots,s_n,s \in S}$*. We allow* subsort overloading*, i.e.,* $f \in F_{s_1 \ldots s_n, s} \cap F_{s'_1 \ldots s'_n, s'}$ *with* $s_1 \leq s'_1, \ldots, s_n \leq s'_n, s \leq s'$*. An* order-sorted algebra $A = (\{A_s\}_{s \in S}, \{f_A\}_{f \in F})$ *contains (1) a nonempty carrier set* A_s *for every* $s \in S$*; we require* $A_s \subseteq A_{s'}$ *whenever* $s \leq s'$*; and (2) a function interpretation* $f \colon M_{s_1} \times \cdots \times M_{s_n} \to M_s$ *for every* $f \in F_{s_1 \ldots s_n, s}$*. Note that overloaded functions must coincide on the overlapped parts.*

A function $f \in F_{s_1 \ldots s_n, s}$ is denoted as $f : s_1 \times \cdots \times s_n \to s$. Let $X = \{X_s\}_{s \in S}$ be an S-indexed set of *sorted variables* denoted $x{:}s, y{:}s$. We use $T_\Sigma(X)$ to denote the *Σ-term algebra on X*, whose elements are (ground and non-ground) terms. We use $T_\Sigma = T_\Sigma(\emptyset)$ to denote the Σ-algebra of ground terms.

An (equational) *order-sorted theory* $(\Sigma, B \cup E)$ consists of an order-sorted signature Σ and a union set $B \cup E$ of (possibly conditional) Σ-equations (explained below). We assume that $F = \Omega \cup \Delta$, where Ω contains *constructors* and Δ contains *defined functions*. We assume that B contains a special class of axioms that usually express properties like associativity, commutativity, and identity of functions in Σ. Let $B_\Omega \cup E_\Omega$ be the axioms (equations) that only contain constructors in Ω. Then, $(B \setminus B_\Omega) \cup (E \setminus E_\Omega)$ is the set of axioms (equations) that specify defined functions in Δ.

Given $(\Sigma, B \cup E)$, its *initial model* is isomorphic to the *canonical term algebra* $C_{\Sigma/E,B}$ that contains $(B_\Omega \cup E_\Omega)$-equivalence classes of ground Ω-terms. For a ground Σ-term t that may contain defined functions, we let $canf(t) = [u]_{B_\Omega \cup E_\Omega}$ denote its *canonical form* in $C_{\Sigma/E,B}$, i.e., $u =_{B \cup E} t$ and $u \in T_\Omega$. Let $\rho \colon X \to T_\Omega$ be a valuation. We define its extension $\_\rho \colon T_\Sigma \to T_\Omega$ in the usual way.

Given $s \in S$, an *s-sorted constrained constructor pattern* is an expression $u|\varphi$, where $u \in T_\Omega(X)$ has sort s and φ is a quantifier-free Σ-formula; see [11, pp. 204]. The set of *constrained constructor pattern predicates*, denoted $PatPred(\Omega, \Sigma)$,

is the smallest set that includes $\bot$ and constrained constructor patterns, and is closed under disjunction and conjunction. The *semantics* of a constrained constructor pattern predicate A is the set $[\![A]\!]_C$ of canonical terms that *satisfies* it:

$$[\![\bot]\!]_C = \emptyset \qquad [\![A \vee B]\!]_C = [\![A]\!]_C \cup [\![B]\!]_C \qquad [\![A \wedge B]\!]_C = [\![A]\!]_C \cap [\![B]\!]_C$$
$$[\![u|\varphi]\!]_C = \{canf(u\rho) \mid \rho\colon X \to T_\Omega, C_{\Sigma/E,B} \models \varphi\rho\}$$

2.1 A Running Example: QLOCK

QLOCK is a mutual exclusion protocol [5] that allows an unbounded number of (numbered) processes that are in one of the three states: "normal" (doing their own things), "waiting" for a resource, and "critical" when using the resource. A QLOCK state is a tuple $\langle n|w|c|q\rangle$ where n, w, c are multisets of identities of the processes that are in "normal", "waiting", and "critical" states, respectively, and q is the waiting queue, i.e., an associative list. In this paper, we are only interested in understanding how constrained constructor patterns express state predicates, so we only consider the static structure of QLOCK states, whose OSA specification [11] is given below:

$S = \{Nat, List, MSet, NeMSet, Conf, State, Pred\}$
$\leq = \{Nat < List, Nat < NeMSet < MSet\} \cup =_S$
Σ_Ω (constructors):
  $\mathbb{0} : \to Nat$, $\mathbb{s}_- : Nat \to Nat$
  $nil : \to List$, $\_;\_ : List \times List \to List$
  $empty : \to MSet$, $\_\_ : MSet \times MSet \to MSet$,
  $\_\_ : NeMSet \times NeMSet \to NeMSet$
  $\_|\_|\_|\_ : MSet \times MSet \times MSet \times List \to Conf$
  $\langle\_\rangle : Conf \to State$
  $tt : \to Pred, ff : \to Pred$
$\Sigma(\text{QLOCK}) = \Sigma_\Omega \cup \{dupl : MSet \to Pred, dupl : NeMSet \to Pred\}$
B_Ω:
  associativity for list concatenation $\_;\_$ with the identity nil
  associativity/commutativity for multiset union $\_;\_$ with the identity $empty$
$E_\Omega = \emptyset$
$E = \{dupl(s\,u\,u) = tt\}$, where s is any multiset (could be empty).

The corresponding canonical model, denoted QLK, is given as:

$\mathsf{QLK}_{Nat} = \{\mathbb{0}, \mathbb{s}\,\mathbb{0}, \mathbb{s}^2\,\mathbb{0}, \ldots\}$
$\mathsf{QLK}_{List} = \mathsf{QLK}_{Nat} \cup \{nil\} \cup \{n_1; \ldots; n_k \mid n_i \in \mathsf{QLK}_{Nat}, 1 \leq i \leq k, k \geq 2\}$
$\mathsf{QLK}_{NeMSet} = Nat \cup \{[n_1, \ldots, n_k] \mid n_i \in \mathsf{QLK}_{Nat}, 1 \leq i \leq k, k \geq 2\}$
$\mathsf{QLK}_{MSet} = \mathsf{QLK}_{NeMSet} \cup \{empty\}$
$\mathsf{QLK}_{Conf} = \{x_1|x_2|x_3|y \mid x_1, x_2, x_3 \in \mathsf{QLK}_{MSet}, y \in \mathsf{QLK}_{List}\}$
$\mathsf{QLK}_{State} = \{\langle x\rangle \mid x \in \mathsf{QLK}_{Conf}\}$
$\mathsf{QLK}_{Pred} = \{tt, ff\}$

An example of a constrained constructor pattern predicate is $\langle n|w|c|q\rangle|dupl(n\,w\,c) \neq tt$, since no process can be waiting and critical at the same time.

3 Matching Logic

We give a compact introduction to matching logic (ML) syntax and semantics, and the important mathematical instruments that can be defined as theories and/or notations. For full details, we refer readers to [10, 3, 2].

3.1 Matching Logic Syntax and Semantics

ML is an unsorted logic whose formulas, called *patterns*, are constructed from constant symbols, two sets of variables (explained below), propositional constructs $\bot$ and $\to$, a binary application function, the FOL-style existential quantifier $\exists$, and the least fixpoint operator μ. In models, patterns are interpreted as the *sets* of elements that *match* them. Important mathematical instruments and structures, as well as various logical systems can be captured in ML.

Definition 2. *We assume two countably infinite sets of variables EV and SV, where EV is the set of* element variables *denoted $x, y, \ldots$ and SV is the set of* set variables *denoted $X, Y, \ldots$. Given an (at most) countable set of constant symbols Σ, the set of Σ-patterns, written* Pattern*, is inductively generated by the following grammar for every $\sigma \in \Sigma$, $x \in EV$, and $X \in SV$:*

$$\varphi ::= \sigma \mid x \mid X \mid \varphi_1\, \varphi_2 \mid \bot \mid \varphi_1 \to \varphi_2 \mid \exists x.\, \varphi \mid \mu X.\, \varphi$$

where in $\mu X.\, \varphi$ we require that φ is positive in X, i.e., X is not nested in an odd number of times on the left-hand side of an implication $\varphi_1 \to \varphi_2$. This syntactic requirement is to make sure that φ is monotone with respect to the set X, and thus the least fixpoint denoted by $\mu X.\, \varphi$ exists.

Both $\exists$ and μ are binders, and we assume the standard notions of free variables, α-equivalence, and capture-avoiding substitution. Specifically, we use $FV(\varphi)$ to denote the set of (element and set) variables that occur free in φ. We regard α-equivalent patterns as syntactically identical. We write $\varphi[\psi/x]$ (resp. $\varphi[\psi/X]$) for the result of substituting ψ for x (resp. X) in φ, where bound variables are implicitly renamed to prevent variable capturing. We define the following logical constructs as syntactic sugar:

$$\neg\varphi \equiv \varphi \to \bot \quad \varphi_1 \vee \varphi_2 \equiv \neg\varphi_1 \to \varphi_2 \quad \varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$$
$$\top \equiv \neg\bot \qquad \forall x.\, \varphi \equiv \neg\exists x.\, \neg\varphi \qquad \nu X.\, \varphi \equiv \neg\mu X.\, \neg\varphi[\neg X/X]$$

We assume the standard precedence between logical constructs and that application $\varphi_1\, \varphi_2$ binds the tightest. We abbreviate the sequential application $(\cdots((\varphi_1\, \varphi_2)\, \varphi_3)\, \cdots\, \varphi_n)$ as $\varphi_1\, \varphi_2\, \varphi_3\, \cdots\, \varphi_n$.

ML has a *pattern matching* semantics where patterns are interpreted in models as the *sets* of elements that *match* them.

Definition 3. *Given a symbol set Σ, a* Σ-model $(M, \_\bullet\_, \{\sigma_M\}_{\sigma\in\Sigma})$ *contains:*
- *M: a nonempty carrier set;*
- *$\_\bullet\_ : M \times M \to \mathcal{P}(M)$ as the interpretation of application, where $\mathcal{P}(M)$ is the powerset of M;*
- *$\sigma_M \subseteq M$: a subset of M as the* interpretation *of $\sigma \in \Sigma$.*

By abuse of notation, we write M for the above model.

For notational simplicity, we extend $\_\bullet\_$ from over elements to over sets, *pointwisely*, as follows:

$$\_\bullet\_ : \mathcal{P}(M) \times \mathcal{P}(M) \to \mathcal{P}(M) \quad A \bullet B = \bigcup_{a \in A, b \in B} a \bullet b \text{ for } A, B \subseteq M$$

Note that $\emptyset \bullet A = A \bullet \emptyset = \emptyset$ for any $A \subseteq M$.

Definition 4. *Given a symbol set Σ and a Σ-model M, an M-valuation ρ: $(EV \cup SV) \to (M \cup \mathcal{P}(M))$ is a function that maps element variables to elements of M and set variables to subsets of M, i.e., $\rho(x) \in M$ and $\rho(X) \subseteq M$ for every $x \in EV$ and $X \in SV$. We extend ρ from over variables to over patterns, denoted $\bar{\rho}$: Pattern $\to \mathcal{P}(M)$, as follows:*

$$\bar{\rho}(x) = \{\rho(x)\} \quad \bar{\rho}(X) = \rho(X) \quad \bar{\rho}(\sigma) = \sigma_M \quad \bar{\rho}(\bot) = \emptyset \quad \bar{\rho}(\varphi_1\, \varphi_2) = \bar{\rho}(\varphi_1) \bullet \bar{\rho}(\varphi_2)$$

$$\bar{\rho}(\varphi_1 \to \varphi_2) = M \setminus (\bar{\rho}(\varphi_1) \setminus \bar{\rho}(\varphi_2)) \quad \bar{\rho}(\exists x.\, \varphi) = \bigcup_{a \in M} \overline{\rho[a/x]}(\varphi) \quad \bar{\rho}(\mu X.\, \varphi) = \mu \mathcal{F}^{\rho}_{X,\varphi}$$

where $\mathcal{F}^{\rho}_{X,\varphi} : \mathcal{P}(M) \to \mathcal{P}(M)$ is a monotone function defined as $\mathcal{F}^{\rho}_{X,\varphi}(A) = \overline{\rho[A/X]}(\varphi)$ for $A \subseteq M$, and $\mu \mathcal{F}^{\rho}_{X,\varphi}$ denotes its unique least fixpoint given by the Knaster-Tarski fixpoint theorem [12].

Definition 5. *Given M and φ, we say M* satisfies *φ, written $M \vDash \varphi$, iff $\bar{\rho}(\varphi) = M$ for all ρ. Given $\Gamma \subseteq$ Pattern, we say M* satisfies *Γ, written $M \vDash \Gamma$, iff $\bar{\rho}(\varphi) = M$ for all ρ and $\varphi \in \Gamma$. We call Γ a* theory *and patterns in Γ* axioms.

3.2 Important Mathematical Instruments

Several mathematical instruments of practical importance, such as definedness, totality, equality, membership, set containment, functions and partial functions, constructors, and sorts can all be defined using patterns. We give a compact summary of their definitions in ML and introduce proper notations for them.

Definedness Symbol and Axiom. ML patterns are interpreted as subsets of M. This is different from the classic FOL, whose formulas evaluate to either true or false. However, it is easy to restore the classic two-value semantics in ML, by using M, the entire carrier set, to represent the logical true, and $\emptyset$, the empty set, to represent the logical false. Since M is nonempty, no confusion is possible. We call φ a *predicate* in M if $\bar{\rho}(\varphi) \in \{\emptyset, M\}$ for all ρ. In the following, we define a set of predicate patterns that represent the important mathematical instruments. These patterns are constructed from a special symbol called *definedness*.

Definition 6. *Let $\lceil \_ \rceil$ be a symbol, which we call the* definedness *symbol. We write $\lceil \varphi \rceil$ instead of $\lceil \_ \rceil\, \varphi$. Let* (Definedness) *be the axiom $\forall x. \lceil x \rceil$. We define the following important notations:*

$$\textit{totality } \lfloor \varphi \rfloor \equiv \neg \lceil \neg \varphi \rceil \qquad \textit{equality } \varphi_1 = \varphi_2 \equiv \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor$$

$$\textit{membership } x \in \varphi \equiv \lceil x \wedge \varphi \rceil \quad \textit{inclusion } \varphi_1 \subseteq \varphi_2 \equiv \lfloor \varphi_1 \to \varphi_2 \rfloor$$

We also define their negations:

$$\varphi_1 \neq \varphi_2 \equiv \neg(\varphi_1 = \varphi_2) \quad x \notin \varphi \equiv \neg(x \in \varphi) \quad \varphi_1 \not\subseteq \varphi_2 \equiv \neg(\varphi_1 \subseteq \varphi_2)$$

In the following, when we say that we consider a theory Γ that contains certain axioms, we implicitly assume that the symbol set contains all symbols that occur in those axioms.

Sorts. ML is an unsorted logic and has no built-in support for sorts or many-sorted functions. However, we can define sorts as constant symbols and use patterns to axiomatize their properties. Specifically, for every sort s, we define a corresponding constant symbol also denoted s that represents its sort name. For technical convenience, we include the following axiom

$$\text{(Sort Name)} \quad \exists x.\, s = x$$

to specify that s is matched by exactly one element, which is the name of the sort s. To get the carrier set of s, we define a symbol $[\![\_]\!]$, which we call the *inhabitant* symbol, and we write $[\![\varphi]\!]$ instead of $[\![\_]\!]\,\varphi$. The intuition is that $[\![s]\!]$ is matched by exactly the elements that have sort s, i.e., it represents the carrier set of s. We also include a symbol *Sort* that is matched by all sort names, by including an axiom $s \in \mathit{Sort}$.

We can specify properties about sorts by patterns. E.g., the following axiom

$$\text{(Nonempty Inhabitant)} \quad [\![s]\!] \neq \bot$$

specifies that the carrier set of s is nonempty. The following axiom

$$\text{(Subsort)} \quad [\![s_1]\!] \subseteq [\![s_2]\!]$$

specifies that the carrier set of s_1 is a subset of that of s_2, i.e., s_1 is a *subsort* of s_2. We define *sorted negation* $\neg_s \varphi \equiv (\neg\varphi) \wedge [\![s]\!]$, which is matched by all elements of sort s that do not match φ. We define *sorted quantification* that restricts the ranges of $x, x_1, \ldots, x_n$ in the quantification:

$$\forall x{:}s.\, \varphi \equiv \forall x.\, x \in [\![s]\!] \to \varphi \qquad \forall x_1, \ldots, x_n{:}s.\, \varphi \equiv \forall x_1{:}s.\ \ldots \forall x_n{:}s.\, \varphi$$
$$\exists x{:}s.\, \varphi \equiv \forall x.\, x \in [\![s]\!] \wedge \varphi \qquad \exists x_1, \ldots, x_n{:}s.\, \varphi \equiv \exists x_1{:}s.\ \ldots \exists x_n{:}s.\, \varphi$$

We can specify sorting restrictions of symbols. For example:

$$\text{(Sorted Symbol)} \quad \sigma\, [\![s_1]\!] \cdots [\![s_n]\!] \subseteq [\![s]\!]$$

requires $\sigma\, x_1 \cdots x_n$ to have sort s, given that $x_1, \ldots, x_n$ have sorts $s_1, \ldots, s_n$, respectively. For notational simplicity, we write $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ to mean that we assume the axiom (Sorted Symbol) for σ.

Functions and Partial Functions. ML symbols are interpreted as relations, when they are applied to arguments. Indeed, $\sigma\, x_1 \cdots x_n$ is a pattern that can be matched zero, one, or more elements. In practice, we often want to specify that σ is a function (or partial function), in the sense that $\sigma\, x_1 \cdots x_n$ can be matched by exactly one (or at most one) element. That can be specified by the following axioms, respectively:

$$\text{(Function)} \qquad \forall x_1{:}s_1.\ \ldots \forall x_n{:}s_n.\, \exists y{:}s.\, \sigma(x_1, \ldots, x_n) = y$$
$$\text{(Partial Function)} \quad \forall x_1{:}s_1.\ \ldots \forall x_n{:}s_n.\, \exists y{:}s.\, \sigma(x_1, \ldots, x_n) \subseteq y$$

Recall that y is an element variable, so it is matched by exactly one element. For notational simplicity, we use the function notation $\sigma\colon s_1 \times \cdots \times s_n \to s$ to mean that we assume the axiom (Function) for σ. Similarly, we use the partial function notation $\sigma\colon s_1 \times \cdots \times s_n \rightharpoonup s$ to mean that we assume the axiom (Partial Function) for σ.

Constructors. *Constructors* are extensively used in building programs and data, as well as semantic structures to define and reason about languages and programs. They can be characterized in the "no junk, no confusion" spirit [6].[3] Specifically, let *Term* be a sort of *terms* and Σ be a set of constructors denoted c. We associate an arity $n_c \geq 0$ with every c. Consider the following axioms:

(Function, for all c) $\quad c\colon \underbrace{\mathit{Term} \times \cdots \times \mathit{Term}}_{n_c \text{ times}} \to \mathit{Term}$

(No Junk) $\quad \bigvee_{c \in C} \exists x_1, \ldots, x_{n_c} \colon \mathit{Term}.\, c\, x_1 \cdots x_{n_c}$

(No Confusion I, for all $c \neq c'$)

$$\forall x_1, \ldots, x_{n_c} \colon \mathit{Term}.\, \forall y_1, \ldots, y_{n_{c'}} \colon \mathit{Term}.\, \neg \left(c\, x_1 \cdots x_{n_c} \wedge c'\, y_1 \cdots y_{n_{c'}} \right)$$

(No Confusion II, for all c)

$$\forall x_1, \ldots, x_{n_c} \colon \mathit{Term}.\, \forall y_1, \ldots, y_{n_c} \colon \mathit{Term}.$$
$$(c\, x_1 \cdots x_{n_c} \wedge c\, y_1 \cdots y_{n_c}) \to c\, (x_1 \wedge y_1) \cdots (x_{n_c} \wedge y_{n_c})$$

(Inductive Domain) $\quad \mu T.\, \bigvee_{c \in C} c\, \underbrace{T \cdots T}_{n_i \text{ times}}$

Intuitively, (No Confusion I) says different constructs build different things; (No Confusion II) says constructors are injective; and (Inductive Domain) says the carrier set of *Term* is the smallest set that is closed under all constructors. We refer to the first two axioms as (No Confusion). Technically, (No Junk) is not necessary as it is implied by (Inductive Domain).

4 Encoding Order-Sorted Algebras

As seen in Section 3.2, the subset relation between the carrier sets of sorts can be captured in ML by patterns. Therefore, OSA and subsorting can be naturally captured in ML; see [2] for details. Specifically, to capture OSA, we define for every sorts $s \in S$ a corresponding sort, also denoted s, in ML. For every $s \leq s'$, we include a subsorting axiom $[\![s]\!] \subseteq [\![s']\!]$. We define for every OSA function $f \in F_{s_1 \ldots s_n, s}$ a corresponding symbol, also denoted f, and include the (Function) axiom, i.e., $f\colon s_1 \times \cdots \times s_n \to s$. This is summarized in Figure 1.

Let $\Sigma = (S, \leq, F)$ be an order-sorted signature and Σ^{ML} be the corresponding ML signature. Let $A = (\{A_s\}_{s \in S}, \{f_A\}_{f \in F})$ be an OSA. We define its derived ML Σ^{ML}-model, denoted A^{ML}, as in [2], which includes the standard interpretations of the definedness and inhabitant symbols, sorts, functions, and elements in A.

Theorem 1 ([2]). *For every formula φ, we have $A^{\mathsf{ML}} \models \varphi^{\mathsf{ML}}$ iff $A \models \varphi$.*

[3] This answers a question asked by Jacques Carette on the *mathoverflow* site (https://mathoverflow.net/questions/16180/formalizing-no-junk-no-confusion) ten years ago: Are there logics in which these requirements ("no junk, no confusion") can be internalized?

	Order-Sorted Algebra	**Matching Logic**
Signature	$\Sigma = (S, \leq, F)$	$\Sigma^{\mathsf{ML}} = \{\lceil \_ \rceil, [\![\_]\!], Sort\} \cup S \cup F$
Axioms	OSA metalanguage	ML axioms
	$s \in S$	$s \in Sort$ $\exists y.\, s = y$ $[\![s]\!] \neq \bot$
	$s \leq s'$	$[\![s]\!] \subseteq [\![s']\!]$
	$f \in F_{s_1 \dots s_n, s}$	$f \colon s_1 \times \cdots \times s_n \to s$
	$x{:}s$ (sorted variable)	$x \in [\![s]\!]$
Terms	t	t^{ML}
	$f(t_1, \dots, t_n)$	$f\, t_1 \cdots t_n$
Sentences	φ	φ^{ML}
	$\{x_1, \dots, x_n\}$ = variables in φ	$x_1 \in [\![s_1]\!] \wedge \cdots \wedge x_n \in [\![s_n]\!] \to (\varphi = \top)$
Model	A	$M \equiv A^{\mathsf{ML}}$
	$f_A : A_{s_1} \times \cdots A_{s_n} \to A_s$ $f_A(a_1, \dots, a_n)$	$f_M\, a_1 \cdots a_n = \{f_A(a_1, \dots, a_n)\}$

Fig. 1. Given an order-sorted signature $\Sigma = (S, \leq, F)$ and a Σ-OSA A, we derive a ML signature Σ^{ML} and a corresponding Σ^{ML}-model $M \equiv A^{\mathsf{ML}}$.

4.1 QLOCK Example in ML

We have shown the OSA specification of QLOCK's static structures in Section 2.1 and the ML encoding of OSA in Section 4. Putting them together, we get an ML specification for QLOCK, which we show below in full details.

Notations

- $\overline{x}$: a syntactic sugar for $x_1, \dots, x_n$
- $\forall \overline{x}{:}\overline{s}$: a syntactic sugar for $\forall x_1{:}s_1.\ \dots \forall x_n{:}s_n$, where we assume $\bar{x}$ and $\bar{s}$ have the same length n.

ML Signature $\Sigma(\text{QLOCK})^{\mathsf{ML}}$ contains the following symbols (we remind readers of the mathematical instruments defined in Section 3.2):

- a definedness symbol $\lceil \_ \rceil$;
- an inhabitant symbol $[\![\_]\!]$;
- a symbol S for sort names;
- a symbol for each sort: *Nat*, *List*, *MSet*, *NeMSet*, *Conf*, *State*, *Pred*;
- a symbol for each function: *nil*, *conc*, *union*, *conf*, *state*, *dupl*, $\mathbb{0}$, $\mathbb{s}$;

ML Axioms Γ^{QLOCK} includes the (Definedness) axiom (see Definition 6) and the following axioms:

ML axioms for sort names

- the sort symbols are functional constants:

$$\exists y.\, y = Nat \quad \exists y.\, y = List \quad \exists y.\, y = MSet \quad \exists y.\, y = NeMSet$$
$$\exists y.\, y = Conf \quad \exists y.\, y = State \quad \exists y.\, y = Pred$$

- S is the set of sorts:

$$S = Nat \vee List \vee MSet \vee NeMSet \vee Conf \vee State \vee Pred$$

– for each sort $s \in S$, its carrier set is non-empty:

$$\forall s{:}S.\, [\![s]\!] \neq \bot$$

ML axioms for the natural numbers

– the constructors are functional:

$$\exists y{:}Nat.\, y = \mathbb{0} \qquad \forall x{:}Nat.\, \exists y{:}Nat.\, y = \mathbb{s}\, x$$

– "no confusion" axioms:

$$\forall x{:}Nat.\, \neg(\mathbb{0} \wedge \mathbb{s}\, x) \qquad \forall x, y{:}Nat.\, \mathbb{s}\, x \wedge \mathbb{s}\, y \rightarrow \mathbb{s}(x \wedge y)$$

– the domain of *Nat* is the smallest set that is closed under $\mathbb{0}$ and $\mathbb{s}$:

$$[\![Nat]\!] = \mu\, X.\, \mathbb{0} \vee \mathbb{s}(X)$$

There is no need to add the "no junk" axiom $[\![Nat]\!] = \mathbb{0} \vee \mathbb{s}\, [\![Nat]\!]$ as it is a consequence of the above axiom.

Remark. Note that we use the sorted quantification in the above functional axioms. In other words, we only specify that $\mathbb{s}$ is a function when it is within the domain of *Nat*. Its behavior outside the domain of *Nat* is *unspecified*. This way, we allow maximal flexibility in terms of modeling, because each model (i.e., implementation) of the specification Γ can decide the behavior of $\mathbb{s}$ outside *Nat*. An "order-sorted-like"model will make $\mathbb{s}\, x$ return $\bot$, the empty set, whenever x is not in *Nat*, while an "error-algebra-like" model will make $\mathbb{s}\, x$ return *error*, a distinguished error element, to denote the "type error". Note that if we do not use the sorted quantification, but use the unsorted version, $\forall x.\, \exists y.\, y = \mathbb{s}\, x$, then we explicitly *exclude* the order-sorted model, which is not what we want.

Remark. We point out that the sorted quantification axioms do not *restrict* $\mathbb{s}$ to be only applicable within *Nat*. The pattern $\mathbb{s}\, x$ when x is outside the domain of *Nat* is still a well-formed pattern, whose semantics is not specified by the theory of natural numbers, but can be specified by other theories. For example, the theory of real numbers may re-use $\mathbb{s}$ and overload it as the increment-by-one function on reals. The theory of bounded arithmetic may re-use and overload $\mathbb{s}$ as the successor "function", which is actually a partial function and is undefined on the maximum value. The theory of transition systems may re-use and overload $\mathbb{s}$ as the successor "function", which is actually the underlying transition relation, and $\mathbb{s}\, x$ yields the set of all next states of the state x. In the last two cases, $\mathbb{s}$ is no longer a function because it is not true that $\mathbb{s}\, x$ always returns one element. Therefore, if we use not the sorted quantification axiom but the unsorted one, we cannot re-use $\mathbb{s}$ in the theories of bounded arithmetic or transition systems, without introducing inconsistency. Thus, by using sorted quantification for $\mathbb{s}$ in the theory of natural numbers, we do not restrict but actually encourage the re-use and overloading of $\mathbb{s}$ in other theories. On the other hand, ML is expressive enough if one wants to allow a restricted use of a symbol. For instance, if we want to restrict the use of $\mathbb{s}$ only to *Nat*, then we can add the axiom $\forall x.\, \lceil \mathbb{s}\, x \rceil \rightarrow x \in [\![Nat]\!]$.

ML axioms for Boolean values *Pred*

– the constructors are functional:

$$\exists y{:}Pred.\, y = tt \qquad \exists y{:}Pred.\, y = ff$$

– "no confusion" axiom: $\neg(tt \wedge ff)$
– the domain of *Pred* consists only of ff and tt:

$$[\![Pred]\!] = ff \vee tt$$

ML axioms for associative lists (over natural numbers)

– the constructors are functional:

$$\forall x,y{:}List.\, \exists z{:}List.\, z = conc\, x\, y \qquad \exists x{:}List.\, x = nil$$

– the associativity axiom:

$$\forall x,y,z{:}List.\, conc(conc\, x\, y)\, z = conc\, x\, (conc\, y\, z)$$

– the unity axioms:

$$\forall x{:}List.\, conc\, x\, nil = x \qquad \forall x{:}List.\, conc\, nil\, x = x$$

– the domain of *List* is the smallest set that includes $[\![Nat]\!]$ and closed under *conc* and *nil*:

$$[\![List]\!] = \mu X.\, [\![Nat]\!] \vee nil \vee conc\, X\, X$$

There is no need to add the subsort axiom $[\![Nat]\!] \subseteq [\![List]\!]$ to Γ since it is a consequence of the above axiom.

ML axioms for multisets (over natural numbers)

– the constructors are functional:

$$\exists y{:}MSet.\, y = empty \qquad \forall x,y{:}MSet.\, \exists z{:}MSet.\, z = union\, x\, y$$

$$\forall x,y{:}NeMSet.\, \exists z{:}NeMSet.\, z = union\, x\, y$$

– the associativity axiom:

$$\forall x,y,z{:}MSet.\, union(union\, x\, y)\, z = union\, x\, (union\, y\, z)$$

– the unity and commutativity axioms:

$$\forall x{:}MSet.\, union\, x\, empty = x \qquad \forall x, y{:}MSet.\, union\, x\, y = union\, y\, x$$

– the domain axiom:

$$[\![NeMSet]\!] = \mu\, X.\, [\![Nat]\!] \vee union\, X\, X \quad [\![MSet]\!] = empty \vee [\![NeMSet]\!]$$

The axioms $[\![Nat]\!] \subseteq [\![NeMSet]\!]$ and $[\![NeMSet]\!] \subseteq [\![MSet]\!]$, corresponding to subsorting relations $Nat < NeMSet$ and respectively $NeMSet < MSet$, are not needed, as they are consequences of the above.

ML axioms for configurations

– the constructors are functional:

$$\forall x_1,x_2,x_3{:}MSet.\, \forall y{:}List.\, \exists z{:}Conf.\, conf\, x_1\, x_2\, x_3\, y = z$$

– "no confusion" axiom:
$\forall x_1,x_2,x_3,x_1',x_2',x_3'{:}MSet.\, \forall y,y'{:}List.$

$$conf\, x_1\, x_2\, x_3\, y \wedge conf\, x_1'\, x_2'\, x_3'\, y' \rightarrow conf\, (x_1 \wedge x_1')(x_2 \wedge x_2')(x_3 \wedge x_3')(y \wedge y')$$

– the domain of *Conf* is the set that is closed under *conf*:

$$[\![Conf]\!] = conf\, [\![MSet]\!]\, [\![MSet]\!]\, [\![MSet]\!]\, [\![List]\!]$$

ML axioms for states

– the constructors are functional:

$$\forall x{:}Conf.\, \exists y{:}State.\, state\, x = y$$

– "no confusion" axiom:

$$\forall x,x'{:}Conf.\ state\, x \wedge state\, x' \rightarrow state\, x \wedge x'$$

– the domain of *State* is the set that is closed under *state*:

$$[\![State]\!] = state\, [\![Conf]\!]$$

The specification of the carrier set for the sorts *Nat*, *List*, *MSet*, and *NeMSet* as least fix points allows to formalize in ML of their induction proof principles. In what follows, $\varphi(x)$ says that the pattern φ depends on the variable x.

ML axioms that define *dupl*

We here give the complete specification of *dupl*:

$$\forall x{:}MSet.\ \exists y{:}Pred.\ dupl\, x = y$$
$$\forall s.\ \exists s', u.\ s =_{NeMSet} union\, s'(union\, u\, u) \rightarrow dupl\, s = tt$$
$$\forall s.\ \forall s', u.\ s \neq_{MSet} union\, s'(union\, u\, u) \rightarrow dupl\, s = ff$$

Proposition 1. $\mathsf{QLK}^{\mathsf{ML}} \models \Gamma^{\mathrm{QLOCK}}$.

Proof. By construction.

In the following, we show that *inductive reasoning* is available in $\mathsf{QLK}^{\mathsf{ML}}$ for natural numbers, (finite) lists, and (finite) multisets. We write $\varphi(x)$ to mean a pattern φ with a distinguished variable x and write $\varphi(t)$ to mean $\varphi[t/x]$.

Proposition 2 (Peano Induction).

$$\Gamma^{\mathrm{QLOCK}} \models \varphi(0) \wedge (\forall y{:}Nat.\ \varphi(y) \rightarrow \varphi(\mathbb{s}\, y)) \rightarrow \forall x{:}Nat.\ \varphi(x)$$

Proof. See [2].

Since the specifications for lists and multisets do not include "no confusion" axioms (due to the associativity, commutativity and identity axioms), their induction principles are given only for the ML model generated from the canonical OSA. This is sufficient for the purpose of this paper, because our goal is to show a faithful ML representation of constrained constructor patterns, whose semantics are given in the canonical model.

Proposition 3 (List and Multiset Induction).

$$\mathsf{QLK}^{\mathsf{ML}} \models \varphi(nil) \wedge \forall x{:}Nat.\varphi(x) \wedge (\forall \ell_1,\ell_2{:}List.\varphi(\ell_1)\wedge\varphi(\ell_2)\rightarrow\varphi(conc\, \ell_1\, \ell_2))) \rightarrow \forall \ell{:}List.\ \varphi(\ell)$$

$$\mathsf{QLK}^{\mathsf{ML}} \models \forall x{:}Nat.\varphi(x) \wedge (\forall m_1,m_2{:}NeMSet.\varphi(m_1)\wedge\varphi(m_2)\rightarrow\varphi(union\, m_1\, m_2)) \rightarrow \forall m{:}NeMSet.\ \varphi(m)$$

$$\mathsf{QLK}^{\mathsf{ML}} \models \varphi(empty) \wedge \forall x{:}Nat.\ \varphi(x) \wedge (\forall m_1,m_2{:}MSet.\ \varphi(m_1) \wedge \varphi(m_2) \rightarrow \varphi(union\, m_1\, m_2)) \rightarrow \forall m{:}MSet.\ \varphi(m)$$

Proof. By the inductive principle of the canonical model QLK and Theorem 1.

5 Encoding Constrained Constructor Patterns in ML

Let $(\Sigma, B \cup E)$ be an order-sorted theory with $(\Omega, B_\Omega \cup E_\Omega)$ being its subtheory of constructors. Recall that $C_{\Sigma/E,B}$ denotes the canonical constructor term algebra. Let $(\Sigma^{\mathsf{ML}}, \Gamma^{\Sigma,E,B})$ be the ML translation of $(\Sigma, E \cup B)$ with $\Gamma^{\Sigma,E,B} = B^{\mathsf{ML}} \cup E^{\mathsf{ML}}$, as discussed in Section 4.

Definition 7. *For a constrained constructor pattern $u|\varphi$, its ML translation is the pattern $u^{\mathsf{ML}} \wedge \varphi^{\mathsf{ML}}$. The ML translations of constrained constructor pattern predicates are defined in the expected way, where $\bot$ translates to $\bot$, conjunction translates to conjunction, and disjunction translates to disjunction.*

The canonical model $C_{\Sigma/E,B}$ has a corresponding $(\Sigma^{\mathsf{ML}}, \Gamma^{\Sigma,E,B})$-model $C^{\mathsf{ML}}_{\Sigma/E,B}$ by Theorem 1. For $\rho\colon X \to T_\Omega$ and a FOL formula φ, we have $C_{\Sigma/B,E} \models \varphi\rho$ iff $C^{\mathsf{ML}}_{\Sigma/E,B} \models (\varphi\rho)^{\mathsf{ML}}$ by the same theorem. This allows us to define the semantics of a constrained constructor pattern $[\![u|\varphi]\!]$ as the interpretation of the ML pattern $\exists \overline{x}{:}\overline{s}.\, u^{\mathsf{ML}} \wedge \varphi^{\mathsf{ML}}$ in $C^{\mathsf{ML}}_{\Sigma/E,B}$, where $\overline{x}{:}\overline{s} = FV(u \wedge \varphi)$.

Next we explain in ML terms some of the constrained constructor pattern operations discussed in [11]. We regard a substitution $\sigma \triangleq \{x_1 \mapsto t_1, \ldots, x_n \to t_n\}$ as the ML pattern $\sigma^{\mathsf{ML}} \triangleq x_1 = t_1 \wedge \cdots \wedge x_n = t_n$.

Constrained Constructor Pattern Subsumption. In [11], the following question is asked: When is the constrained constructor pattern $u|\psi$ an instance of a finite family $\{(v_i|\psi_i) \mid i \in I\}$, i.e., $[\![u|\varphi]\!] \subseteq \bigcup_{i\in I} [\![v_i|\psi_i]\!]$? Perhaps, at this level of abstraction, the above question is unclear, because we do not know yet what exactly it means by "when". Let us elaborate it. The constrained constructor patterns are evaluated in the canonical model $C_{\Sigma/E,B}$, so the above question asks when there is a computationally efficient way to decide whether[4]

$$C_{\Sigma/E,B} \models [\![u|\varphi]\!] \subseteq \textstyle\bigcup_{i\in I} [\![v_i|\psi_i]\!]$$

The answer is given by $E_\Omega \cup B_\Omega$-matching. Let $\textsc{match}(u, \{v_i \mid i \in I\})$ denote the set of all the pairs (i, β) with β a substitution such that $u =_{E_\Omega \cup B_\Omega} v_i\beta$, i.e., β matches v_i on u modulo $E_\Omega \cup B_\Omega$. We assume that $u|\psi$ and $\{(v_i|\psi_i) \mid i \in I\}$ do not share variables. Then the constrained constructor pattern subsumption is formally defined as follows:

Definition 8 ([11]). *A family of constrained constructor patterns $\{(v_i|\psi_i) \mid i \in I\}$* subsumes *$u|\varphi$, denoted $u|\varphi \sqsubseteq \{(v_i|\psi_i) \mid i \in I\}$, iff*

$$C_{\Sigma/B,E} \models \varphi \to \textstyle\bigvee_{(i,\beta)\in \textsc{match}(u,\{v_i|i\in I\})} \psi_i\beta.$$

Defined in this way, the constrained constructor pattern subsumption is computationally cheap in some cases; see [11]. One such case for example is when $E = \emptyset$ and Ω consists of associativity or associativity-commutativity and the terms are not too large. Note that $u|\varphi \sqsubseteq \{(v_i|\psi_i) \mid i \in I\}$ implies $[\![u|\varphi]\!] \subseteq \bigcup_{i\in I} [\![v_i|\psi_i]\!]$, but the inverse implication is not always true. The following counterexample is from [11], where a simple "inductive" instantiation of variable m by 0 and

[4] This is an informal notation because $[\![u|\varphi]\!] \subseteq \bigcup_{i\in I} [\![v_i|\psi_i]\!]$ is not exactly a formula.

$s(k)$ can yield a proof by subsumption for the above set inclusion. Formally, let $\langle \_, \_ \rangle$ denote the pairing of natural numbers. Then we have $[\![\langle n, m\rangle|\top]\!] \subseteq [\![\langle x, 0\rangle|\top \vee \langle y, \mathbb{s}(z)\rangle|\top]\!]$, but $\langle n, m\rangle|\top \not\sqsubseteq \langle x, 0\rangle|\top \vee \langle y, \mathbb{s}(z)\rangle|\top$.

Let us discuss the ML counterpart of the subsumption. The ML pattern that corresponds to $[\![u|\varphi]\!] \subseteq \bigcup_{i\in I}[\![(v_i|\psi_i)]\!]$, is

$$\left(\exists \overline{x}{:}\overline{s}.\, u^{\mathsf{ML}} \wedge \varphi^{\mathsf{ML}}\right) \subseteq \left(\bigvee_{i\in I} \exists \overline{y_i}{:}\overline{s_i}.\, v_i^{\mathsf{ML}} \wedge \psi_i^{\mathsf{ML}}\right)$$

where $\overline{x}{:}\overline{s} = FV(u|\varphi)$, and $\overline{y_i}{:}\overline{s_i} = FV(v_i|\psi_i)$. Since the two patterns do not share variables by assumption, the above is a well-formed ML pattern (we remind that $\varphi \subseteq \varphi'$ is the sugar-syntax of the ML pattern $\lfloor \varphi \rightarrow \varphi' \rfloor$).

The ML translation of the definition for $u|\varphi \sqsubseteq \{(v_i|\psi_i) \mid i \in I\}$ is

$$C^{\mathsf{ML}}_{\Sigma/B,E} \models \varphi^{\mathsf{ML}} \rightarrow \bigvee_{(i,\beta)\in \textsc{match}(u,\{v_i|i\in I\})} \left(\psi_i^{\mathsf{ML}} \wedge \beta^{\mathsf{ML}}\right)$$

where β^{ML} is the pattern describing the substitution β. We can prove now that the two ML patterns are equivalent:

Theorem 2.

$$C^{\mathsf{ML}}_{\Sigma/E,B} \models \left(\exists \overline{x}{:}\overline{s}.\, u^{\mathsf{ML}} \wedge \varphi^{\mathsf{ML}}\right) \subseteq \left(\bigvee_{i\in I} \exists \overline{y_i}{:}\overline{s_i}.\, v_i^{\mathsf{ML}} \wedge \psi_i^{\mathsf{ML}}\right) \leftrightarrow \left(\varphi^{\mathsf{ML}} \rightarrow \bigvee_{(i,\beta)\in \textsc{match}(u,\{v_i|i\in I\})} \left(\psi_i^{\mathsf{ML}} \wedge \beta^{\mathsf{ML}}\right)\right)$$

Regarding the counterexample, we show that

$$C^{\mathsf{ML}}_{\Sigma/E,B} \models \exists m{:}Nat.\, \langle n, m\rangle \subseteq \exists x,y,z{:}Nat.\, \langle x, 0\rangle \vee \langle y, \mathbb{s}(z)\rangle \qquad (*)$$

is proved in ML. Consider $\varphi(m) \triangleq \forall n,x,y,z{:}Nat.\, \langle n, m\rangle \subseteq \langle x, 0\rangle \vee \langle y, \mathbb{s}(z)\rangle$ and applying the induction principle for natural numbers, given by Proposition 2, we obtain

$$C^{\mathsf{ML}}_{\Sigma/E,B} \models \forall m{:}Nat.\, \exists n{:}Nat.\, \langle n, m\rangle \subseteq \exists x,y,z{:}Nat.\, \langle x, 0\rangle \vee \langle y, \mathbb{s}(z)\rangle$$

which implies $(*)$.

Over-Approximating Complements. In [11] it is showed that the complement of a constrained constructor pattern cannot be computed using negation, i.e, $[\![u|\top]\!] \setminus [\![u|\varphi]\!] = [\![u|\neg\varphi]\!]$ does not always hold, but the inclusion $[\![u|\top]\!] \setminus [\![u|\varphi]\!] \subseteq [\![u|\neg\varphi]\!]$ holds. Therefore an over-approximation of the difference is defined as:

$$[\![u|\varphi]\!] \setminus\!\setminus [\![u|\psi]\!] \triangleq [\![u|\varphi]\!] \cap [\![u|\neg\psi]\!] \quad (= [\![u|\varphi \wedge \neg\psi]\!])$$

Since ML has negation, the difference $[\![u|\top]\!] \setminus [\![u|\varphi]\!]$ is the same with the interpretation in $C^{\mathsf{ML}}_{\Sigma/E,B}$ of the ML pattern

$$\exists \overline{x}{:}\overline{s}.\, u^{\mathsf{ML}} \wedge \neg(\exists \overline{x}{:}\overline{s}.\, (u^{\mathsf{ML}} \wedge \varphi^{\mathsf{ML}}))$$

The constructor pattern predicate $[\![u|\top]\!]$ is the same with the interpretation in $C^{\mathsf{ML}}_{\Sigma/E,B}$ of the pattern $\exists \overline{x}{:}\overline{s}.\, u^{\mathsf{ML}}$, where $\overline{x}{:}\overline{s}$ is the set of variables occurring in u, and constructor predicate $[\![u|\neg\varphi]\!]$ is the same with the interpretation of $\exists \overline{x}{:}\overline{s}.\, (u^{\mathsf{ML}} \wedge \neg\varphi^{\mathsf{ML}})$.

The counterexample for equality as in [11] is $u \triangleq (x, y, z)$, as a multiset over $\{a, b, c\}$, $\varphi \triangleq x \neq y$. Using ML we may explain why $[\![u|\top]\!] \setminus [\![u|\varphi]\!] = [\![u|\neg\varphi]\!]$ does not hold in a more generic way. We use the notation from the QLOCK example. Apparently, the interpretations of $\exists x,y,z{:}MSet.\,(union\,x\,y\,z) \wedge x \neq y$ and $\exists x,y,z{:}MSet.\,(union\,x\,y\,z) \wedge x = y$ are disjoint because $a \neq b$ and $a = b$ are contradictory. This is not true because Γ includes the axioms ACU for the multisets; let us denote these axioms by ϕ. Then the two patterns are equivalent to $\exists x,y,z{:}MSet.\,(union\,x\,y\,z) \wedge x \neq y \wedge \phi$ and $\exists x,y,z{:}MSet.\,(union\,x\,y\,z) \wedge x = y \wedge \phi$, respectively. Obviously, $x \neq y \wedge \phi$ and $x = y \wedge \phi$ are not contradictory and the two patterns could match common elements.

The difference $[\![u|\varphi]\!] \setminus [\![u|\psi]\!]$ is the same as the interpretation of the pattern

$$\exists \overline{x}{:}\overline{s}.\,(u^{\mathsf{ML}} \wedge \varphi^{\mathsf{ML}}) \wedge \neg(\exists \overline{x}{:}\overline{s}.\,(u^{\mathsf{ML}} \wedge \psi^{\mathsf{ML}}))$$

and $[\![u|\varphi]\!] \setminus\!\setminus [\![u|\psi]\!]$ is the same as the interpretation of

$$\exists \overline{x}{:}\overline{s}.\,(u^{\mathsf{ML}} \wedge \varphi^{\mathsf{ML}}) \wedge \exists \overline{x}{:}\overline{s}.\,(u^{\mathsf{ML}} \wedge \neg\psi^{\mathsf{ML}}),$$

which is equivalent to $\exists \overline{x}{:}\overline{s}.\,(u^{\mathsf{ML}} \wedge \varphi^{\mathsf{ML}} \wedge \neg\psi^{\mathsf{ML}})$. We can prove that $[\![u|\varphi]\!] \setminus\!\setminus [\![u|\psi]\!]$ is indeed an over-approximation of the difference:

Proposition 4.
$C^{\mathsf{ML}}_{\Sigma/E,B} \models \exists \overline{x}{:}\overline{s}.\,(u^{\mathsf{ML}} \wedge \varphi^{\mathsf{ML}}) \wedge \neg(\exists \overline{x}{:}\overline{s}.\,(u^{\mathsf{ML}} \wedge \psi^{\mathsf{ML}})) \subseteq \exists \overline{x}{:}\overline{s}.\,(u^{\mathsf{ML}} \wedge \varphi^{\mathsf{ML}} \wedge \neg\psi^{\mathsf{ML}})$

Parameterized Intersections. The intersection of two constrained constructor patterns that share a set of variables Y is defined as

$$(u|\varphi) \wedge_Y (v|\psi) \triangleq \bigvee_{\alpha \in Unif_{E_\Omega \cup B_\Omega}(u,v)} (u|\varphi \wedge \psi\alpha)$$

where $Unif_{E_\Omega \cup B_\Omega}(u, v)$ is a complete set of $E_\Omega \cup B_\Omega$-unifiers (the parameterized intersection is defined only when such a set exists). We have

$$[\![(u|\varphi) \wedge_Y (v|\psi)]\!] = \bigcup_{\rho \in [Y \to T_\Omega]} [\![u|\varphi]\!] \cap [\![v|\psi]\!]$$

For the case when $E = B = \emptyset$, it is shown in [1] that

$$\Gamma^{\Sigma} \models u \wedge v \leftrightarrow u \wedge \sigma^{\mathsf{ML}}$$

where σ is the most general unifier of u and v. We obtain as a consequence that $(u \wedge \varphi) \wedge (v \wedge \psi)$ is equivalent to $u \wedge \sigma^{\mathsf{ML}} \wedge \varphi \wedge \psi$, which is the ML translation of the corresponding constrained constructor pattern $(u|\varphi) \wedge_Y (v|\psi)$. We claim that this result can be generalized:

Theorem 3. *If $\{\sigma_1, \ldots, \sigma_k\}$ is a complete set of $B_\Omega \cup E_\Omega$-unifiers for u_1 and u_2, then $C^{\mathsf{ML}}_{\Sigma/E,B} \models (u_1 \wedge u_2) \leftrightarrow (u_i \wedge (\sigma_1^{\mathsf{ML}} \vee \cdots \vee \sigma_k^{\mathsf{ML}}))$, for $i = 1, 2$.*

So, the parameterized intersection of two constrained constructor patterns is encoded in ML by the conjunction of the corresponding ML patterns.

Parameterized Containments. Given the constrained constructor patterns $u|\varphi$ and $\{(v_i|\psi_i) \mid i \in I\}$ with the shared variables Y, their set containment is defined as follows:

$$[\![u|\varphi]\!] \subseteq_Y [\![\bigvee_{i \in I}(v_i|\psi_i)]\!] \text{ iff } \forall \rho \in [Y \to T_\Omega].\, [\![(u|\varphi)\rho]\!] \subseteq [\![\bigvee_{i \in I}(v_i|\psi_i)\rho]\!]$$

The Y-parameterized subsumption of $u|\varphi$ by $\{(v_i|\psi_i) \mid i \in I\}$, denoted $u|\varphi \sqsubseteq_Y \bigvee_{i \in I}(v_i|\psi_i)$, holds iff $C_{\Sigma/E,B} \models \varphi \to \bigvee_{(i,\beta) \in \mathrm{MATCH}(u, \{v_i|i \in I\}, Y)} (\psi_i\beta)$. The following result holds: if $u|\varphi \sqsubseteq_Y \bigvee_{i \in I}(v_i|\psi_i)$ then $[\![u|\varphi]\!] \subseteq_Y [\![\bigvee_{i \in I}(v_i|\psi_i)]\!]$.

Let us discuss the ML counterpart of the parameterized subsumption. The ML pattern expressing $[\![u|\varphi]\!] \subseteq \bigcup_{i\in I}[\![(v_i|\psi_i)]\!]$ is

$$\forall \overline{z}{:}\overline{s'}.\ \left(\exists \overline{x}{:}\overline{s}.\, u^{\mathsf{ML}} \wedge \varphi^{\mathsf{ML}} \subseteq \bigvee_{i\in I} \exists \overline{yi}{:}\overline{si}.\, v_i^{\mathsf{ML}} \wedge \psi_i^{\mathsf{ML}}\right)$$

where $\overline{z}{:}\overline{s'}$ is the set of variables freely occurring in both $u|\varphi$ and $\{(v_i|\psi_i) \mid i \in I\}$, $\overline{x}{:}\overline{s}$ is the set of variables different of $\overline{z}{:}\overline{s'}$ that freely occur in $u|\varphi$, and $\overline{yi}{:}\overline{si}$ is the set of variables different of $\overline{z}{:}\overline{s'}$ that freely occur in $v_i|\psi_i$.

The ML translation of $u|\varphi \sqsubseteq \{(v_i|\psi_i) \mid i \in I\}$ is

$$C^{\mathsf{ML}}_{\Sigma/B,E} \models \varphi^{\mathsf{ML}} \rightarrow \bigvee_{(i,\beta)\in \textsc{match}(u,\{v_i|i\in I\},Y)} \left(\psi_i^{\mathsf{ML}} \wedge \beta^{\mathsf{ML}}\right)$$

where $\textsc{match}(u, \{v_i \mid i \in I\}, Y)$ include substitutions β defined over $var(v_i) \setminus Y$, and β^{ML} is the pattern describing the substitution β. We can prove now that the two ML patterns are equivalent.

Theorem 4.

$$C^{\mathsf{ML}}_{\Sigma/E,B} \models \left(\forall \overline{z}{:}\overline{s'}.\ \left(\exists \overline{x}{:}\overline{s}.\, u^{\mathsf{ML}} \wedge \varphi^{\mathsf{ML}} \subseteq \bigvee_{i\in I} \exists \overline{yi}{:}\overline{si}.\, v_i^{\mathsf{ML}} \wedge \psi_i^{\mathsf{ML}}\right)\right) \leftrightarrow \left(\varphi^{\mathsf{ML}} \rightarrow \bigvee_{(i,\beta)\in \textsc{match}(u,\{v_i|i\in I\})} (\psi_i\beta)^{\mathsf{ML}}\right)$$

6 Conclusion

The paper establishes the exact relationship between two approaches that formalize state predicates of distributed systems: constrained constructor patterns [11] and matching logic [2]. The main conclusion from this comparison is that there is a mutual benefit. Matching logic can benefit from borrowing the computationally efficient reasoning modulo $E \cup B$. A first step is given in [1], but we think that there is more potential that can be exploited. On the other hand, the theory of constrained constructor patterns can get more expressiveness from its formalization as a fragment of the matching logic.

References

1. Arusoaie, A., Lucanu, D.: Unification in matching logic. In: Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11800, pp. 502–518 (2019). https://doi.org/10.1007/978-3-030-30942-8_30, https://doi.org/10.1007/978-3-030-30942-8_30
2. Chen, X., Roşu, G.: Applicative matching logic. Tech. Rep. http://hdl.handle.net/2142/104616, University of Illinois at Urbana-Champaign (July 2019)
3. Chen, X., Rosu, G.: Matching μ-logic. In: 34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019. pp. 1–13. IEEE (2019). https://doi.org/10.1109/LICS.2019.8785675, https://doi.org/10.1109/LICS.2019.8785675

4. Escobar, S., Sasse, R., Meseguer, J.: Folding variant narrowing and optimal variant termination. J. Log. Algebr. Program. **81**(7-8), 898–928 (2012). https://doi.org/10.1016/j.jlap.2012.01.002, https://doi.org/10.1016/j.jlap.2012.01.002
5. Futatsugi, K.: Fostering proof scores in cafeobj. In: Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6447, pp. 1–20. Springer (2010). https://doi.org/10.1007/978-3-642-16901-4_1, https://doi.org/10.1007/978-3-642-16901-4_1
6. Goguen, J.A., Thatcher, J.W., , Wagner, E.G.: An initial algebra approach to the specification, correctness, and implementation of abstract data types. Tech. Rep. RC 6487, IBM Res. Rep. (1976), see also Current Trends in Programming Methodology, Vol. 4: Data Structuring, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1978, pp. 80-149
7. Meseguer, J.: Variant-based satisfiability in initial algebras. Sci. Comput. Program. **154**, 3–41 (2018). https://doi.org/10.1016/j.scico.2017.09.001
8. Meseguer, J.: Generalized rewrite theories, coherence completion, and symbolic methods. J. Log. Algebr. Meth. Program. **110** (2020). https://doi.org/10.1016/j.jlamp.2019.100483
9. Meseguer, J.: Twenty years of rewriting logic. The Journal of Logic and Algebraic Programming **81**(7), 721 – 781 (2012). https://doi.org/10.1016/j.jlap.2012.06.003, rewriting Logic and its Applications
10. Roşu, G.: Matching logic. Logical Methods in Computer Science **13**(4), 1–61 (December 2017)
11. Skeirik, S., Stefanescu, A., Meseguer, J.: A constructor-based reachability logic for rewrite theories. In: Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10855, pp. 201–217. Springer (2018). https://doi.org/10.1007/978-3-319-94460-9_12, https://doi.org/10.1007/978-3-319-94460-9_12
12. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics **5**(2), 285–309 (1955)

Analysis of the Runtime Resource Provisioning of BPMN Processes using Maude

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

[1] ITIS Software, University of Málaga, Málaga, Spain
[2] Pontificia Universidad Javeriana, Cali, Colombia
[3] Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, F-38000 Grenoble France

Abstract. Companies are continuously adjusting their resources to their needs following different strategies. However, the dynamic provisioning strategies are hard to compare. This paper proposes an automatic analysis technique to evaluate and compare the execution time and resource occupancy of a business process relative to a workload and a provisioning strategy. Such analysis is performed on models conforming to an extension of BPMN with quantitative information, including resource availability and constraints. Within this framework, the approach is fully mechanized using a formal and executable specification in the rewriting logic framework, which relies on existing techniques and tools for simulating probabilistic and real-time specifications.

1 Introduction

A crucial concern in most organizations is to have explicit and precise models of their business processes. These models may allow organizations to better understand, control, and manage critical activities and, possibly, make improvements to their processes. Indeed, process optimization is at the heart of business process management because of its potential to increase profit margins and reduce operational costs.

A business process is a collection of structured activities or tasks that produce a specific product and fulfil a specific organizational goal for a customer or market. A process aims at modeling activities, and their causal and temporal relationships by defining specific business rules. Process instances then have to comply with such a description once they are deployed. The Business Process Model and Notation (BPMN) [10] is a graphical modeling language for specifying business processes, which has become the common notation for designing business processes. Several industrial platforms have been developed during the last 10 years to support the modeling and management of BPMN processes. Nowadays, organizations are making efforts to use such platforms to define their organizational processes, aiming at achieving better control over the processes when they are deployed.

Once a process description has been obtained, a key question to ask —from the business perspective— is the following: can this process be improved to, e.g., save money? Process optimization is becoming a strategic activity in organizations because of its potential to increase profit margins and reduce operational costs. One of the main problems in process optimization is concerned with the task of streamlining resource provisioning, allocation, and sharing. A resource can be a machine, a robot, a tool, or an employee profile, and it may be associated with a cost. Given the strategic importance in saving costs where possible, a collection of resource patterns have been defined in the context of the workflow patterns initiative [1].

Providing automated techniques for analyzing and optimizing BPMN processes is a challenging problem. It requires a model of the process including execution time of tasks and flows, as well as an explicit description of resource usage requirements. A solution to this problem would take such a process as input and compute a set of metrics (e.g., process execution time, waiting times, resource occupancy) as output. These measures would then be used as part of a further analysis stage with the goal of optimizing the process relative to a cost model. All this effort is to have a proposal for, e.g., better allocation of resources and, thus, reducing the overall time and/or costs of the process when it is finally deployed. However, the assignment of resources is seldom static, rendering the optimization problem more interesting. Modern enterprises and systems have access to resource repositories and to the possibility of acquiring/releasing or hiring/firing them with great flexibility. Thus, they can provision and release resource instances as needed. Since the analysis procedure involves complex computations and lengthy simulations, it is highly convenient to be able to perform resource analysis in a fully automated way, especially at design time before the processes are deployed.

This paper presents a solution for the analysis of alternative strategies for the dynamic adaptation of resource assignments in process models. Instead of focusing on the allocation of a fixed set of available resources, alternative strategies are analyzed for the dynamic provisioning of such resources. Once the best strategy is chosen using one of the proposed methods, such adaptation strategy will allow to automatically adjust the number of required instances of resources at runtime, depending on the workload and the behavior of the process. The annotations on the BPMN processes will provide the necessary information on task durations, probabilistic choice, and information regarding resources (e.g., initial number of available resources, resources required per task, maximum number of resources). The approach relies on a formal specification in rewriting logic of BPMN processes. The specification is given in the rewriting-logic based language Maude and serves as an executable semantics of the BPMN language under consideration. Since it is *executable*, it has the advantage of enabling the use of Maude's verification tools for computing a number of metrics of processes with a precise mathematical meaning.

The approach presented here is concerned with the analysis of quantitative properties associated to BPMN processes. Although it encompasses a broad selection of quantitative measures, the main focus in this paper is given to execution time (i.e., the time it takes to execute a process) and resource occupancy (i.e., the percentage of usage of any or all replicas of a resource). The final goal is thus to use such analyses to streamline a process by reducing its operational costs in relation to execution time and resources, which can be directly inferred from the estimated execution times and resource usage. Since these measures are computed by significant simulations, and also along the actual executions, they can be used to dynamically adjust the number of resources at runtime. The given formalization and the accompanying tools will enable the comparison of different strategies for resource provisioning in a dynamic environment. More precisely, given a process description, and taking as parameters the workload and the provisioning strategy, the techniques and tools presented here provide detailed information on the evolution of execution times, resources in use, and therefore costs, which altogether will help in deciding on the best fit for the specific needs.

The application of the approach is presented and discussed on a case study with dynamic allocation of resources. It is used to show how the proposed approach can be helpful to effectively reduce the cost and execution time of a process. The current Maude specification

builds on the one developed by the same authors in previous related work [6, 7] for different forms of analysis of business processes. The reader is referred to `http://maude.lcc.uma.es/BPMN-RA` for details on the formal specification, experiments, and additional examples.

Explicitly, the list of original contributions of this paper are as follows: (i) an extension of BPMN with annotations to include quantitative information in the process description, (ii) a formal specification of annotated BPMN in rewriting logic to give a formal semantics to this extended language, (iii) a formal description of mechanisms for dynamically adjusting the number of required resources to the workload, (iv) automated analysis techniques for the comparison of alternative dynamic resource provisioning strategies, and (v) the validation of this approach on a realistic case study.

The organization of the rest of the paper is as follows. Section 2 presents the BPMN notation extended with the annotations supporting the proposed approach. Section 3 introduces Real-Time Maude. Section 4 overviews the specification of the annotated BPMN extension in Maude's rewriting logic, which serves as a semantics for the language and makes automated analysis possible using Maude's tools. Section 5 presents the novel analysis techniques and case studies illustrating how the number of resources evolve, and how costs and time can be reduced in practice without the need for human intervention. Section 6 presents a discussion on related work. Finally, Section 7 concludes the paper.

2 Annotated BPMN

Familiarity with the BPMN notation is assumed. In this section the focus is on its extension to support quantitative information. In essence, times are expressed as stochastic expressions and branching alternatives as probabilities associated to branches. These parameters are supposed to be provided by the experts that specify the business process, or are learnt from available execution logs using —for instance— recent contributions on process mining and discovery such as [19, 12]. Figure 1 summarizes the BPMN constructs supported in this work. These elements are used to develop activity and collaboration diagrams of process models. In addition to the description of specific tasks and their sequencing, collaboration diagrams also involve *pools* and *lanes*, which are structuring elements that split processes into pieces.

To introduce and illustrate the use of the BPMN constructs and annotations supported, and the analysis techniques presented in this paper, a process describing a parcel ordering and delivery by drones will be used. Figure 2 presents a collaboration diagram modeling such a process. It consists of three lanes, namely, one for the client, one for the order management, and one for the delivery process. In this process, the client first signs in and then repeatedly looks for products. Eventually, the client can decide to give up (i.e., termination) or to make an order by submitting it to the order management lane. The client then waits for a response (i.e., acceptance or refusal of this order). If the order can be completed, then the parcel is received and the client pays for it. Otherwise (i.e., timeout or order refused), the client fills in a feedback form. As far as the management lane is concerned, the first task aims at verifying whether the goods ordered by the client are available. If they are not available, then the order is canceled; otherwise, the order is confirmed. The order management takes care of the payment of the order whereas the delivery lane is triggered to prepare the parcel to be delivered by a drone. This process exhibits different kinds of gateways, probabilities for choice gateways, stochastic functions for time associated to tasks, and a loop (Search products task).

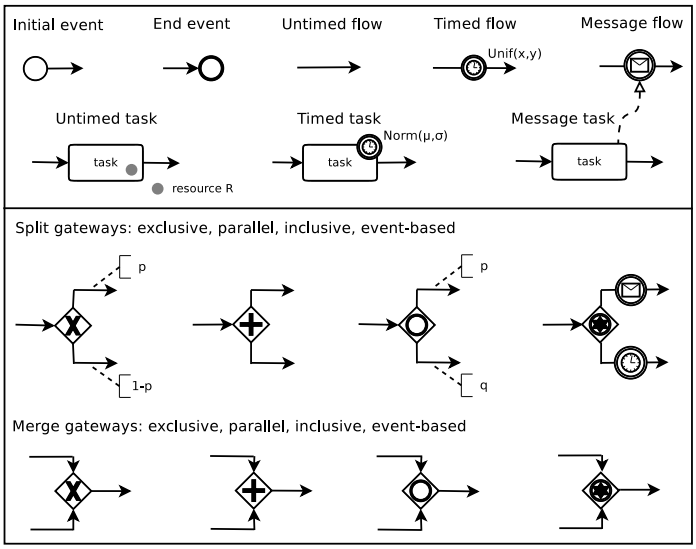


Fig. 1. Supported BPMN syntax.

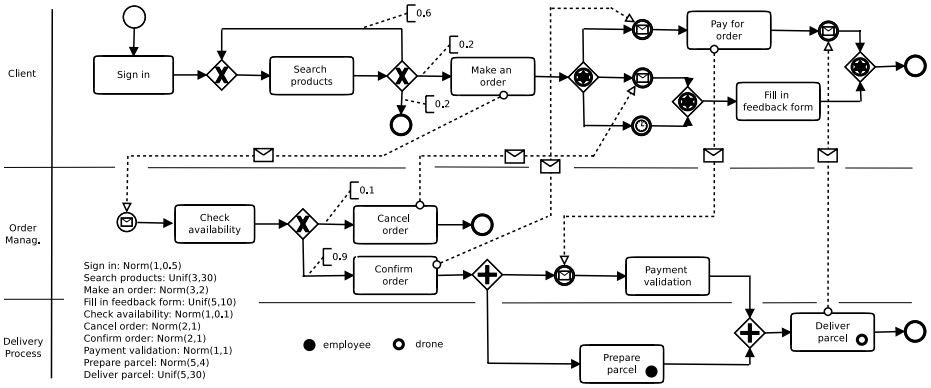


Fig. 2. Running example: parcel delivery by drones.

The timing information associated to tasks and flows (durations or delays) is described either as a literal value (a non-negative real number) or sampled from a probability distribution function according to some meaningful parameters. The probability distribution functions currently available include exponential, normal/Gauss, and uniform (see, e.g., [20]). To simplify the reading of the process in Figure 2, the delays in all flows are set to 0 and the specification of the task duration has been placed apart from the process description, at the bottom-left corner. In the modelling tool, these parameters would be specified as properties of the corresponding elements. For instance, the duration of the Sign in task follows a normal distribution with mean 1 and variance 0.5, and the Search products task follows a uniform distribution in the interval [3,30].

Four types of gateways are considered: *exclusive*, *inclusive*, *parallel*, and *event-based*. Both BPMN 1.0 and 2.0 semantics for inclusive gateways are supported in this work. Data-based conditions for split gateways are modeled using probabilities associated to outgoing flows of exclusive and inclusive split gateways. For instance, notice the exclusive split after the Search products task in the Client lane of the running example, which has outgoing branches with probabilities 0.6, 0.2, and 0.2, specifying the likelihood of following each corresponding path. The probabilities of the outgoing flows in an exclusive split must sum up to 1, while each outgoing flow in an inclusive split can be equipped with a probability between 0 and 1 without a restriction on their total sum.

Each lane in a collaboration diagram corresponds to a specific role or resource. However, instead of implicitly associating resources to lanes, resources are explicitly defined at the task level. Thus, a task that requires resources can include, as part of its specification, the number of required instances (or replicas) of a resource. The process in Figure 2 relies on employees for parcel packing and drones for parcel delivery. The small circles at the bottom-right corner of the Prepare parcel and Deliver parcel tasks indicate that one instance of the employee resource and another one of the drone resource are required, resp., for the tasks completion. Several tasks could compete for the same resources. Furthermore, since *multiple* instances of a same process may be executed concurrently, instances also access and may compete for the shared resources. Notice that resources can refer to humans (e.g., employee, cashier, executive) as well as non-human ones (e.g., drone, virtual machine, paper, money), and we can specify the number of instances or replicas as a natural number. In case of unlimited resources, the number of units of time, length, or volume can also be considered.

As presented in Section 4, provisioning strategies are specified in Maude. Although in future work alternative mechanisms to specify them can be developed, in the present work the interest is in their comparative analysis, and therefore a catalog of strategies is presented from which the desired one can be chosen. Independently of the criteria used for the provisioning/releasing of resources, it is assumed that the amount of resources is accounted for periodically (time between checks or TBC), and that in that check the recent history of the process is considered, being the length of the considered history another parameter (history length or HL) of the optimization process. The provisioning and releasing of resource instances is supposed to happen in accordance to some given thresholds, which will also be provided as parameters. Finally, it is assumed that there is a maximum and minimum number of instances available for each resource, which will be given by a range (*min*,*max*). Unlimited availability of a resource can then be modeled by just assigning a negative *max* value.

3 Real-Time and Probabilistic Rewrite Theories

This section provides an overview of real-time and probabilistic features of rewriting logic [13] and Maude [5]. The executable specification of BPMN presented in the following sections is a probabilistic rewrite theory [3] $\mathcal{R}=(\Sigma,E\uplus B,R)$, where $(\Sigma,E\uplus B)$ is a membership equational logic [4] theory with Σ its signature, E a set of conditional equations, B a set of equational axioms so that equational rewriting is performed modulo B, and R is a set of labeled conditional rules. The equational subtheory offers the infrastructure for defining a process in the sublanguage of BPMN described in Section 2, including the timing behavior for tasks and flows, resource dynamics, and probabilities for outgoing flows of split gateways.

The real-time aspects are modeled using Real-Time Maude [16], which supports the formal specification and analysis of *real-time systems*. Specifically, the probabilistic rewrite rules R axiomatize how time advances and probabilistic choices are made in this infrastructure, in order for a given process to transition from an initial to a final state.

Real-Time Maude provides a sort Time to model the time domain, which can be either discrete or dense. Time advancement is modeled with *tick rules*, e.g.,

$$\texttt{crl [}l\texttt{] : \{ }t\texttt{, }T\texttt{ \} => \{ }t'\texttt{, }T+\tau\texttt{ \} if }C\texttt{ .}$$

where t and t' are system states (an evolving model in our case), T is the global time and τ is a term of sort Time that denotes the *duration* of the rewrite, affecting the *global time elapse*. Since tick rules affect the global time, in Real-Time Maude time elapse is usually modeled by one single tick rule and the system dynamic behavior by instantaneous transitions. Although there can be many sampling strategies, in this work time elapse is modeled with a single tick rule with the help of two functions: the delta function, that defines the effect of time elapse over every model element, and the mte (maximal time elapse) function, that defines the maximum amount of time that can elapse before any action is performed (see [16] for additional details).

In a standard rewrite theory, the conditions of rewrite rules are assumed to be purely equational. A rewrite rule $l(\overrightarrow{x}) \rightarrow r(\overrightarrow{x})$ if $\phi(\overrightarrow{x})$ specifies a pattern $l(\overrightarrow{x})$ that can match some fragment of the system's state t if there is a substitution θ for the variables $\overrightarrow{x}$ that makes $\theta(l(\overrightarrow{x}))$ equal modulo B to that state fragment, changing it to the term $\theta(r(\overrightarrow{x}))$ in a local transition if the condition $\theta(\phi(\overrightarrow{x}))$ is true. In a probabilistic rewrite theory, rewrite rules can have the more general form $l(\overrightarrow{x}) \rightarrow r(\overrightarrow{x},\overrightarrow{y})$ if $\phi(\overrightarrow{x})$ with probability $\overrightarrow{y} := \phi(\overrightarrow{x})$, where some new variables $\overrightarrow{y}$ are present in the pattern r on the right-hand side. Because the pattern $r(\overrightarrow{x},\overrightarrow{y})$ may have new variables $\overrightarrow{y}$, the next state specified by such a rule is not uniquely determined: it depends on the choice of an additional substitution ρ for the variables $\overrightarrow{y}$. In this case, the choice of ρ is made according to the family of probability functions π_θ: one for each matching substitution θ of the variables $\overrightarrow{x}$. Therefore, a probabilistic rewrite theory can express both non-deterministic and probabilistic behavior of a concurrent system.

4 Executable Specification of BPMN

This section presents the Maude representation of the timed and probabilistic extensions of BPMN introduced in Section 2. The algebraic semantics of BPMN is provided by a MEL theory $Spec_{\text{BPMN}}$ so that a process model P is an element of the initial algebra $\mathcal{T}_{Spec_{\text{BPMN}}}$. The rewrite theory RT_{BPMN} extends $Spec_{\text{BPMN}}$ and defines the behavior of BPMN processes by providing some additional definitions and rules specifying such a behavior. The Maude specification of BPMN therefore consists of two parts: the process structure as an equational specification and its evolution semantics using rewrite rules.

Process description. In the Maude specification of BPMN, a process is represented as an object with sets of flows and nodes as attributes. The representation of each node type includes the necessary information to describe its structure and to contribute to the overall process analysis. For instance, a task node involves an identifier, a description, two flow identifiers (input and output), a stochastic function modeling its duration (0 if there is no

```
< pid : Process |
    nodes : (start(initial, cf1),
             merge(g1, exclusive, (cf2, cf5), cf3),
             split(g2, exclusive, cf4, ((cf5, 0.6) (cf6, 0.2) (cf7, 0.2))),
             split(g3, eventbased, cf8, (cf9, cf10, cf11)),
             task(t10, "Prepare parcel", mf7, df1, Norm(5.0, 4.0), employee, empty),
             task(t11, "Deliver parcel", df1, df2, Unif(5.0, 30.0), drone, parceldelivered),
             ...),
    flows : (flow(cf1,  0),
             flow(cf9,  0, message(orderconfirmed, "Order confirmed")),
             flow(cf10, 0, message(ordercanceled, "Order canceled")),
             flow(cf11, 0, timer(timeout, 60)),
             ...) >
```

Fig. 3. Running example: representation in Maude of the parcel delivery process.

duration), a set of resources required for its execution, and a set of messages to be delivered after its completion. A split node includes a node identifier, a gateway type (exclusive, parallel, inclusive, or event-based), an input flow identifier, and a set of output flow identifiers. A merge node includes a node identifier, a gateway type, a set of input flow identifiers, and an output flow identifier. The representation of any flow includes a probability distribution function specifying its delay, a message produced by a task that blocks the flow until the message is received, and a timer representing a delay after which the execution can be triggered.

Figure 3 gives an excerpt of the representation for the running example. It shows how a Process object has attributes with the definition of its nodes and flows connecting them. For example, the exclusive split g2 has as incoming flow cf4 and outgoing flows cf5, cf6, and cf7, with associated probabilities 0.6, 0.2, and 0.2, respectively. As another example, the event-based split gate g3 has as incoming flow cf8 and outgoing flows cf9, cf10, and cf11. These flows are defined in the set of flows.

The transformation from the BPMN diagrammatic representation of processes into the corresponding Maude representation is carried out using the VBPMN platform [11].

Execution semantics. The operational semantics of BPMN is defined using rewrite rules, modeling how *tokens* (see below) evolve through a process, thus defining the execution semantics of BPMN. Each observable action is modeled as a rewrite rule. E.g., when a token arrives at an event-based split gateway, the token is made active with its optional timer. In that rule, if there is an outgoing flow with a timer, an event is added with the corresponding time to the set of available events. Another rule specifies the case where there is an outgoing flow with a message in the set of events. For instance, in that case, that branch is activated and one token is added for that flow. Additional objects of classes Workload and Supervisor are in charge of, respectively, modelling the workload of the process, and provisioning resources depending on the whereabouts of the process execution. In general, rewrite rules operate on systems composed of a Process object, a Simulation object, a Workload object, and a Supervisor object.

Simulation. While the process object represents the BPMN process and does not change during executions, a simulation object keeps information on an execution of the process. It stores a collection of tokens (in a scheduler, see below), a global time (gtime), a set of events (messages and timers), and a set of resources. It also keeps track of the metrics being

```
01 class Simulation |
02   tokens : List{Token},              ---- scheduler
03   gtime : Time,                      ---- global time
04   resources : Set{Resource},         ---- resources in the system
05   events : Map{Id,Set{Event}},       ---- events in each execution
06   process-execs : Map{Id,Time},      ---- execution time of each execution
07   sync-times : Map{Id,Map{Id,Time}}, ---- synchronization time of each gate in each execution
08   task-times : Map{Id,Map{Id,Time}}, ---- task execution times
09   ...
```

Fig. 4. Declaration of the Simulation class (partial, please, note the ellipsis).

computed. Figure 4 presents the parameters of the Simulation class. We can get an intuition of how these values get updated in the rule in Figure 5.

Tokens. Tokens are used to represent the evolution of the workflow under execution. A token is represented as a term token(TId, Id, T). Since several executions are simultaneously happening, each execution has a unique identifier. Tokens are identified by the execution instance TId they belong to, and the flow or node Id they are attached to. The expression T represents a timer, of sort Time, modeling a delay on the token. Once this timer becomes 0, the token can be consumed.

Scheduling. Tokens are stored in a *scheduler* — see the attribute tokens of the Simulation object in Figure 4 — implemented as a priority queue, so that tokens are stored according to their due time. However, even with its timer set to 0, the token at the front may be not enough to fire some action. Consider for example a task that requires some resource that is not available or a parallel merge for which some incoming flow is not yet active. To avoid blocking situations, the scheduler is provided with a *shifting* mechanism, which moves the first active token to the front of the scheduler in case the current head cannot fire the corresponding action. This scheduler is similar to those used in typical discrete event simulations.

Events.[4] A message event may be associated to a flow, which is blocked until the message is received. A timer event may be associated to a flow. When a token arrives at a timer event, its countdown is started: once the countdown is completed, the token moves to the outgoing flow. Both message and timer events are usually associated to event-based gateways, but it is not necessarily the case (see, e.g., the initial flow for the order management lane in the process in Figure 2). Asynchronous events are modeled using an event set in the Simulation object. When a message is dispatched, a corresponding event is added to the set. Flows and gateways that are waiting for specific messages use this set to check whether the messages have arrived.

Dynamic resources. Each resource is described with an identifier, the number of available replicas (initially the total number), the total amount of time this resource has been in use, and the intervals of time on which it was used. These two last parameters are required for analysis purposes only. When a task requires several resources, it atomically uses them or waits for them to be available.

Tasks. A task execution is modeled with two rules. The first rule, the initTask rule shown in Figure 5, represents the task initiation, which is applied when a token with zero time is available at the incoming flow (Line 05). If all the resources required by this task are

[4] Only inter-lane events are considered; to consider environment events, the environment may be added to the simulation model.

```
01 rl [initTask] :
02    < PId : Process |
03       nodes : (task(NId, TaskName, FId1, FId2, SE, RIds, SEI), Nodes), Atts >
04    < SId : Simulation |
05       tokens : (token(TId, FId1, 0) Tks),
06       task-tstamps : TTSs, gtime : T, resources : Rs, Atts1 >
07    < CId : Counter | counter : N >
08 => if allResourcesAvailable(RIds, Rs)
09    then < PId : Process |
10            nodes : (task(NId, TaskName, FId1, FId2, SE, RIds, SEI), Nodes), Atts >
11         < SId : Simulation |
12            tokens : insert(Tks, token(TId, NId, time(eval(SE, N)))),
13            task-tstamps : if TTSs[TId][NId] == undefined
14                           then insert(TId, insert(NId, T, TTSs[TId]), TTSs)
15                           else TTSs fi,  ---- for loops, stamps get overwritten
16            gtime : T,
17            resources : grabResources&updateTime(RIds, Rs, time(eval(SE, N)), T), Atts1 >
18         < CId : Counter | counter : int(eval(SE, N)) >
19    else ... fi .                          ---- if necessary, the scheduler is updated
```

Fig. 5. Task initiation rule.

```
01 rl [supervisor] :
02    < SId : Simulation | resources : Rs, gtime : T, Atts1 >
03    < Sup : SupervisorUsage | TBC : TBC, time-to-next-check : 0,
04                              CI : CI, thresholds : Thds, Atts2 >
05 => < SId : Simulation | gtime : T,
06       resources : update(Rs, Thds, CI, T), Atts1 >
07    < Sup : SupervisorUsage | TBC : TBC, time-to-next-check : TBC,
08                              CI : CI, thresholds : Thds, Atts2 >
```

Fig. 6. Usage-based strategy supervisor rule.

available, which is checked with the allResourcesAvailable function (Line 08), then a new token is generated with the task identifier and the task duration (Line 12). Otherwise, the scheduler's token shifting mechanism is invoked (Line 19 —note the ellipsis). If available, all required resources are removed from the resource set and the time those resources have been in use is updated (grabResources&updateTime function, Line 17). Note also that rules update the information on execution times, task durations, etc. (see, e.g., the update of the task-tstamps attribute, Lines 13–15).

Merge gateways. When a merge gateway is triggered, the incoming tokens are removed, a new token is added to the scheduler for the outgoing flow, and simulation information is updated with synchronization times. For inclusive gateways, the semantics of BPMN 1.0 and 2.0 are both supported in this research.

Supervisor. A Simulation object collects all data relevant for the analysis, which is then used by a supervisor object to decide on the number of resource instances. Intuitively, the supervisor object is in charge of collecting the data on the chosen metric for the specified window of time (history length) and then decides in accordance. It takes into account ranges and thresholds for each resource to change, every TBC time units, the total amount of resources available to the process. Figure 6 represents the resource check action. Every TBC time units, the supervisor object updates the number of resource instances (Line 06) according to the state of the resources (Rs), the thresholds (Thds), the interval to consider (CI), and the current global time (T).

```
01 rl [Workload] :
02    < WId : Workload | timer : 0, rate : SE, works : s W >
03    < SId : Simulation | tokens : Tks, events : ME, Atts1 >
04    < CId : Counter | counter : N >
05 => < WId : Workload | timer : time(eval(SE, N)), rate : SE, works : W >
06    < SId : Simulation |
07        tokens : insert(Tks, token(token(s W), initial, 0)),
08        events : (token(s W) |-> empty, ME),
09        Atts1 >
10    < CId : Counter | counter : int(eval(SE, N)) >
```

Fig. 7. Workload rule.

Workloads. Simulation-based analysis techniques are typically parameterized by the workload. They define the rate at which new instances of a given process are executed. The rule in Figure 7 specifies the behavior of closed workloads. Given a number of works, or times the process is to be executed (attribute works), and a stochastic expression SE describing the inter-arrival time (kept in the rate attribute), the rule generates a new work after the specified amount of time until all works have been created. Notice that the timer attribute of the Workload object is initialized with the result of evaluating the stochastic expression (Line 05). The rule is applicable when the timer becomes 0 and then a new token in the initial node is inserted in the scheduler (Line 07). The evaluation of stochastic expressions is carried out by the eval operation. Random numbers are generated using a pseudo-random number algorithm, which takes a number that indicate the position in the sequence (the Counter object is in charge of appropriately increasing these numbers).

5 Dynamic Resource Allocation

This section presents automated techniques for analyzing dynamic adjustment of resource allocation. The provisioning of resources may be carried out using different criteria. These adaptation strategies present trade-offs between the difficulty of use —mainly due to the amount of parameters that need to be specified or the difficulty to estimate them— and the benefits of an adaptive provisioning in terms of resource costs and response time. In this section, two alternative strategies are presented: one based on the observed resource usage (*usage-based* strategy) and another one based on the demand on the resources (*queue-based* strategy). Although only these two strategies are used here, other metrics could have been used instead. That is, the resource adaptation strategy presented is a parameter of the generic analysis techniques proposed here. It is fair to say that the selected strategies cover two alternative and complementary approaches: while the usage-based one is based on the observation of the behavior of the system, the queue-based one relies on the prediction of the resource demand. As behavioral observations, other typical metrics could have been considered, including the observed response time or its average or variance. As predictive indicators, synchronization times, bottlenecks, or other observations on the internals of processes could also be used.

Whatever the metric used to adapt the processes is, it is assumed that each of them is driven by a recommended range of values: If the observed value goes over some maximum threshold, then the number of instances of a resource is increased; if it goes below some minimum value, then the number is decreased. In trying to avoid under- or over-provisioning, it is assumed that the minimum and maximum number of instances are also bounded. E.g.,

	TBC	HL	range		initial number of instances	thresholds	
			min	max		min	max
employee	5	10	1	3	1	50	70
drone			1	6	1	50	75

Table 1. Sample set of parameters for the delivery process.

due to office space limitations, a process cannot have more than ten employees, independently of its cost or productivity. It is also assumed that the strategy proceeds by checking on some given metrics periodically, and considering the latest values of such metrics in order to make a decision on the provisioning or releasing of resources.

Table 1 summarizes the parameters for the analysis of the process and a possible instantiation of them. TBC and HL stand for the Time Between Checks and the History Length or window of values considered in the check, respectively. Every TBC time units, the state of the system is evaluated and the amount of resource instances correspondingly updated. The evaluation takes into account the given metric for HL time units. Although these strategies consider the average value for the samples in the window, other strategies could also check that all values in the window were over/under a given threshold or any other check considered useful.

Notice that each resource has its own range and threshold. Table 1 specifies a possible selection of values with which the simulations of the delivery running example may be executed, as well as the threshold values for the usage-driven strategy. Specifically, the average usage of each resource replica is expected to be in the range $[50\%, 70\%]$ for employees and [50%,75%] for drones.

Given a process description, a specification of resources (i.e., specific values for the above parameters), and a workload, the experiments discussed in what follows illustrate how information on execution times and resource usage is collected. This information can then be used to find the best strategy or best fit of its parameters. All simulations were performed assuming a closed workload with 1000 instances and an exponentially distributed inter-arrival time ($\lambda = 0.5$).

Figure 8 shows the evolution of the total amount of resources provisioned along the execution of the delivery process using the usage-guided strategy. The evolution of the number of instances is shown on the left for employees and on the right for drones. In this case, the parameters are: unlimited availability, thresholds (50,70) for employee and (50,75) for drone, TBC 1, and HL 0. Notice, first, that the variability is very high; since the TBC is set to 1, the amount of resources is almost continuously re-evaluated. A HL value of 0 increases this continuous adaptation, since decisions are taken by considering the values at the given time, even if that value is not maintained for some time. Also, note that employees, an expensive resource, move between 1 and 14, although most of the time it ranges between 2 and 6-7. For drones, although the most frequent values are in the range 10-20, it moves between 1 and 38. However, there are other values to take into consideration before changing the process parameters. The average execution time for the process is 55.01, with variance 0.61. The usage percentage was rather low, 42.86% for employees and 34.34% for drones. This results in a total cost, assuming the cost per hour for employee is 50€ and 20€ per drone, of 991,613.3€. A comparative study of these values will be presented later to better understand what these numbers mean for the example (see Table 2). But even with these raw data, a poor use of our resources can be observed, which means a higher cost than possibly required.

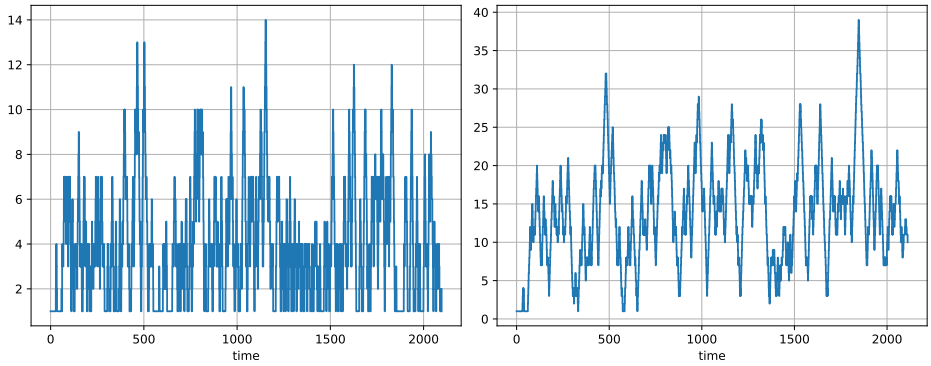


Fig. 8. Number of employee (left) and drone (right) instances along execution with a resource-usage-based strategy (unlimited availability, thresholds: employee (50,70), drone (50,75)). TBC: 1, HL: 0.

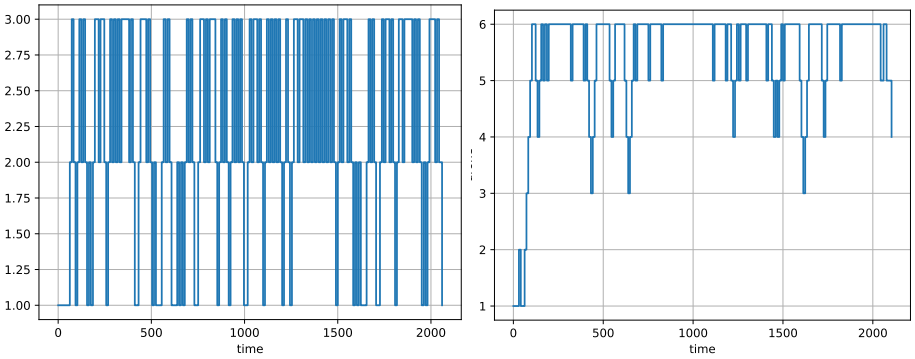


Fig. 9. Number of employee (left) and drone (right) instances along execution with a resource-usage-based strategy (ranges: employee [1,3], drone [1,6], thresholds: employee (50,70), drone (50,75)). TBC: 10, HL: 0.

Figure 9 shows the evolution of the amounts of resources for a new set of parameters. Specifically, two parameters have been changed: the number of instances is now restricted, so that employee instances are now in the range [1,3] and drone instances in the range [1,6]; the number of instances is now re-evaluated every 10 time units. With these parameters, the execution time has slightly improved (average 57.22 and variance 0.72), and also the usage percentage (52.12% for employees and 67.05% for drones). A bigger TBC is allowing the system to stabilize before attempting a new adaptation. This leads to a significant reduction in the total cost to 454,713.8€ (assuming the same costs per hour for employee and drone as above).

As it is shown in the comparative study below, these results are quite good, although they are obtained at the expense of a great variability, as Figures 8 and 9 show. This variability could have been reduced by deciding on the provisioning or release of the resource instances with a larger TBC or a larger window of values, instead of just the latest one. Furthermore, there is also the more realistic alternative of deciding on the current demand of resources and not on the history of results, whatever the size of the window one may want to consider.

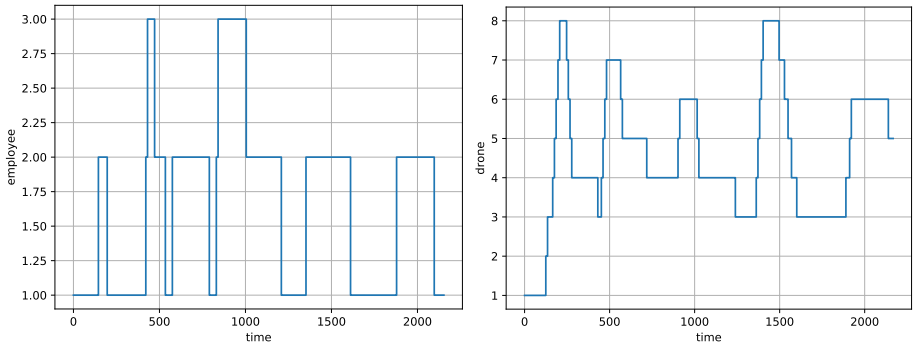


Fig. 10. Number of employee (left) and drone (right) instances along execution with a resource-queue-based strategy (unlimited availability, thresholds: employee (3,6), drone (2,8)). TBC: 10, HL: 5.

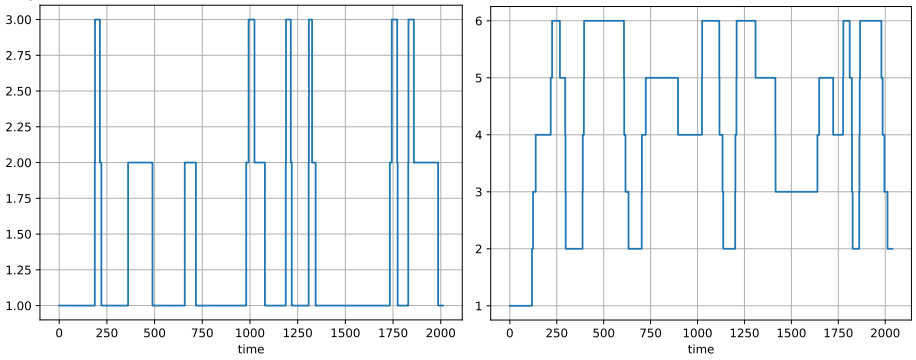


Fig. 11. Number of employee (left) and drone (right) instances along execution with a resource-queue-based strategy (ranges: employee [1,3], drone [1,6], thresholds: employee (3,6), drone (2,8)). TBC: 1, HL: 5.

Figure 10 depicts the evolution of the total number of instances for employees and drones using the queue-based strategy with the following parameters: unlimited availability, thresholds [3,6] for employee and [2,8] for drone, TBC 10, and HL 5. It can be observed in the figure that the total number of instances remains much more stable: the number of employees stays between 1 and 3, and the number of drones varies between 1 and 8, going up and down repeatedly depending on the demand. The information observed in these graphs is complemented with the execution times (average 69.09, variance 1.49), and usage percentages (71.37% for employees, 86.38% for drones). Assuming the same costs as above, this results in a total cost of 370,742.7€.

Although many other values could have been considered for these parameters, the set of graphs we present here is completed by showing what happens if, again with the queue-based strategy, the number of instances of employees is restricted to [1,3] for employees and to [1,6] for drones, and re-evaluation takes place more often (TBC: 1). The evolution of the resources is depicted in Figure 11. The execution times improved (average 70.04, variance 1.57) as well as the resource usage (82.81% for employees, 93.83% for drones). The total cost with these parameters is 305,961.9€.

	strat.	TBC	HL	range	threshold	exec. time		usage (%)		total cost
				empl.-drone	empl.-drone	avg	var	empl.	drone	(in €)
1	usage	1	0	unrestricted	(50,70)-(50,75)	55.01	0.61	42.86	34.34	991,613.3
2		10	0	unrestricted	(50,70)-(50,75)	57.06	0.71	36.10	29.87	972,868.4
3		1	0	[1,3]-[1,6]	(50,70)-(50,75)	57.30	0.73	62.79	70.69	453,761.4
4		10	0	[1,3]-[1,6]	(50,70)-(50,75)	57.22	0.72	52.12	67.05	454,713.8
5	queue	1	5	unrestricted	[3,6]-[2,8]	70.10	1.58	85.93	92.94	313,555.1
6		10	5	unrestricted	[3,6]-[2,8]	69.09	1.49	71.37	86.38	370,742.7
7		1	5	[1,3]-[1,6]	[3,6]-[2,8]	70.04	1.57	82.81	93.83	305,961.9
8		10	5	[1,3]-[1,6]	[3,6]-[2,8]	68.63	1.46	76.02	90.08	303,030.7

Table 2. Exec. times, resource usage, and total costs for different parameters

Table 2 shows the execution times, usage percentages and total costs for several simulations using different parameters. Specifically, a combination of TBCs of 1 and 10, HLs of 0 and 5, restricted and unrestricted resource amounts, and different threshold values are considered. As the previous discussion shows, the selection of the right parameters is indeed a multi-objective problem, where the goal is to minimize execution times and total costs. However, it is not only that, restrictions such as the tolerable variability and the maximum amount of resource instances available need to also be taken into account. It can be observed that the minimum cost in the table is obtained for Row 8, which is the result of restricted availability and stability. Unrestricted resource availability results in higher costs. Furthermore, having unrestricted amounts of resources may be unrealistic in practice. Notice, however, that the difference is not that significant with the queue-based strategy, where resource queues are already representing the accumulated demand. This is indeed what makes this strategy better in general terms than the usage-based strategy.

6 Related Work

Oliveira *et al.* [15] use generalized stochastic Petri nets for correctness verification and performance evaluation of business processes. In their work, an activity can be associated to multiple roles and the completion of an activity can use a portion of the resources available for a role. They also propose metrics for evaluating process performance such as: the minimum number of resources needed for a role in order to complete a process, the expected number of activity instances when completing a process under the assumption of sufficient resources, and the expected activity response time. Colored Petri Nets are used in [14] for understanding how bounded resources can impact the behavior of a process. They introduce the notion of "flexible resource allocation" as a way to assign resources associated to a given role based on priorities. In their approach, they use alternative strategies to better allocate a fixed number of available resources. Havur *et al.* [9] study the problem of resource allocation in business processes management systems where constraints can be assigned to resources (e.g., time of availability) and have dependencies. Their technique is based on the answer set programming formalism and is capable of deriving optimal schedules. Sperl *et al.* [18] describe a stochastic method for quantifying resource utilization relative to structural properties of processes and historical executions. In [7], Maude is used to model and analyze the resource allocation of business processes. In this work, optimal allocation is presented as a multi-objective optimization

problem, where response time and resource usage are minimized. None of the aforementioned works attempts at providing analysis techniques or tools for the dynamic allocation of resources with respect to response time and resource usage, as the proposed approach does.

There are many tools supporting the design and management of business processes (e.g., Arena, ARIS10, iGraphx, Signavio, BPMOne, BIMP, Camunda), of which a subset supports the analysis and optimization of processes. This is the case of, for instance, Signavio [2], which packs tools such as the Signavio Process Intelligence for process optimization. This tool automatically mines process models from currently running systems and monitors those processes with the purpose of collecting data that enables end-users to make decisions for process improvement. Our proposal takes a different approach, since the focus here is on predicting the behavior of designed models given resource provisioning strategies: thus, the approach presented in this work supports the decision making at design time, even before a process is deployed. Then, given a resource allocation strategy, resources are dynamically provisioned and released, respecting the constraints specified as parameters to the process model.

7 Concluding Remarks

This paper focuses on the problem of dynamic resource allocation using BPMN as modeling language for business processes. It presents a version of BPMN extended with annotations for describing the duration of task execution, probabilities in split gateways, and additional information about resources. Given such a process specification, automated techniques are proposed for analyzing its behavior and dynamically adjusting the number of necessary resources following some given adaptation strategy. In this paper, two strategies for resource provisioning (usage-based and queue-based) were presented and illustrated on a concrete example showing how parameters (namely, time between checks, history length, resource ranges, and adaptation thresholds) could be adjusted. The automatic approach was able to handle analyses about the response time and total cost associated to the process. These results were possible thanks to an encoding of the annotated BPMN language into rewriting logic and by using Maude's tools for automating all checks on the concurrent executions of the processes.

Providing mechanisms to automatically finding the best values for given strategies is the first future work direction. This is a multi-objective problem, which is restricted by the concrete nature of the process at hand. The plan is also to investigate on alternative provisioning strategies, with the goal of providing more precise decision criteria. Considering that the provisioning of resources depends on the predictive analysis of the future executions, it is something to also be considered (see, e.g., [8, 17]). Another aim would be at designing and implementing more precise modeling support for the provisioning/releasing procedure, by taking into account aspects such as the time to provision, the releasing cost, etc. Finally, the plan is also to consider a broader form of resources, and to cover resource patterns not currently covered such as the chain and pile-based execution patterns (see [1]).

Acknowledgements. The first author was partially supported by projects PGC2018-094905-B-I00 (Spanish MINECO/FEDER) and UMA18-FEDERJA-180 (J. Andalucía/FEDER). The second author was partially supported by CAPES, Colciencias, and INRIA via the STIC AmSud project "EPIC: EPistemic Interactive Concurrency" (Proc. No 88881.117603/2016-01) and by via the Colciencias ECOS-NORD project "FACTS: Foundational Approach to Computation in Today's Society" (Proc. code 63561).

References

1. Workflow resource patterns. BETA Working Paper Series, WP 127, 2004.
2. Signavio. Available: https://www.signavio.com, 2019.
3. G. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based Specification Language for Probabilistic Object Systems. *ENTCS*, 153(2):213 – 239, 2006.
4. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and Proof in Membership Equational Logic. *Theoretical Comput. Sci.*, 236(1):35–132, 2000.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.
6. F. Durán, C. Rocha, and G. Salaün. Stochastic Analysis of BPMN with Time in Rewriting Logic. *Sci. Comput. Program.*, 168:1–17, 2018.
7. F. Durán, C. Rocha, and G. Salaün. A Rewriting Logic Approach to Resource Allocation Analysis in Business Process Models. *Sci. Comput. Program.*, 183, 2019.
8. C. D. Francescomarino, C. Ghidini, F. M. Maggi, G. Petrucci, and A. Yeshchenko. An Eye into the Future: Leveraging A-priori Knowledge in Predictive Business Process Monitoring. In *Proc. of BPM'17*, volume 10445 of *LNCS*, pages 252–268. Springer, 2017.
9. G. Havur, C. Cabanillas, J. Mendling, and A. Polleres. Resource Allocation with Dependencies in Business Process Management Systems. In *Business Process Management Forum*, pages 3–19. Springer, 2016.
10. ISO/IEC. International Standard 19510, Information technology – Business Process Model and Notation. 2013.
11. A. Krishna, P. Poizat, and G. Salaün. VBPMN: Automated Verification of BPMN Processes. In *Proc. of IFM'17*, volume 10510 of *LNCS*, pages 323–331. Springer, 2017.
12. S. Leemans, D. Fahland, and W. van der Aalst. Scalable Process Discovery and Conformance Checking. *Software & Systems Modeling*, 17(2):599–631, 2018.
13. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
14. N. Netjes, W. van der Aalst, and H. Reijers. Analysis of Resource-Constrained Processes with Colored Petri Nets. In *Proc. of CPN*, volume 576 of *DAIMI*, pages 251–266, 2005.
15. C. Oliveira, R. Lima, H. Reijers, and J. Ribeiro. Quantitative Analysis of Resource-Constrained Business Processes. *Trans. on Syst., Man, and Cybern.*, 42(3):669–684, 2012.
16. P. C. Ölveczky and J. Meseguer. Semantics and Pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.
17. M. Polato, A. Sperduti, A. Burattin, and M. de Leoni. Time and activity sequence prediction of business process instances. *Computing*, 100(9):1005–1031, 2018.
18. S. Sperl, G. Havur, S. Steyskal, C. Cabanillas, A. Polleres, and A. Haselböck. Resource Utilization Prediction in Decision-Intensive Business Processes. In *Proc. of SIMPDA*, CEUR Workshop Proceedings, pages 128–141, 2017.
19. W. van der Aalst, R. De Masellis, C. Di Francescomarino, and C. Ghidini. Learning Hybrid Process Models from Events - Process Discovery Without Faking Confidence. In *Proc. of BPM'17*, volume 10445 of *LNCS*, pages 59–76. Springer, 2017.
20. C. Walck. Hand-Book on Statistical Distributions for Experimentalists. Technical Report SUF–PFY/96–01, Universitet Stockholms, Stockholm, Sept. 2007.

Evaluation of Logic Programs with Built-Ins and Aggregation: A Calculus for Bag Relations

Matthew Francis-Landau[0000−0002−4139−1111], Tim Vieira[0000−0002−2043−1073], and Jason Eisner[0000−0002−8861−0772]

Johns Hopkins University, Baltimore, MD 21218 USA
{mfl,timv,jason}@cs.jhu.edu

Abstract. We present a scheme for translating logic programs with built-ins and aggregation into algebraic expressions that denote bag relations over ground terms of the Herbrand universe. To evaluate queries against these relations, we develop an operational semantics based on term rewriting of the algebraic expressions. This approach can exploit arithmetic identities and recovers a range of useful strategies, including lazy strategies that defer work until it becomes both possible and necessary. Code is available at https://github.com/matthewfl/dyna-R.

Keywords: Logic Programming, Relational Algebra, Rewriting Systems

1 Introduction

We are interested in developing execution strategies for deductive databases whose defining rules make use of aggregation, recursion, and arithmetic. Languages for specifying such deductive databases are expressive enough that it can be challenging to answer queries against a given database. Term rewriting systems are an attractive approach because they can start with the query-database pair itself as an intensional description of the answer, and then attempt to rearrange it into a more useful extensional description such as a relational table. We will give a notation for algebraically constructing potentially infinite bag relations from constants and built-in relations by unions, joins, projection, aggregation, and recursion. We show how programs in existing declarative languages such as Datalog, pure Prolog, and Dyna can be converted into this notation.

We will then sketch a term rewriting system for simplifying these algebraic expressions, which can in principle be used to answer queries against a bag relation. Term rewriting naturally handles materialization, delayed constraints, constraint propagation, short-circuit evaluation, lazy iteration, and enumerative strategies such as nested-iterator join.

There remains a practical challenge (which is beyond the scope of this paper): determining which rewrites to apply and when, much as in automated theorem proving. Our current implementation is essentially a priority rewrite

system with heuristic priorities and memoization.[1] While an improved execution engine is work in progress, our design includes fair (breadth-first) nondeterministic search, polyvariant specialization through generating new rewrite rules and programmable reduction strategies, static analysis by abstract interpretation, and guess-check-update strategies to solve recursive systems of constraints.

1.1 Approach

Dyna [4] is a generalization of Datalog [1,7] and pure Prolog [3,2]. Our methods apply to all three of these logic programming languages (and more trivially to languages like SQL that can be written in standard relational algebra).

We are given a Herbrand universe $\mathcal{G}$ of ground terms. A Dyna program serves to define a partial map from $\mathcal{G}$ to $\mathcal{G}$, which may be regarded as a set of key-value pairs. Datalog and Prolog are similar, but they can map a key only to `true`, so the program serves only to define the set of keys.

Given a program, a user may query the value of a specific key (ground term). More generally, a user may use a non-ground term to query the values of all keys that match it, so that the answer is itself a set of key-value pairs.

A set of key-value pairs—either a program or the answer to a query—may be interpreted as a relation on two $\mathcal{G}$-valued variables. Our method in this paper will be to describe the desired relation algebraically, building up the description from simpler relations using a relational algebra. These simpler relations can be over any number of variables; they may or may not have functional dependencies as the final key-value relation does; and they may be bag relations, i.e., a given tuple may appear in the relation more than once, or even infinitely many times. Given this description of the desired relation, we will use rewrite rules to simplify it into a form that is more useful to the user. Although we use a **term rewriting system**, we refer to the descriptions being rewritten as **R**-exprs (relational expressions) to avoid confusion with the terms of the object language, Dyna.

1.2 Dyna Examples

To illustrate the task, we first briefly give a couple of examples (adapted from [9]) of useful Dyna programs and queries against them. In §4, we will sketch how to translate Dyna programs into our algebra.

First, we start with a canonical Datalog program written in standard Dyna notation to compute the shortest path in a graph. Additional rules not shown here define the values of `start` and the various `edge` terms.

```
1 path(start) min= 0.
2 path(Sink)  min= path(Source) + edge(Source, Sink).
```

[1] This is already more flexible than Prolog's SLD resolution, which confines its attention at any time to a specific subgoal term and will crash with an "instantiation fault" if that subgoal cannot be rewritten.

This program defines a map with keys such as `edge("albany", "buffalo")`, whose value is the distance from Albany to Buffalo, and keys such as `path("chicago")`, whose value is the total length of the shortest path from `start` to Chicago.

The second rule implies that the value of `path("chicago")` is the minimum value achieved by `path(Source) + edge(Source, "chicago")` for any instantiation of the variable `Source`. If there are no instantiations of `Source` such that `path(Source)` and `edge(Source, "chicago")` both have values, then `path("chicago")` is not a key in the database at all, meaning that Chicago is not reachable at all from `start`.[2]

How is this database used? A user who is located at `start` might query `path("chicago")` to find out how far away it is, or they might query `path(Y)` to find all reachable cities `Y` along with how far away they are. Other queries would return other objects in the database, such as edges.

In the above example, `Sink` and `Source` may range over only a finite set of cities. However, we can easily encounter cases that define infinite relations, as in the following program that runs a simple convolutional neural network:

```
σ(X) = 1/(1+exp(-X)).                          % define sigmoid function
out(J) = σ(in(J)).                              % apply sigmoid function
in(J) += out(I) * edge(I,J).                    % vector-matrix product
in(input(X,Y)) += pixel_brightness(X,Y)                 % external input
loss += (out(J) - target(J))**2.            % L2 loss of the predictions
edge(input(X,Y),hidden(X+DX,Y+DY)) = weight_conv(DX,DY).       % layer 1
edge(hidden(X,Y),output(Z)) = weight_output(X,Y,Z).            % layer 2
weight_output(X,Y,Z) := random(*,-1,1).               % init with random
weight_conv(DX,DY) := random(*,-1,1) for DX:-4..4, DY:-4..4.
```

Without giving a detailed exposition of this program, we point out that it again has `edge` keys, which specify the weighted edges of a neural network. This time, however, the rules that specify such edges define infinitely many of them, in a convolutional structure on an infinite set of neurons.

As a result, a query `edge(I,J)` must return a description of an infinite set of edges. Even so, only finitely many of these edges contribute to the value of `loss`, provided that the input image specifies `pixel_brightness(X,Y)` at only finitely many `(X,Y)` coordinates, and the loss function specifies `target` values for only finitely many neurons. As a result, a clever system will be able to answer a query for `loss` in finite time, essentially by finding the paths between the pixels and the targets (which requires addition/subtraction of `DX` and `DY`) and by determining which of the infinitely many keys `σ(X)` need to compute their values in order to find the `out` activations of the neurons on these paths.

A version of this program indeed runs in our present implementation, although there is not space here to work through such a detailed example. Broadly speaking, the implementation unrolls the definition of `loss` until it is possible to apply rewrites that can make progress on simplifying the definition, for example, by performing arithmetic on known quantities.

[2] Unless `"chicago"` happens to be the value of the `start` key, in which case the first rule comes into play as well. In this case 0 is included in the minimum, so Chicago is always reachable with total distance of at most 0.

2 Syntax and Semantics of **R**-exprs

Let $\mathcal{G}$ be the Herbrand universe of **ground terms** built from a given set $\mathcal{F}$ of ranked functors. We treat constants (including numeric constants) as 0-ary functors. Let $\mathcal{M} = \mathbb{N} \cup \{\infty\}$ be the set of **multiplicities**. A simple definition of a **bag relation** [8] would be a map $\mathcal{G}^n \to \mathcal{M}$ for some n. Such a map would assign a multiplicity to each possible *ordered n-tuple* of ground terms. However, we will use names rather than positions to distinguish the roles of the n objects being related: in our scheme, the n tuple elements will be named by variables.

Let $\mathcal{V}$ be a distinguished set of **variables**. An **environment** $E : \mathcal{V} \mapsto \mathcal{G}$ is a partial function from variables $\mathcal{V}$ to ground terms.

Below, we will inductively define the set $\mathcal{R}$ of **R**-exprs. The reader may turn ahead to later sections to see some examples. Each **R**-expr `R` has a finite set of **free variables** vars(`R`) $\subseteq \mathcal{V}$, namely the variables that appear in `R` in positions where they are not bound by an operator such as `proj` or `sum`. The idea is for `R` to specify a bag relation over domain $\mathcal{G}$, with columns named by vars(`R`).

We will also inductively define the **semantic interpretation function** $[\![\cdot]\!]_E$, which assigns a multiplicity $[\![\texttt{R}]\!]_E$ to any **R**-expr `R` such that vars(`R`) $\subseteq$ domain(E).

For any $\mathcal{U} \subseteq \mathcal{V}$, the environments with domain $\mathcal{U}$ are just the possible tuples over $\mathcal{G}$ whose elements are named by $\mathcal{U}$. If vars(`R`) $\subseteq \mathcal{U}$, we can dually regard `R` as inducing a function $E \mapsto [\![\texttt{R}]\!]_E$ from these tuples to multiplicities. This is the sense in which `R` specifies a bag relation. More precisely, for each $\mathcal{U}$, `R` specifies a version of the relation whose column names are $\mathcal{U}$ but where `R` constrains only the columns vars(`R`). The other columns can take any values in $\mathcal{G}$. A tuple's multiplicity never depends on its values in those other columns, since our definition of $[\![\cdot]\!]_E$ will ensure that $[\![\texttt{R}]\!]_E$ depends only on the restriction of E to vars(`R`).)

We say that `T` is a **term** if $\texttt{T} \in \mathcal{V}$ or $\texttt{T} = f(\texttt{T}_1, \ldots, \texttt{T}_n)$ where $f \in \mathcal{F}$ has rank n and $\texttt{T}_1, \ldots, \texttt{T}_n$ are also terms. Terms typically appear in the object language (e.g., Dyna) as well as in our meta-language (**R**-exprs). Let $\mathcal{T} \supseteq \mathcal{G}$ be the set of terms. Let vars(`T`) be the set of vars appearing in `T`, and extend E in the natural way over terms `T` for which vars(`T`) $\subseteq$ domain(E): $E(f(\texttt{T}_1, \ldots, \texttt{T}_n)) = f(E(\texttt{T}_1), \ldots, E(\texttt{T}_n))$.

We now define $[\![\texttt{R}]\!]_E$ for each type of **R**-expr `R`, thus also presenting the different types of **R**-exprs. First, we have **equality constraints** between non-ground terms, which are true in an environment that grounds those terms to be equal. True is represented by multiplicity 1, and false by multiplicity 0.

1. $[\![\texttt{T=U}]\!]_E =$ **if** $E(\texttt{T}) = E(\texttt{U})$ **then** 1 **else** 0, where $\texttt{T}, \texttt{U} \in \mathcal{T}$

We also have **built-in constraints**, such as

2. $[\![\texttt{plus(I,J,K)}]\!]_E =$ **if** $E(\texttt{I}) + E(\texttt{J}) = E(\texttt{K})$ **then** 1 **else** 0, where $\texttt{I}, \texttt{J}, \texttt{K} \in \mathcal{T}$
   (use this constraint only in type-safe environments: $E(\texttt{I}), E(\texttt{J}), E(\texttt{K}) \in \mathbb{R}$)

The above **R**-exprs are said to be **constraints** because they always have multiplicity 1 or 0 in any environment. Taking the **union** of **R**-exprs via `+` may yield larger multiplicities:

3. $[\![\texttt{R+S}]\!]_E = [\![\texttt{R}]\!]_E + [\![\texttt{S}]\!]_E$, where $\texttt{R}, \texttt{S} \in \mathcal{R}$

The **R**-expr `0` denotes the empty bag relation, and more generally, $\mathtt{M} \in \mathcal{M}$ denotes the bag relation that contains `M` copies of the empty tuple:

4. $[\![\mathtt{M}]\!]_E = \mathtt{M}, \qquad$ where $\mathtt{M} \in \mathcal{M}$

When we join two bag relations, we must use multiplication $*$ to combine their multiplicities [8]:

5. $[\![\mathtt{R*S}]\!]_E = [\![\mathtt{R}]\!]_E \cdot [\![\mathtt{S}]\!]_E, \qquad$ where $\mathtt{R}, \mathtt{S} \in \mathcal{R}$

We can regard both `R` and `S` as bag relations over columns $\mathcal{U} = \text{vars}(\mathtt{R}) \cup \text{vars}(\mathtt{S}) = \text{vars}(\mathtt{R+S}) = \text{vars}(\mathtt{R*S})$. The names **intersect**, **join** (or **equijoin**), and **Cartesian product** are conventionally used for the cases of `R*S` where (respectively) $\text{vars}(\mathtt{R}) = \text{vars}(\mathtt{S})$, $|\text{vars}(\mathtt{R}) \cap \text{vars}(\mathtt{S})| = 1$, and $|\text{vars}(\mathtt{R}) \cap \text{vars}(\mathtt{S})| = 0$. As a special case of cross product, notice that `R*3` denotes the same bag relation as `R+R+R`.

We next define **projection**, which removes a column `X` from a bag relation, summing the multiplicities of rows that have thus become equal. When we translate a logic program into an **R**-expr (§4), we will generally apply projection operators to each rule to eliminate its local variables.

6. $[\![\mathtt{proj(X,R)}]\!]_E = \sum_{x \in \mathcal{G}} [\![\mathtt{R}]\!]_{E[\mathtt{X}=x]}$
   where $\mathtt{X} \in \mathcal{V}, \mathtt{R} \in \mathcal{R}$, and where $E[\mathtt{X} = x]$ means a version of E that has been modified to set $E(\mathtt{X}) = x$

Projection collapses each group of rows that are identical except for their value of `X`. **Summation** does the same, but instead of adding up the multiplicities of the rows in each possibly empty group, it adds up their `X` values to get a `Y` value for the new row, which has multiplicity 1. Thus, it removes column `X` but introduces a new column `Y`. The summation operator is defined as follows (note that $\mathtt{sum} \notin \mathcal{F}$):

7. $[\![\mathtt{A=sum(X,R)}]\!]_E =$ **if** $E(\mathtt{A}) = \sum_{x \in \mathcal{G}} x * [\![\mathtt{R}]\!]_{E[\mathtt{X}=x]}$ **then** 1 **else** 0
   where $\mathtt{A}, \mathtt{X} \in \mathcal{V}, \mathtt{R} \in \mathcal{R}$, and the $*$ in the summand means that we sum up $[\![\mathtt{R}]\!]_{E[\mathtt{X}=x]}$ copies of the value x. If there are no summands, the result of the $\sum \cdots$ is defined to be the identity element $\mathsf{id}_{\mathtt{sum}}$.

Notice that an **R**-expr of this form is a constraint. `sum` is just one type of **aggregation operator**, based on the binary + operation and its identity element $\mathsf{id}_{\mathtt{sum}} = 0$. In exactly the same way, one may define other aggregation operators such as `min`, based on the binary min operation and its identity element $\mathsf{id}_{\mathtt{min}} = \infty$. Variants of these will be used in §4 to implement the aggregations in `+=` and `min=` rules like those in §1.2.

In projections `proj(X,R)` and aggregations such as `sum(X,R)` and `min(X,R)`, we say that occurrences of `X` within `R` are **bound** by the projection or aggregation operator,[3] so that they are not in the vars of the resulting **R**-expr. However, the most basic aggregation operator does not need to bind a variable:

8. $[\![\mathtt{M=count(R)}]\!]_E =$ **if** $E(\mathtt{M}) = [\![\mathtt{R}]\!]_E$ **then** 1 **else** 0

[3] But notice that `sum(X,R)` does not correspond to $\sum_{\mathtt{X}} \cdots$ but rather to $\sum_{\text{row} \in \mathtt{R}} \text{row}[\mathtt{X}]$.

In effect, M=count(R) is a version of R that changes every tuple's multiplicity to 1 but records its original multiplicity in a new column M. It is equivalent to M=sum(N,(N=1)*R) (where N $\notin$ vars(R)), but we define it separately here so that it can later serve as an intermediate form in the operational semantics.

Finally, it is convenient to augment the built-in constraint types with **user-defined** relation types. Choose a new functor of rank n that is $\notin \mathcal{F}$, such as f, and choose some **R**-expr $\mathtt{R_f}$ with vars($\mathtt{R_f}$) $\subseteq \{\mathtt{X}_1, \ldots, \mathtt{X}_n\}$ (which are n distinct variables) to serve as the **definition** (macro expansion) of f. Now define

9. $[\![\mathtt{f}(\mathtt{T}_1, \ldots, \mathtt{T}_n)]\!]_E = [\![\mathtt{R_f}\{\mathtt{X}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{X}_n \mapsto \mathtt{T}_n\}]\!]_E$ where $\mathtt{T}_1, \ldots, \mathtt{T}_n \in \mathcal{T} \cup \mathcal{R}$
   The $\{\mapsto\}$ notation denotes substitution for variables, where bound variables of $\mathtt{R_f}$ are renamed to avoid capturing free variables of the $\mathtt{T}_i$.

With user-defined relation types, it is possible for a user to write **R**-exprs that are circularly defined in terms of themselves or one another (similarly to a `let rec` construction in functional languages). Indeed, a Dyna program normally does this. In this case, the definition of $[\![\cdot]\!]_E$ is no longer a well-founded inductive definition. Nonetheless, we can still interpret the $[\![\cdots]\!]_E = \cdots$ equations in the numbered points above as *constraints* on $[\![\cdot]\!]_E$, and attempt to solve for a semantic interpretation function $[\![\cdot]\!]_E$ that satisfies these constraints [4]. Some circularly defined **R**-exprs are constructed so as to have unique solutions, but this is not the case in general.

3 Rewrite Rules

Where the previous section gave a denotational semantics for **R**-exprs, we now sketch an operational semantics. The basic idea is that we can use rewrite rules to simplify an **R**-expr until it is either a finite materialized relation—a list of tuples—or has some other convenient form. All of our rewrite rules are semantics-preserving, but some may be more helpful than others. For some **R**-exprs that involve infinite bag relations, there may be no way to eliminate all built-in constraints or aggregation operations. The reduced form then includes delayed constraints (just as in constraint logic programming) or delayed aggregators. Even so, conjoining this reduced form with a query can permit further simplification; therefore, some queries may still yield simple answers.

3.1 Finite Materialized Relations

We may express the finite bag relation shown at left by a simple **sum-of-products** **R**-expr, shown at the right. In this example, the ground values being related are integers.

$$g = \left[\begin{array}{cc} \mathtt{X}_1 & \mathtt{X}_2 \\ \hline 1 & 1 \\ 2 & 6 \\ 2 & 7 \\ 2 & 7 \\ 5 & 7 \end{array}\right] \xrightarrow{\text{to } \mathbf{R}\text{-expr}} \begin{aligned} \mathtt{R_g} = \; & (\ (\mathtt{X}_1 = 1) * (\mathtt{X}_2 = 1) \\ & + (\mathtt{X}_1 = 2) * (\mathtt{X}_2 = 6) \\ & + (\mathtt{X}_1 = 2) * (\mathtt{X}_2 = 7) \\ & + (\mathtt{X}_1 = 2) * (\mathtt{X}_2 = 7) \\ & + (\mathtt{X}_1 = 5) * (\mathtt{X}_2 = 7)\ ) \end{aligned} \tag{1}$$

We see that each individual row of the table (tuple) translates into a product ($*$) of several (`Variable` = `value`) expressions, where the variable is the column's name and the value is the cell in the table. To encode multiple rows, the **R**-expr simply adds the **R**-exprs for the individual rows. When evaluated in a given environment, the **R**-expr is a sum of products of multiplicities. But abstracting over possible environments, it represents a union of Cartesian products of 1-tuples, yielding a bag relation.

We may use this **R**-expr as the basis of a new user-defined **R**-expr type `g` (case 9 of §2) by taking its definition $\texttt{R}_\texttt{g}$ to be this **R**-expr. Our **R**-exprs can now include constraints such as `g(A,B)` or `g(A,7)`. When adding a new case in the denotational semantics in this way, we always match it in the operational semantics by introducing a corresponding rewrite rule $\texttt{g}(\texttt{X}_1, \texttt{X}_2) \to \texttt{R}_\texttt{g}$.

A sum-of-products **R**-expr simply enumerates the tuples of a bag relation, precisely as a boolean expression in disjunctive normal form (that is, an "or-of-ands" expression) enumerates the satisfying assignments. Just as in the boolean case, a disjunct does not have to constrain every variable: it may have "don't care" elements. For example, `(A=1)*(B=1) + (A=2)` describes an infinite relation because the second disjunct `A=2` is true for any value of `B`.

3.2 Equality Constraints and Multiplicity Arithmetic

We may wish to query whether the relation g in the above section relates 2 to 7. Indeed it does—and twice. We may discover this by considering `g(2,7)`, which rewrites immediately via substitution to an **R**-expr that has no variables at all (`(2=1)*(7=1)+` $\cdots$), and finding that this further reduces to the multiplicity `2`.

How does this reduction work? First, we need to know how to evaluate the equality constraints: we need to rewrite `2=1` $\to$ `0` but `2=2` $\to$ `1`. The necessary rewrite rules are special cases of the following structural unification rules:

$\texttt{(}f\texttt{(}\texttt{U}_1\texttt{,}\ldots\texttt{,}\texttt{U}_n\texttt{)=}g\texttt{(}\texttt{V}_1\texttt{,}\ldots\texttt{,}\texttt{V}_m\texttt{))} \to \texttt{0}$ **if** $f, g \in \mathcal{F}$ **and** $(f,n) \neq (g,m)$ ▷*functor clash*

$\texttt{(}f\texttt{(}\texttt{U}_1\texttt{,}\ldots\texttt{,}\texttt{V}_n\texttt{)=}f\texttt{(}\texttt{V}_1\texttt{,}\ldots\texttt{,}\texttt{V}_n\texttt{))} \to \texttt{(}\texttt{U}_1\texttt{=}\texttt{V}_1\texttt{)} * \cdots * \texttt{(}\texttt{U}_n\texttt{=}\texttt{V}_n\texttt{)}$ **if** $f \in \mathcal{F}$ **and** $n \geq 0$

`(T=X)` $\to$ `(X=T)` **if** `X` $\in \mathcal{V}$ **and** `T` $\in \mathcal{T}$ ▷*put var on left to match rules below*

`(X=X)` $\to$ `1` ▷*true for every value of* `X`

`(X=T)` $\to$ `0` **if** `X` $\in \mathcal{V}$ **and** `T` $\in \mathcal{T}$ **and** `V` $\in \text{vars}($`T`$)$ ▷*not true for any* `X` *(occurs check)*

We now have an arithmetic expression, which we can simplify to the multiplicity `2` via rewrites that implement basic arithmetic on multiplicities $\mathcal{M}$:

`M + N` $\to$ `L` **if** `M,N` $\in \mathcal{M}$ **and** `M+N=L`; `M * N` $\to$ `L` **if** `M,N` $\in \mathcal{M}$ **and** `M*N=L`

Above, we relied on the definition of the new relation type `g`, which allowed us to request a specialization of $\texttt{R}_\texttt{g}$. Do we need to make such a definition in order to query a given bag relation? No: we may do so by conjoining the **R**-expr with additional constraints. For example, to get the multiplicity of the pair $(2,7)$ in $\texttt{R}_\texttt{g}$, we may write $\texttt{R}_\texttt{g}\texttt{*(}\texttt{X}_1\texttt{=2)*(}\texttt{X}_2\texttt{=7)}$. This filters the original relation to just the pairs that match $(2,7)$, and simplifies to $\texttt{2*(}\texttt{X}_1\texttt{=2)*(}\texttt{X}_2\texttt{=7)}$. To accomplish this simplification, we need to use the following crucial rewrite:

`(X=T)*R` $\to$ `(X=T)*R{X` $\mapsto$ `T}` **if** `X` $\in \mathcal{V}$ **and** `T` $\in \mathcal{T}$ ▷*equality propagation*

As a more interesting example, the reader may consider simplifying the query $\texttt{R}_\texttt{g}\texttt{*lessthan(}\texttt{X}_2\texttt{,7)}$, which uses a built-in inequality constraint (see §3.4).

3.3 Joining Relations

Analogous to eq. (1), we define a second tabular relation f with a rewrite rule $\texttt{f}(\texttt{X}_1, \texttt{X}_2) \to \texttt{R}_\texttt{f}$.

$$f = \left[\begin{array}{cc} \texttt{X}_1 & \texttt{X}_2 \\ \hline 1 & 2 \\ 3 & 4 \end{array}\right] \xrightarrow{\text{to } \mathbf{R}\text{-expr}} \texttt{R}_\texttt{f} = \begin{array}{r} (\texttt{X}_1 = 1) * (\texttt{X}_2 = 2) \\ + (\texttt{X}_1 = 3) * (\texttt{X}_2 = 4) \end{array} \tag{2}$$

We can now consider simplifying the **R**-expr `f(I,J) * g(J,K)`, which the reader may recognize as an equijoin on f's second column and g's first column.[4] Notice that the **R**-expr has *renamed* these columns to its own free variables (`I`, `J`, `K`). Reusing the variable `J` in each factor is what gives rise to the join on the relevant column. (Compare `f(I,J) * g(J',K)`, which does not share the variable and so gives the Cartesian product of the f and g relations.)

We can "materialize" the equijoin by reducing it to a sum-of-products form as before, if we wish: `(I=1)*(J=2)*(K=6) + 2*(I=1)*(J=2)*(K=7)`

To carry out such simplifications, we use the fact that multiplicities form a commutative semiring under `+` and `*`. Since any **R**-expr evaluates to a multiplicity, these rewrites can be used to rearrange unions and joins of **R**-exprs $\texttt{Q}, \texttt{R}, \texttt{S} \in \mathcal{R}$:

`1 * R` $\leftrightarrow$ `R`		▷*multiplicative identity*
`0 * R` $\leftrightarrow$ `0`		▷*multiplicative annihilation*
`0 + R` $\leftrightarrow$ `R`		▷*additive identity*
∞ `* R` $\to \infty$ **if** `R` $\in \mathcal{M}$ **and** `R` $>$ `0`		▷*absorbing element*
∞ `+ R` $\to \infty$		▷*absorbing element*
`R + S` $\leftrightarrow$ `S + R`;	`R * S` $\leftrightarrow$ `S * R`	▷*commutativity*
`Q + (R + S)` $\leftrightarrow$ `(Q + R) + S`;	`Q * (R * S)` $\leftrightarrow$ `(Q * R) * S`	▷*associativity*
`Q * (R + S)` $\leftrightarrow$ `Q * R + Q * S`		▷*distributivity*
`R * M` $\to$ `R + (R * N)` **if** `M,N` $\in \mathcal{M}$ **and** `(1+N` $\to$ `M)`		▷*implicitly does* `M` $\to$ `1+N`
`R` $\leftrightarrow$ `R * R` **if** `R` is a constraint		▷*as defined in §2*

We can apply some of these rules to simplify our example as follows:

```
f(I,J)*g(J,K)
→ ((I=1)*(J=2)+(I=3)*(J=4)) * g(J,K)                    ▷eq. (2)
→ (I=1)*(J=2)*g(J,K) + (I=3)*(J=4)*g(J,K)               ▷distributivity
→ (I=1)*(J=2)*g(2,K) + (I=3)*(J=4)*g(4,K)               ▷equality propagation
→ (I=1)*(J=2)*( (K=6)+(K=7)*2 ) + (I=3)*(J=4)*0         ▷via eq. (1)
→ (I=1)*(J=2)*( (K=6)+(K=7)*2 )                         ▷annihilation
→ (I=1)*(J=2)*(K=6) + (I=1)*(J=2)*(K=7)*2               ▷distributivity
```

Notice that the factored intermediate form `(I=1)*(J=2)*( (K=6)*1 + (K=7)*2 )` is more compact than the final sum of products, and may be preferable in some settings. In fact, it is an example of a **trie** representation of a bag relation. Like the root node of a trie, the expression partitions the bag of (`I`,`J`,`K`) tuples into disjuncts according to the value of `I`. Each possible value of `I` (in this case only `I=1`) is multiplied by a trie representation of the bag of (`J`,`K`) tuples that can co-occur

[4] This notation may be familiar from Datalog, except that we are writing the conjunction operation as `*` rather than with a comma, to emphasize the fact that we are multiplying multiplicities rather than merely conjoining booleans.

with this `I`. That representation is a sum over possible values of `J` (in this case only `J=2`), which recurses again to a sum over possible values of `K` (`K=6` and `K=7`). Finally, the multiplicities `1` and `2` are found at the leaves. A trie-shaped **R**-expr generally has a smaller branching factor than a sum-of-products **R**-expr. As a result, it is comparatively fast to query it for all tuples that strongly restrict `I` or `(I,J)` or `(I,J,K)`, by narrowing down to the matching summand(s) at each node. For example, multiplying our example trie by the query `I=5` gives an **R**-expr that can be immediately simplified to `0`, as the single disjunct (for `I=1`) does not match.

That example query also provides an occasion for a larger point. This trie simplification has the form `(I=5)*(I=1)*R`, an expression that in general may be simplified to `0` on the basis of the first two factors, without spending any work on simplifying the possibly large expression `R`. This is an example of **short-circuiting** evaluation—the same logic that allows a SAT solver or Prolog solver to backtrack immediately upon detecting a contradiction.

3.4 Rewrite Rules for Built-In Constraints

Built-in constraints are an important ingredient in constructing infinite relations. While they are not the only method,[5] they have the advantage that libraries of built-in constraints such as `plus(I,J,K)` (case 2 of §2) usually come with rewrite rules for reasoning about these constraints [6]. Some of the rewrite rules invoke opaque procedural code.

Recall that the arguments to a `plus` constraint are terms, typically either variables or numeric constants. Not all `plus` constraints can be rewritten, but a library should provide at least the following cases:

$$\texttt{plus(I,J,K)} \rightarrow \underbrace{\mathbb{I}(\texttt{I} + \texttt{J} = \texttt{K})}_{\in\{0,1\}} \textbf{ if } \texttt{I},\texttt{J},\texttt{K} \in \mathbb{R}$$

$$\texttt{plus(I,J,X)} \rightarrow \texttt{(X=I + J)} \textbf{ if } \texttt{I},\texttt{J} \in \mathbb{R} \textbf{ and } \texttt{X} \in \mathcal{V}$$

$$\texttt{plus(I,X,K)} \rightarrow \texttt{(X=K - I)} \textbf{ if } \texttt{I},\texttt{K} \in \mathbb{R} \textbf{ and } \texttt{X} \in \mathcal{V}$$

$$\texttt{plus(X,J,K)} \rightarrow \texttt{(X=}\underbrace{\texttt{K - J}}_{\in\mathbb{R}}\texttt{)} \textbf{ if } \texttt{J},\texttt{K} \in \mathbb{R} \textbf{ and } \texttt{X} \in \mathcal{V}$$

The **R**-expr `R = proj(J,plus(I,3,J)*plus(J,4,K))` represents the infinite set of `(I,K)` pairs such that `K = (I+3)+4` arithmetically. (The intermediate temporary variable `J` is projected out.) The rewrite rules already presented (plus a rewrite rule from §3.5 below to eliminate `proj`) suffice to obtain a satisfactory answer to the query `I=2` or `K=9`, by reducing either `(I=2)*R` or `R*(K=9)` to `(I=2)*(K=9)`.

On the other hand, if we wish to reduce `R` itself, the above rules do not apply. In the jargon, the two `plus` constraints within `R` remain as **delayed constraints**, which cannot do any work until more of their variable arguments are replaced by constants (e.g., due to equality propagation from a query, as above).

We can do better in this case with a library of additional rewrite rules that implement standard axioms of arithmetic [6], in particular the associative law. With these, `R` reduces to `plus(I,7,K)`, which is a simpler description of this infinite relation. Such rewrite rules are known as **constraint propagators**. Other

[5] Others are structural equality constraints and recursive user-defined constraints.

useful examples concerning `plus` include `plus(0,J,K)` → `K=J` and `plus(I,J,J)` → `(I=0)`, since unlike the rules at the start of this section, they can make progress even on a single `plus` constraint whose arguments include more than one variable. Similarly, some useful constraint propagators for the `lessthan` relation include `lessthan(J,J)` → `0`; the transitivity rule `lessthan(I,J)*lessthan(J,K)` → `lessthan(I,J)*lessthan(J,K)*lessthan(I,K)`; and `lessthan(0,I)*plus(I,J,K)` → `lessthan(0,I)*plus(I,J,K)*lessthan(J,K)`. The integer domain can be **split** by rules such as `int(I)` → `int(I)*(lessthan(0,I)+lessthan(I,1))` in order to allow case analysis of, for example, `int(I)*myconstraint(I)`. All of these rules apply even if their arguments are variables, so they can apply early in a reduction before other rewrites have determined the values of those variables. Indeed, they can sometimes short-circuit the work of determining those values.

Like all rewrites, built-in rewrites `R` → `S` must not change the denotation of `R`: they ensure $[\![R]\!]_E = [\![S]\!]_E$ for all E. For example, `lessthan(X,Y)*lessthan(Y,X)` →* `0` is semantics-preserving because both forms denote the empty bag relation.

3.5 Projection

Projection is implemented using the following rewrite rules. The first two rules make it possible to push the `proj(X,...)` operator down through the sums and products of `R`, so that it applies to smaller subexpressions that mention `X`:

`proj(X,R+S)` ↔ `proj(X,R) + proj(X,S)` ▷*distributivity over +*
`proj(X,R*S)` ↔ `R*proj(X,S)` **if** `X` ∉ vars(`R`) ▷*see also the* `R*`∞ *rule below*

Using the following rewrite rules, we can then eliminate the projection operator from smaller expressions whose projection is easy to compute. (In other cases, it must remain as a delayed operator.) How are these rules justified? Observe that `proj(X,R)` in an environment E denotes the number of `X` values that are consistent with E's binding of `R`'s other free variables. Thus, we may safely rewrite it as another expression that always achieves the same denotation.

`proj(X,(X=T))` → 1 **if** `T` ∈ $\mathcal{T}$ **and** `T` does not contain `X` ▷*occurs check*
`proj(A,(A=sum(X,R)))` → 1 **if** `A` ∉ vars(`R`) ▷*cardinality of an aggregated variable*
`proj(X,R)` → `R*`∞ **if** `X` ∉ vars(`R`) ▷*cardinality of an unconstrained variable*
`proj(X,bool(X))` → 2 ▷*cardinality of a variable given a certain unary constraint*
`proj(X,int(X))` → ∞ ▷*cardinality of a variable given a certain unary constraint*
`proj(X,proj(Y,nand(X,Y)))` → 3 ▷*card. of a pair given a certain binary constraint*

As a simple example, let us project column `K` out of the table `g(J,K)` from eq. (1).

```
                    proj(K,((J=1) * (K=1)        ( (J=1) * proj(K, (K=1))
                          + (J=2) * (K=6)        + (J=2) * proj(K, (K=6))
 proj(K, g(J,K)) →        + (J=2) * (K=7)   →    + (J=2) * proj(K, (K=7))
                          + (J=2) * (K=7)        + (J=2) * proj(K, (K=7))
                          + (J=5) * (K=7)  ))    + (J=5) * proj(K, (K=7)) )
→ (J=1) + (J=2)*3 + (J=5)
```

When multiple projection operators are used, we may push them down independently of each other, since they commute:

`proj(X, proj(Y, R))` → `proj(Y, proj(X, R))`

3.6 Aggregation

The simple `count` aggregator from §2 is implemented with the following rewrite rules, which resemble those for `proj`:

```
M=count(R+S) → proj(L,(L=count(R)) * proj(N,(N=count(S)) * plus(L,N,M)))))
M=count(R*S) → proj(L,(L=count(R)) * proj(N,(N=count(S)) * times(L,N,M)))))
    if vars(R) ∩ vars(S) = ∅
M=count(N) → (M=N) if N ∈ ℳ
```

In the first two rules, `L` and `N` are new bound variables introduced by the right-hand side of the rule. (When the rewrite rule is applied, they will—as is standard—potentially be renamed to avoid capturing free variables in the arguments to the left-hand side.) They serve as temporary registers. The third rule covers the base case where the expression has been reduced to a constant multiplicity: e.g.,

```
M=count(5=5) → M=count(1) → M=1
M=count(plus(I,J,J)*(I=5)) → M=count((I=0)*(I=5)) →* M=count(0) → M=0
```

The following rewrite rules implement `sum`. (The rules for other aggregation operators are isomorphic.) The usual strategy is to rewrite `A=sum(X,R)` as a chain of `plus` constraints that maintain a running total. The following rules handle cases where `R` is expressed as a union of 0, 1, or 2 bag relations, respectively. (A larger union can be handled as a union of 2 relations, e.g., `(Q+R)+S`.)

```
A=sum(X, 0) → (A=id_sum)
A=sum(X, (X=T)) → (A=T) if T ∈ 𝒯 and T does not contain X        ▷occurs check
A=sum(X, R+S) → proj(B, (B=sum(X,R)) * proj(C, (C=sum(X,S)) * plus(B,C,A)))
```

The second rule above handles only one of the base cases of 1 bag relation. We must add rules to cover other base cases, such as these:[6]

```
A=sum(X, (X=sum(Y,R))) → (A=sum(Y,R)) if X ∉ vars(R)
A=sum(X, (X=min(Y,R))) → (A=min(Y,R)) if X ∉ vars(R)
```

Most important of all is this case, which is analogous to the second rule of §3.5 and is needed to aggregate over sum-of-products constructions:

```
A=sum(X,R*S) ↔ sum_copies(R,B,A)*(B=sum(X,S))   if X ∉ vars(R)
```

Here, `sum_copies(M,B,A)` for `M` $\in \mathcal{M}$ constrains `A` to be the aggregation of `M` $\in \mathcal{M}$ copies of the aggregated value `B`. The challenge is that in the general case we actually have `sum_copies(R,B,A)`, so the multiplicity `M` may vary with the free variables of `R`. The desired denotational semantics are

$$[\![\texttt{sum\_copies(R,B,A)}]\!]_E = \textbf{ if } [\![\texttt{R}]\!]_E \cdot [\![\texttt{B}]\!]_E = [\![\texttt{A}]\!]_E \textbf{ then } 1 \textbf{ else } 0$$
$$[\![\texttt{min\_copies(R,B,A)}]\!]_E = \textbf{ if } ([\![\texttt{R}]\!]_E = 0 \textbf{ and } [\![\texttt{A}]\!]_E = \mathsf{id}_{\texttt{min}}) \textbf{ or } ([\![\texttt{R}]\!]_E > 0 \textbf{ and } [\![\texttt{A}]\!]_E = [\![\texttt{B}]\!]_E) \textbf{ then } 1 \textbf{ else } 0$$

where `R` $\in \mathcal{R}$ and `B`, `A` $\in \mathcal{T}$

[6] As in §3.5, we could also include special rewrites for certain aggregations that have a known closed-form result, such as certain series sums.

where we also show the interesting case of `min_copies(M,B,A)`, which is needed to help define the `min` aggregator. We can implement these by the rewrite rules

`sum_copies(R,B,A)` $\rightarrow$ `proj(M,(M=count(R))*times(M,B,A))` $\quad \triangleright$ *assumes* $\mathsf{id}_{\mathtt{sum}} = 0$
`min_copies(R,B,A)` $\rightarrow$ `proj(M,(M=count(R))*((M=0)*(A=`$\mathsf{id}_{\mathtt{min}}$`)+lessthan(0,M)*(A=B)))`

Identities concerning aggregation yield additional rewrite rules. For example, since multiplication distributes over $\sum$, summations can be merged and factored via `(B=sum(I,R))*(C=sum(J,S))*times(B,C,A)` $\leftrightarrow$ `A=sum(K,R*S*times(I,J,K))` provided that `I` $\in$ vars(`R`), `J` $\in$ vars(`S`), `K` $\notin$ vars(`R*S`), and vars(`R`) $\cap$ vars(`S`) $= \emptyset$. Other distributive properties yield more rules of this sort. Moreover, projection and aggregation operators commute if they are over different free variables.

To conclude this section, we now attempt aggregation over infinite streams. We wish to evaluate `A=exists(B,proj(I,peano(I)*myconstraint(I)*(B=true)))` to determine whether there exists any Peano numeral that satisfies a given constraint. Here `exists` is the aggregation operator based on the binary `or` operation.

`peano(I)` represents the infinite bag of Peano numerals, once we define a user constraint via the rewrite rule `peano(I)` $\rightarrow$ `(X=zero) + proj(J,(I=s(J))*peano(J))`. Rewriting `peano(I)` provides an opportunity to apply the rule again (to `peano(J)`). After $k \geq 0$ rewrites we obtain a representation of the original bag that explicitly lists the first k Peano numerals as well as a continuation that represents all Peano numerals $\geq k$:

$$\rightarrow \texttt{(X=zero)} + \cdots + (\texttt{X=}\underbrace{\texttt{s}(\cdots\texttt{s}}_{k-1 \text{ times}}\texttt{(zero)}\cdots)) + \overbrace{\texttt{proj(J,(X=}\underbrace{\texttt{s}(\cdots\texttt{s}}_{k-1 \text{ times}}\texttt{(J)}\cdots)))\texttt{*peano(J)}}^{\text{continuation}}$$

Rewriting the `exists` query over this $(k+1)$-way union results in a chain of k `or` constraints. If one of the first k Peano numerals in fact satisfies `myconstraint`, then we can "short-circuit" the infinite regress and determine our answer without further expanding the continuation, thanks to the useful rewrite `or(true,C,A)` $\rightarrow$ `(A=true)`, which can apply even while `C` remains unknown.

In general, one can efficiently aggregate a function of the Peano numerals by alternating between expanding `peano` to draw the next numeral from the iterator, and rewriting the aggregating **R**-expr to aggregate the value of the function at that numeral into a running "total." If the running total ever reaches an absorbing element a of the aggregator's binary operation—such as `true` for the `or` operation—then one will be able to simplify the expression to `A=` a and terminate. We leave the details as an exercise.

4 Translation of Dynabases to R-exprs

The translation of a Dyna program to a single recursive **R**-expr can be performed mechanically. We will illustrate the basic construction on the small contrived example below. We will focus on the first three rules, which define `f` in terms of `g`. The final rule, which defines `g`, will allow us to take note of a few subtleties.

```
12 f(X) += X*X.
13 f(4) += 3.
14 f(X) += g(X,Y).
15 g(4*C,Y) += C-1 for Y > 99.
```

Recall that a Dyna program represents a set of key-value pairs. `is(Key,Val)` is the conventional name for the key-value relation. The above program translates into the following user-defined constraint, which recursively invokes itself when the value of one key is defined in terms of the values of other keys.

```
is(Key,Val) → (Val=sum(Result,                ▷sum represents the += aggregator
    proj(X, (Key=f(X))*times(X,X,Result) )                    ▷f(X) += X*X.
  +         (Key=f(4))*(Result=3)                               ▷f(4) += 3.
  +proj(X, (Key=f(X))*proj(Y,is(g(X,Y),Result))) )          ▷f(X) += g(X,Y).
  +proj(C, proj(Y, proj(Temp,(Key=g(Temp,Y))*times(4,C,Temp))   ▷g(4*C,Y) +=
                *minus(C,1,Result)*lessthan(99,Y)     ▷... C-1 for Y > 99.
 ) * notnull(Val)         ▷notnull discards any pair whose Val aggregated nothing
```

Each of the 4 Dyna rules translates into an **R**-expr (as indicated by the comments above) that describes a bag of (`Key`,`Result`) pairs of ground terms. In each pair, `Result` represents a possible ground value of the rule's body (the Dyna expression to the right of the aggregator `+=`), and `Key` represents the corresponding grounding of the rule head (the term to the left of the aggregator), which will receive `Result` as an aggregand. Note that the same (`Key`,`Result`) pair may appear multiple times. Within each rule's **R**-expr, we project out the variables such as `X` and `Y` that appear locally within the rule, so that the **R**-expr's free variables are only `Key` and `Result`.[7]

Dyna mandates in-place evaluation of Dyna expressions that have values [4]. For each such expression, we create a new local variable to bind to the result. Above, the expressions such as `X*X`, `g(X,Y)`, `4*C`, and `C-1` were evaluated using `times`, `is`, `times`, and `minus` constraints, respectively, and their results were assigned to new variables `Result`, `Result`, `Temp`, and `Result`. Importantly, `g(X,Y)` refers to a key of the Dyna program itself, so the Dyna program translated into an `is` constraint whose definition recursively invokes `is(g(X,Y),Result)`.

The next step is to take the bag union (`+`) of these 4 bag relations. This yields the bag of all (`Key`,`Result`) pairs in the program. Finally, we wrap this combined bag in `Val=sum(Result,...)` to aggregate the `Result`s for each `Key` into the `Val` for that key. This yields a set relation with exactly one value for each key. For `Key=f(4)`, for example, the first and second rules will each contribute one `Result`, while the third rule will contribute as many `Result`s as the map has keys of the form `g(4,Y)`.[8]

We use `sum` as our aggregation operator because all rules in this program specify the `+=` aggregator. One might expect `sum` to be based on the binary operator that is implemented by the `plus` builtin, as described before, with identity element $\mathsf{id}_{\texttt{sum}} = 0$. There is a complication, however: in Dyna, a `Key` that has

[7] It is always legal to project out the local variables at the top level of the rule, e.g., `proj(X,proj(Y,...))` for rule 3. However, we have already seen rewrite rules that can narrow the scope of `proj(Y,...)` to a sub-**R**-expr that contains all the copies of `Y`. Here we display the version after these rewrites have been done.

[8] The reader should be able to see that the third Dyna rule will contribute infinitely many copies of `0`, one for each `Y > 99`. This is an example of multiplicity ∞. Fortunately, `sum_copies` (invoked when rewriting the `sum` aggregation operator) knows that summing any positive number of `0` terms—even infinitely many—will give `0`.

no Results should not in fact be paired with Val=0. Rather, this Key should not appear as a key of the final relation at all! To achieve this, we augment $\mathcal{F}$ with a new constant null (similar to Prolog's no), which represents "no results." We define $\mathsf{id}_{\mathtt{sum}} = \mathsf{null}$, and we base sum on a modified version of plus for which null is indeed the identity (so the constraints plus(null,X,X) and sum_copies(0,X,null) are both true for all X). All aggregation operators in Dyna make use of null in this way. Our Val=sum(Result,...) relation now obtains null (rather than 0) as the unique Val for each Key that has no aggregands. As a final step in expressing a Dyna program, we always remove from the bag all (Key, Val) pairs for which Val=null, by conjoining with a notnull(Val) constraint. This is how we finally obtain the **R**-expr above for is(Key,Val).

To query the Dyna program for the value of key f(4), we can reduce the expression is(f(4),Val), using previously discussed rewrite rules. We can get as far as this before it must carry out its own query g(4,Y):

```
proj(C, (C=sum(Result,proj(Y,is(g(4,Y),Result))))
       * plus(19,C,Val)                          ) * notnull(Val)
```

where the local variable C captures the total contribution from rule 3, and 19 is the total contribution of the other rules. To reduce further, we now recursively expand is(g(4,Y),Result) and ultimately reduce it to Result=0 (meaning that g(4,Y) turns out to have value 0 for all ground terms Y). proj(Y,Result=0) reduces to (Result=0)*∞—a bag relation with an infinite multiplicity—but then C=sum(Result,(Result=0)*∞) reduces to C=0 (via sum_copies, as footnote 8 noted). The full expression now easily reduces to Val=19, the correct answer.

What if the Dyna program has different rules with different aggregators? Then our translation takes the form

```
is(Key,Val) → Val=only(Val1,  (Val1=sum(Result, RSum))
                            + (Val1=min(Result, RMin))
                            + (Val1=only(Result, REq))
                            + ...                      ) * notnull(Val)
```

where RSum is the bag union of the translated += rules as in the previous example, RMin is the bag union of the translated min= rules, REq is the bag union of the translated = rules, and so on. The new aggregation operator only is based on a binary operator that has identity $\mathsf{id}_{\mathtt{only}} = \mathsf{null}$ and combines any pair of non-null values into error. For each Key, therefore, Val is bound to the aggregated result Val1 of the *unique* aggregator whose rules contribute results to that key. If multiple aggregators contribute to the same key, the value is error.[9]

A Dyna program may consist of multiple *dynabases* [4]. Each dynabase defines its own key-value mapping, using aggregation over only the rules in that dynabase, which may refer to the values of keys in other dynabases. In this case, instead of defining a single constraint is as an **R**-expr, we define a different named constraint for each dynabase as a different **R**-expr, where each of the **R**-exprs may call any of these named constraints.

[9] A Dyna program is also supposed to give an error value to the key a if the program contains both the rules a = 3 and a = 4, or the rule a = f(X) when both f(0) and f(1) have values. So we also used only above as the aggregation operator for = rules.

5 Conclusions and Ongoing Work

We have shown how to algebraically represent the bag-relational semantics of any program written in a declarative logic-based language like Dyna. A query against a program can be evaluated by joining the query to the program and simplifying the resulting algebraic expression with appropriate term rewriting rules. It is congenial that this approach allows evaluation to flexibly make progress, propagate information through the expression, exploit arithmetic identities, and remove irrelevant subexpressions rather than wasting possibly infinite work on them.

In ongoing work, we are considering methods for practical interpretation and compilation of rewrite systems. Our goal is to construct an evaluator that will both perform static analysis and "learn to run fast" [9] by constructing or discovering reusable strategies for reducing expressions of frequently encountered sorts. In the case of cyclic rewrite systems, the system should be able to guess portions of a solution and then verify that the guesses are consistent with the rewrite rules. A forward chaining mechanism can be used to invalidate or correct inconsistent guesses, as is common in Datalog, and this mechanism can also be used for change propagation when the program or the query is externally updated [5].

Acknowledgements. This material is based upon work supported by the National Science Foundation under Grant No. 1629564. We thank Scott Smith for helpful comments, and the WRLA program chairs Santiago Escobar and Narciso Martí Oliet for allowing generous time to revise the paper.

References

1. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). In: IEEE Transactions on Knowledge and Data Engineering (1989)
2. Clocksin, W.F., Mellish, C.S.: Programming in Prolog. Springer-Verlag (1984)
3. Colmerauer, A., Roussel, P.: The Birth of Prolog, chap. 7. Association for Computing Machinery (1996)
4. Eisner, J., Filardo, N.W.: Dyna: Extending Datalog for modern AI. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog Reloaded. Lecture Notes in Computer Science, Springer (2011)
5. Filardo, N.W., Eisner, J.: A flexible solver for finite arithmetic circuits. In: Technical Communications of the International Conference on Logic Programming. Leibniz International Proceedings in Informatics (LIPIcs) (2012)
6. Frühwirth, T.: Theory and practice of constraint handling rules. The Journal of Logic Programming **37**(1) (1998)
7. Gallaire, H., Minker, J., Nicolas, J.M.: Logic and databases: A deductive approach. ACM Comput. Surv. **16**(2) (1984)
8. Green, T.J.: Bag semantics. In: LIU, L., ÖZSU, M.T. (eds.) Encyclopedia of Database Systems. Springer (2009)
9. Vieira, T., Francis-Landau, M., Filardo, N.W., Khorasani, F., Eisner, J.: Dyna: Toward a self-optimizing declarative language for machine learning applications. In: Proceedings of the ACM SIGPLAN Workshop on Machine Learning and Programming Languages (MAPL). ACM (2017)

Well-Founded Induction via Term Refinement in CafeOBJ

Kokichi Futatsugi

Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
futatsugi@jaist.ac.jp

Abstract. Well-founded induction is formalized as a method for constructing proof scores in the CafeOBJ specification verification language system, and the power of the method is demonstrated with several instructive examples.

In CafeOBJ, a property of interest on a specification (i.e. a goal property) is described as a Boolean ground term $\mathtt{prp}(\mathtt{c}_1, \mathtt{c}_2, \cdots, \mathtt{c}_n)$ with fresh constants $\mathtt{c}_1, \mathtt{c}_2, \cdots, \mathtt{c}_n$ that correspond to the parameters of the goal property. A specification is formalized as a set of equations[1] each of which can be interpreted as a left to right reduction rule. The goal property proves to hold if the ground term $\mathtt{prp}(\mathtt{c}_1, \mathtt{c}_2, \cdots, \mathtt{c}_n)$ is reduced to `true` with the reduction rules.

Usually, term refinements of and inductions on $\mathtt{c}_1, \mathtt{c}_2, \cdots, \mathtt{c}_n$ are necessary to show that $\mathtt{prp}(\mathtt{c}_1, \mathtt{c}_2, \cdots, \mathtt{c}_n)$ is reduced to `true`. The well-founded induction in CafeOBJ naturally combines the term refinements of and the inductions on $\mathtt{c}_1, \mathtt{c}_2, \cdots, \mathtt{c}_n$, and can support a variety of induction schemes in a uniform and transparent way.

1 Introduction

CafeOBJ [15,9,3] is an executable algebraic specification language system that can be used as a specification verification system by writing proof scores for verifying that a specification (or any constructor model of the specification) satisfies properties of interest. A proof score in CafeOBJ is a piece of code that constructs proof modules and does reductions on them with the equations in the specification and the proof modules [14,13] (See **2.1** for more details).

The main goal of specification verification is to increase the quality of a specification high enough to construct systems on it. Fully automatic verifications often fail to convey important structures of the systems. One should seek to make balanced optimal use of the respective abilities of humans and computers, so that computers do the tedious formal calculations, and humans do the high level planning. Proof scores intend to achieve these goals [10,11].

Proof scores can describe many kinds of induction schemes including structural induction [2,8], and quite a few cases are developed until now [10,11,13].

[1] CafeOBJ also allows transition/rewriting rules as axioms for specifying state transition systems. The method of this paper can apply to specifications with transition rules by making use of CafeOBJ's *built-in search predicates*.

Well-founded induction (WFI) proposed in this paper can subsume various induction schemes in a uniform and transparent way, and enhances the power of proof scores.

Section **2** explains fundamental concepts and techniques with simple examples. Section **3** describes several fundamentals that nicely coordinate to provide WFI via term refinement. Section **4** shows three important WFI schemes with examples. Section **5** describes related works, and section **6** concludes the paper.

[**Contributions of This Paper**] (1) WFI is formalized as a concrete method for constructing proof scores. (2) Fundamentals of the method are formalized. (3) Instructive proof scores with WFI are presented for three important induction schemes.

2 Explanatory Examples

Fundamental concepts and techniques necessary for constructing proof scores in CafeOBJ are presented, and simple proof score examples without and with WFI are explained. We intentionally use simple instructive examples for explaing our method in this section and section **4**, because it is the best way to show a method works. That is, examples are not subsidiary but essential.

2.1 Proof Scores in CafeOBJ

A CafeOBJ's specification (i.e. module) M is a set of equations and denotes **constructor models** each of which satisfies the equations. A sort that is a co-arity of some constructor operator is called **constrained**. For example, the sort `Pnat` declared at **2.2**`/05:` is constrained because it is the co-arity of the constructors `0` and `s_` (**2.2**`/06:,07:`). A sort that is a supersort of a constrained sort is also a constrained sort. A sort that is not constrained is called **loose**. A constructor model is the model where the carrier sets for the constrained sorts consist of interpretations of terms formed with constructors and elements of loose sorts (see 3.2.4. of [14] for a formal definition). Considering only the constructor models justifies inductive inference based on the inductive structures of the constrained sorts.

Specification verification is to show that any constructor model of a specification (module) M satisfies some desirable properties. In CafeOBJ, a property of interest on a module is described as a Boolean ground term p with fresh constants. If the term p is inferred to be equal to `true` with the M's set of equations (in symbols $M \vdash_{\mathrm{e}} p$), any constructor model of M satisfies p (in symbols $M \models p$) by the soundness of equational inference (in symbols $M \vdash_{\mathrm{e}} p\,\texttt{implies}\,M \models p$). If the term p is reduced to `true` by using the M's set of equations as reduction (rewriting) rules from left to right (in symbols $M \vdash_{\mathrm{r}} p$), $M \vdash_{\mathrm{e}} p$ holds by the soundness of reduction with respect to equational inference (in symbols $M \vdash_{\mathrm{r}} p\,\texttt{implies}\,M \vdash_{\mathrm{e}} p$). Because implication is transitive the following implication or proof rule is obtained.

(1) $\quad M \vdash_{\mathrm{r}} p\,\texttt{implies}\,M \models p$

Usually $M \vdash_{\mathrm{r}} p$ is difficult to prove directly, and we need to find modules $M_1, M_2, \cdots, M_n$ that satisfy the following implication or proof rule.

(2) $\quad (M_1 \models p$ `and` $M_2 \models p$ `and` $\cdots$ `and` $M_n \models p)$ `implies` $M \models p$

$M_i \vdash_{\mathrm{r}} p$ would be still difficult to prove and **(2)** is applied repeatedly. The repeated applications of **(2)** generates proof trees successively whose root is $M \models p$. A leaf node $M_l \models p$ of a proof tree is called effective if $M_l \vdash_{\mathrm{r}} p$ holds. A proof tree is called effective if all of whose leaf nodes are effective. The proof of $M \models p$ is achieved if an effective proof tree whose root is $M \models p$ is constructed. A **proof score** in CafeOBJ is a piece of code for constructing an effective proof tree.[2]

2.2 Specification of _+_ on Peano Natural Numbers and its Associativity

The following CafeOBJ module `PNAT` (`04:-09:`) specifies Peano nutural numbers, and module `PNAT+` (`13:-18:`) specifies a binary plus operator `_+_` (`15:`) on the natural numbers.[3] Line number like `09:` is for explanation and is not part of CafeOBJ code, and CafeOBJ code starts from the 6th column in each line. In the CafeOBJ code, "`--` " or "`-->` " indicates that the following text until the end of the line is a comment (`01:-03:`, `10:-12:`). Two constructor operators `0` and `s_` are declared as `06:-07:` and a constrained sort `Pnat` (`05:`) is defined to be the set of terms of the form $\overbrace{\texttt{s s} \cdots \texttt{s}}^{n}$ `0` $(n \in \{0, 1, 2, \cdots\})$. That is,

$$\texttt{Pnat} \stackrel{\text{def}}{=} \{\texttt{0}, \texttt{s 0}, \texttt{s s 0}, \texttt{s s s 0}, \cdots\} .$$

`08:` declares the equality predicate on sort `Pnat`, `15:` declares the rank (i.e. arity and co-arity) of the binary plus operator `_+_`, and `09:`, `16:-18:` define their function (or meaning). `M:Pnat` in `09:` is an in-line variable declaration, and valid until the end of the equation declaration (i.e. the end of `09:`). "`vars X Y : Pnat .`" in `16:` is an off-line variable declaration, and valid until the end of module declaration (i.e. the end of `18:`).

```
01:  --> ---------------------------------------------------------
02:  --> PNAT: Peano NATural numbers
03:  --> ---------------------------------------------------------
04:  mod! PNAT {
05:   [Pnat]
06:  op 0 : -> Pnat {constr} .
07:  op s_ : Pnat -> Pnat {constr} .
08:  pred _=_ : Pnat Pnat {comm} .
09:  eq (M:Pnat = M) = true . }
10:  --> ---------------------------------------------------------
11:  --> PNAT with addition _+_
12:  --> ---------------------------------------------------------
13:  mod! PNAT+ {
14:  pr(PNAT)
```

[2] Definitions of proof scores develop in accordance with the developments of proof scores, and the definition in this paper is hopefully more articulate and transparent than ones in [14,12].

[3] All the CafeOBJ specifications and proof scores in this paper are posted at `https://cafeobj.org/~futatsugi/misc/wrla2020/` and readers can execute them.

```
15:  op _+_ : Pnat Pnat -> Pnat .
16:  vars X Y : Pnat .
17:  eq 0 + Y = Y .
18:  eq s X + Y = s(X + Y) . }
```

The plus operator `_+_` satisfies the following associativity property.
$\forall x, y, z \in$ `Pnat` `((`x `+` y`)` `+` z `=` x `+ (`y `+` z`))`
The associativity property of `_+_` is described as follows in CafeOBJ.

```
19:  ops x y z : -> Pnat . -- fresh constants
20:  eq[assoc+]: (x + y) + z = x + (y + z) .
```

Where `x,y,z` (`19:`) are fresh constants, i.e. they did not exist in the current context before, and each of which stands for any element of `Pnat`.

Let `assoc+` stand for "`(x + y) + z = x + (y + z)`". The following sections **2.3** and **2.4** show two proof scores, one with ordinary induction and one with WFI, for proving that any constructor model of module `PNAT+` satisfies `assoc+`, that is `PNAT+` $\models$ `assoc+` holds.

2.3 Proof Score with Ordinary Induction

If a CafeOBJ command "`red in PNAT+ : assoc+ .`"[4]returns `true`, then `PNAT+` $\vdash_{\mathrm{r}}$ `assoc+` proves to hold. Then because of **2.1-(1)**, `PNAT+` $\models$ `assoc+` holds and the proof is done. But it is not the case, and we need to apply **2.1-(2)** by making use of the induction on `x`.

The CafeOBJ command "`open PNAT+ .`" (`09:`) opens module `PNAT+` and create a tentative module, and `16:` closes the module. Let this tentative module be `PNAT+ib`. In `PNAT+ib`, all sorts, operators, and equation of `PNAT+` are available, `13:` specifies induction base case, and reduction command `15:` checks whether `PNAT+ib` $\vdash_{\mathrm{r}}$ `assoc+` holds.

```
01:  --> ========================================================
02:  --> Proof score for proving:
03:  -->  ops x y z : -> Pnat .
04:  -->  eq[assoc+]: (x + y) + z = x + (y + z) .
05:  --> at the module PNAT+ by constructor induction on x
06:  --> ========================================================
07:  --> --------------------------------------------------------
08:  --> induction base on x
09:  open PNAT+ .
10:  -- fresh constants
11:  ops x y z : -> Pnat .
12:  -- specify the induction base case
13:  eq x = 0 .
14:  -- check the induction base
15:  red (x + y) + z = x + (y + z) .
16:  close
17:  --> --------------------------------------------------------
18:  --> induction step on x
19:  open PNAT+ .
20:  -- fresh constants
21:  ops x x$1 y z : -> Pnat .
22:  -- specify the induction step case
23:  eq x = s x$1 .
24:  -- declare induction hypothesis
25:  eq (x$1 + Y:Pnat) + Z:Pnat = x$1 + (Y + Z) .
26:  -- check the induction step
27:  red (x + y) + z = x + (y + z) .
28:  close
29:  --> ========================================================
30:  --> QED [assoc+]
31:  --> ========================================================
```

[4] Same as "`select PNAT+ .`" and "`red assoc+ .`".

Let the tentative module opened by `19:` and closed by `28:` be `PNAT+is`. In `PNAT+is`, `23:` specifies induction step case, `25:` declared induction hypothesis, and reduction command `27:` checks whether `PNAT+is` $\vdash_{\mathrm{r}}$ `assoc+` holds.

The induction on `x` guarantees that the proof rule
(`PNAT+ib` $\models$ `assoc+ and PNAT+is` $\models$ `assoc+`) `implies PNAT+` $\models$ `assoc+`
holds. Inputting the two tentative modules (`09:-28:`) into CafeOBJ shows that two reductions `15:` and `27:` return `true`, and `PNAT+ib` $\vdash_{\mathrm{r}}$ `assoc+` and `PNAT+is` $\vdash_{\mathrm{r}}$ `assoc+` prove to hold. Hence, the proof score `09:-28:` proves to describe two effective leaf nodes of an effective proof tree that has `PNAT+` $\models$ `assoc+` as the root, and `PNAT+ib` $\models$ `assoc+`, `PNAT+is` $\models$ `assoc+` as two leaf nodes.

The proof score `09:-28:` is written in classical `open-close` style [22,10], and describes an effective proof tree by showing every contents of all the effective leaf nodes directly.

2.4 Proof Score with WFI

The principle of WFI we use is formulated as follows.

Proposition 1. [WFI] Let T be a set and $wf>$ be a well-founded binary relation on T, i.e. $wf> \subseteq T \times T$ and there is no infinite sequence of $t_i \in T$ with $(t_i, t_{i+1}) \in wf>$ $(i \in \{1, 2, \cdots\})$. Let p be a predicate on T (i.e. a function from T to truth values $\{\mathtt{true}, \mathtt{false}\}$) then the following implication or proof rule holds where $t\ wf> t'$ means $(t, t') \in wf>$.

$$\frac{\forall t \in T([\forall t' \in T((t\ wf> t')\ \mathtt{implies}\ p(t'))]\mathtt{implies}\ p(t))}{\forall t \in T(p(t))}\ \mathtt{implies} \qquad (\mathbf{1})$$

That is, if $p(t)$ whenever $p(t')$ for all $t' \in T$ such that $t\ wf> t'$, then $p(t)$ for all $t \in T$ (See **3.1** for proof). □

The following `01:-36:` is a proof score for proving `PNAT+`$\models$`assoc+` with WFI.

`12:` imports module `PNAT+` with `pr` (protecting) mode. `15:` declares fresh constants that are used in `31:` to define a proof goal and in `24:` to define an induction hypothesis. `18:` defines a sort `Ppp` of three tuples of `Pnat`, and `Ppp` consists of all the terms of the form `t(`x`,`y`,`z`)` such that $x, y, z \in$ `Pnat`. `20:` declares variables that are used in `22:` and `24:`. `22:` defines a well-founded binary relation `_wf>_` on `Ppp`, and `24:` defines the induction hypothesis.

It is easy to prove that `_wf>_` defined at `22:` is well-founded on `Ppp`, because the first argument of `t(`x`,`y`,`z`)` decreases to a strict subterm from left to right across `_wf>_` (see **3.6**).

Let `Ppp`, `_wf>_`, and `assoc+` be considered as T, $wf>$, and p of **Proposition 1**. Note that `assoc+` with three fresh constants can be considered as a predicate on `Ppp`. It is interesting to see that `24:` declares the premise $[\forall t' \in T((t\ wf> t')$ `implies` $p(t'))]$ of **Proposition 1-(1)** as a conditional equation succinctly.

Because each of `x,y,z` of module `PNAT+assoc-wfi` (`11:-26:`) is a fresh constant and denotes arbitrary element of `Pnat`, `PNAT+assoc-wfi`$\models$`assoc+` corresponds to the premise and `PNAT+`$\models$`assoc+` corresponds to the conclusion of the

outermost `implies` of **Proposition 1-(1)**. Hence **Proposition 1-(1)** asserts (`PNAT+assoc-wfi`⊨`assoc+ implies PNAT+`⊨`assoc+`) .

```
01:  --> ========================================================
02:  --> Proof score for proving:
03:  -->  ops x y z : -> Pnat .
04:  -->  eq[assoc+]: (x + y) + z = x + (y + z) .
05:  --> at the module PNAT+ by WFI on 3 tuple t(x,y,z).
06:  --> ========================================================
07:  --> --------------------------------------------------------
08:  --> module for proof by WFI
09:  --> on 3 tuple (x,y,z)
10:  --> --------------------------------------------------------
11:  mod PNAT+assoc-wfi {
12:  pr(PNAT+) -- base module on which the proof applies
13:  -- fresh constants for defining the proof goal
14:  -- :goal{eq (x + y) + z = x + (y + z) .} (31:)
15:  ops x y z : -> Pnat .
16:  -- declaring a sort Ppp of 3 tuples of X:Pnat,Y:Pnat,Z:Pnat
17:  -- and its constructor t
18:  [Ppp] op t : Pnat Pnat Pnat -> Ppp {constr} .
19:  -- declaration of variables
20:  vars X Y Z : Pnat .
21:  -- a well-founded binary relation on Ppp
22:  pred _wf>_ : Ppp Ppp . eq t(s X,Y,Z) wf> t(X,Y,Z) = true .
23:  -- induction hypothesis of WFI
24:  cq (X + Y) + Z = X + (Y + Z) if t(x,y,z) wf> t(X,Y,Z) .
25:  -- fresh constant for refining x
26:  op x$1 : -> Pnat . }
27:  --> --------------------------------------------------------
28:  --> executing proof by WFI on Ppp with PTcalc
29:  --> --------------------------------------------------------
30:  select PNAT+assoc-wfi .
31:  :goal{eq (x + y) + z = x + (y + z) .}
32:  :def x = :csp{eq x = 0 . eq x = s x$1 .}
33:  :apply(x rd-)
34:  --> ========================================================
35:  --> QED [assoc+]
36:  --> ========================================================
```

`30:-33:` are CafeOBJ commands for constructing proof trees and checking their effectiveness with **proof tree calculus** (PTcalc for short, see **3.5**).

`30:` selects module `PNAT+assoc-wfi` as the current context module. `31:` creates a proof tree only with a root node, and sets the root node as the current node. The root node created is a CafeOBJ module that imports the current context module, and has the equation "`eq (x + y) + z = x + (y + z) .`" as the goal proposition. This equation is for denoting the goal proposition, and is not executed as a reduction rule. Let the root node be called `PNAT+assoc-wfi+pg`. Clearly (`PNAT+assoc-wfi-pg`⊨`assoc+ implies PNAT+assoc-wfi`⊨`assoc+`).

`32:` defines `x` as the command name of the command `:csp{eq x = 0 . eq x = s x$1 .}`. Note that the name `x` is overloaded to denote the command `x` and the constant `x`. `:csp` command defines a **case sprit** with two equations "`eq x = 0 .`" and "`eq x = s x$1 .`" (see **3.4**). `:def` command only defines name of command, and application of the defined command is done by using `:apply` command as `33:`.

`33:` applies `:apply(x rd-)` to the current node (i.e. the root node) and first applies `:apply(x)` to the root node and then applies `:apply(rd-)` to all the child nodes created by the application of `:apply(x)` (See **3.5** for more details).

`x` is the case split command `:csp{eq x = 0 . eq x = s x$1 .}` with two equations that give an exhaustive case split because `0` and `s_` are two constructors of `Pnat`. Hence `:apply(x)` creates two child nodes `PNAT+assoc-wfi+pg-0` and

PNAT+assoc-wfi+pg-s by adding each of the two equations to the root node PNAT+assoc-wfi+pg respectively (see **3.3-(1)** and **3.4**). Hence we get
(PNAT+assoc-wfi+pg-0⊨assoc+ and PNAT+assoc-wfi+pg-s⊨assoc+)
implies PNAT+assoc-wfi+pg⊨assoc+ .

Applying :apply(rd-) to the two leaf nodes shows their effectiveness by reducing the proof goal with the equations in each node (module), and we get PNAT+assoc-wfi+pg-0$\vdash_{\mathrm{r}}$ assoc+ and PNAT+assoc-wfi+pg-s$\vdash_{\mathrm{r}}$ assoc+. Hence the proof score 11:-33: proves to construct an effective proof tree. Note that the effectiveness of a leaf node is checked only by applying :apply(rd-) to the leaf node.

3 Fundamentals of WFI via Term Refinement

This section shows fundamentals that nicely coordinate to provide the WFI via term refinement in CafeOBJ. The fundamentals are fairly simple and almost obvious, we hope, and it should be nice for users of the method proposed.

3.1 WFI

We present the proof of **Proposition 1** as follows to show that the logic for the WFI we use is simple and transparent. Assume there exits $t \in T$ such that not$(p(t))$, then there should be $t' \in T$ such that $t\ wf{>}\ t'$ and not$(p(t'))$. Repeating this produces an infinite $wf{>}$-descending sequence. It conflicts with well-foundedness of $wf{>}$.

The principle of WFI is well established (e.g. 14.1.5 of [18], A.1.6 & A.1.7 of [28]). It is worthwhile to notice that it works for any well-founded *binary relations*, for Rod Burstall refers, in his famous seminal paper on structural induction [2], "generalised principle of induction (Noetherian induction)" of [5] that is based on *partial order relations*.

The relation _wf>_ defined by **2.4**/22: is not a partial order relation, and we make use of **Proposition 1** on general binary relations.

3.2 Theorem of Constants

In CafeOBJ, a property of interest on a module M is described as a Boolean term composed of operators available at the module M. The Boolean term is expressed as prp$(v_1,\cdots,v_n)$, has $n \in \{0,1,2,\cdots\}$ parameters (or variables) v_1:Sort$_1$,$\cdots$,v_n:Sort$_n$ of specific sorts, and the property of interest is
$\forall v_1$:Sort$_1$,$\cdots$,$\forall v_n$:Sort$_n$ (prp$(v_1,\cdots,v_n)$)
(i.e. the parameters are universally quantified). Let c$_1$:->Sort$_1$,$\cdots$,c$_n$:->Sort$_n$ be n fresh constants which correspond to the n parameters v_1:Sort$_1$,$\cdots$,v_n:Sort$_n$.

Theorm of constants (Theorem 3.3.11 of [17]) states the definitional equivalence of the variables v_1:Sort$_1$,$\cdots$,v_n:Sort$_n$ and the fresh constants c$_1$:->Sort$_1$, $\cdots$, c$_n$:->Sort$_n$ in defining algebraic models, and can be stated as follows.

Proposition 2. [Theorem of Constants] Let prp be a predicate on a module M, then ($M \models (\forall v_1$:Sort$_1$,$\cdots$,$\forall v_n$:Sort$_n$(prp$(v_1,\cdots,v_n)$))) if and only if ($M \cup$ {c$_1$:->Sort$_1$,$\cdots$,c$_n$:->Sort$_n$} $\models$ prp$(c_1,\cdots,c_n)$). □

Based on **Proposition 2** a property of interest on a module can be described as a Boolean ground term with fresh constants in proof scores.

3.3 Case Split with Equations

Let M be a module, p be a property on M, and $e_1, \cdots, e_n$ be equations such that at least one of them holds for any constructor model of M. That is, the equations cover all the possibilities (i.e. are exhaustive). Let M_{+e_i} be a module gotten by adding e_i to M, then each constructor model of M is a model of M_{+e_j} for some $j \in \{1, 2, \cdots, n\}$, and we get the following proof rule that states **the principle of case split with equations**.

(**1**) $\quad (M_{+e_1} \models p$ `and` $M_{+e_2} \models p$ `and` $\cdots$ `and` $M_{+e_n} \models p)$ `implies` $M \models p$

3.4 Term Refinement with Equations

A fresh constant is declared to be of some constrained sort *Srt*, and the constrained sort has the inductive structure of constructors. This makes it possible to refine a fresh constant of sort *Srt* with equations that cover all possibilities (i.e. are exhaustive) in constructing a term of sort *Srt*. For example, **2.2**`/05:-07:` specifies constrained sort `Pnat` and its constructors `0`, `s_` as follows.

```
[Pnat]
op 0 : -> Pnat {constr} .
op s_ : Pnat -> Pnat {constr} .
```

Based on the above specification, a fresh constant "`op n : -> Pnat .`" can be refined with two equations "`eq n = 0 .`" and "`eq n = s n$1 .`", where `n$1` is another fresh constant of sort `Pnat` (i.e. "`op n$1 : -> Pnat .`"). The case split in **2.4**`/32:` is justified by this term refinement of `n`.

`n$1` can be refined with similar two equations in turn, and `n` can be refined into a sufficiently detailed term "`s s ... s n$m`" of sort `Pnat`.

Term refinement with equations is an important instance of case split with equations (**3.3**).

3.5 Proof Tree Calculus (PTcalc)

A piece of CafeOBJ code like **2.4**`/30:-33:` is a proof score for executing proofs with proof tree calculus (a PTcalc proof score for short). PTcalc is a subsystem of CafeOBJ for constructing proof trees and checking their effectiveness, was first implemented based on [16], and have developed to the current form by incorporating `:def`, `:init` (see below) and extending `:apply`. PTcalc is considered to be an implementation of a simplified version of [14]'s "6.4. specification calculus for proof scores" and "6.5. Proof trees and proof scores" via case split and term refinement with equations.

The commands of PTcalc start with `:` like `:goal`,`:def`,`:csp`,`:apply` that are used in **2.4**`/30:-33:`, and the functions of these commands are explained in **2.4**. Another PTcalc command `:init` that is used in **4.1**`/54:-55:` is explained there. A PTcalc proof score is typically doing the following two successively. (i) Gives names to `:csp` and `:init` commands (e.g. **2.4**`/32:`, **4.1**`/54:-57:`) (ii) Applies the named commands and `rd-` to a leaf node by executing `:apply` command with a sequence of the names and `rd-` as the argument (e.g. **2.4**`/33:`, **4.1**`/58:`).

Let c be a command name or `rd-` and γ be a sequence of command names and `rd-`. Applying `:apply(`$c\,\gamma$`)` to a leaf node ln does the followings successively.[5] (1) Applies `:apply(`c`)` to ln; (1-1) if c is a `:csp` command name, $i \in \{1, 2, ..\}$ child node(s) are created and they are new leaf nodes instead of ln; (1-2) if c is a `:init` command name or `rd-`, no child node is created and ln is still leaf node. (2) Applies `:apply(`γ`)` to each of the leaf nodes that has not proved to be effective by the (c = `rd-`) case of (1-2). Note that (2) calls (1) recursively until γ becomes nil sequence, and `:apply(`$c\,\gamma$`)` can generate & check exponentially growing proof trees.[6]

3.6 Correctness of PTclac Proof Scores

The piece of code **2.4**/30:-33: is executing the proof with PTcalc based on the induction hypothesis **2.4**/24: that uses the well-founded relation `_wf>_` (**2.4**/22:). The module `PNAT+assoc-wfi` (**2.4**/11:-26:) is for declaring the induction hypothesis and necessary fresh constants. Hence, a piece of code like **2.4**/11:-33: is called **a PTcalc Proof Score**. Assuming that the PTcalc subsystem is working properly, the correctness of a PTcalc Proof Score is implied if **(i)** the binary relation `_wf>_` is well-founded, and **(ii)** each case sprit `:csp{...}` is exhaustive. Note that a correct PTcalc proof score should be checked its effectiveness by applying `rd-` to all the leaf nodes.

With respect to the PTcalc proof score **2.4**/11:-33:, the validities of **(i)** and **(ii)** was proved and the effectiveness was also proved by applying `rd-` to all the leaf nodes. Readers are recommended to execute **2.4**/11:-33: and check its effectiveness by themselves. The PTcalc proof scores shown in **4.1**, **4.2**, **4.3** prove to be correct and effective with the same reasoning.

It can be said that proving "**(ii)** each case sprit `:csp{...}` is exhaustive" is still easy even for complex examples. However, finding an effective well-founded relation `_wf>_` is most crucial, and it is sometimes not simple to prove the relation is well-founded. It is worthwhile to notice that a relation `_wf>_` defines a rewriting relation from left to right on the terms (of the form `t(...)` at **2.4**/18: and **4.1**/40:, or `s...` of sort `Pnat` in **4.2**), and `_wf>_` is well-founded if the rewriting relation is terminating. The termination property of rewriting relations is well studied and quite a few usable methods are developed for checking termination (e.g. see Chap 6 of [28]). We can make use of those methods or even tools for proving `_wf>_` is well-founded.

4 Three Other Induction Schemes with WFI

This section presents three instructive proof scores that codify three other important induction schemes in a uniform and transparent way.

4.1 WFI with Well-Founded Relation Depending on Many Parameters

The well-founded relation defined in **2.4**/22: is based only on the first parameter `X`, and **2.3** and **2.4** show that ordinary structural (or constructor) induction on a

[5] The current node is also adjusted, but those details are omitted here.

[6] `:apply(`$c\,\gamma$`)` provides almost all of what we expect from the "generate & check method" of [12].

single parameter can be translated into WFI in which the well-founded relation is based only on the single parameter.

The following proof score describes WFI with well-founded relation depending on many parameters.[7] Similarity of `27:-58:` and **2.4**/`11:-33:` is clear, and a well-founded relation depending on two parameters `M` and `N` is defined in `42:-43:` in a similar way as **2.4**/`22:`.

It is also easy to prove that `_wf>_` defined at `42:-43:` is well-founded on `Pp`, because each of the first and second arguments of `t(`M`,`N`)` decreases to a strict subterm from left to right across `_wf>_`.

Cases depending on more than 2 parameters can be described similarly, even though finding the well-founded relation `_wf>_` for constructing an effective proof tree is one of the most crucial points in WFI (see **3.6**).

```
01:  --> ---------------------------------------------------------
02:  --> PNAT with _>_  and _=_
03:  --> ---------------------------------------------------------
04:  mod! PNAT>= {
05:  pr(PNAT)
06:  -- greater relation on Pnat
07:  op _>_ : Pnat Pnat -> Bool .
08:  eq s N:Pnat > 0 = true .
09:  eq 0 > M:Pnat = false .
10:  eq (s M:Pnat > s N:Pnat) = M > N .
11:  -- equal relation on Pnat
12:  eq (s M:Pnat = s N:Pnat) = (M = N) .
13:  eq ((s M:Pnat) = 0) = false . }
14:  --> =========================================================
15:  --> Proof score for proving
16:  --> ops m n : -> Pnat .
17:  --> eq[oo><=]:
18:  -->   (((m > n) and not(n > m) and not(m = n)) or
19:  -->    (not(m > n) and (n > m) and not(m = n)) or
20:  -->    (not(m > n) and not(n > m) and (m = n))) = true .
21:  --> at module PNAT>=.
22:  --> =========================================================
23:  --> ---------------------------------------------------------
24:  --> module for proof by WFI
25:  --> on 2 tuple (m,n)
26:  --> ---------------------------------------------------------
27:  mod OO><=wfi {
28:  -- base module on which the proof applies
29:  pr(PNAT>=)
30:  -- predicate for expressing the proof goal
31:  pred oo><= : Pnat Pnat .
32:  eq oo><=(M:Pnat,N:Pnat) =
33:     ((M > N) and not(N > M) and not(M = N)) or
34:     (not(M > N) and (N > M) and not(M = N)) or
35:     (not(M > N) and not(N > M) and (M = N)) .
36:  -- fresh constants for defining the proof goal
37:  -- :goal{eq oo><=(m,n) = true .} (53:)
38:  ops m n : -> Pnat .
39:  -- a sort Pp of 2 tuples (pairs) of Pnat and its constructor
40:  [Pp] op t : Pnat Pnat -> Pp {constr} .
41:  -- a well-founded binary relation on Pp
42:  pred _wf>_ : Pp Pp .
43:  eq t(s M:Pnat,s N:Pnat) wf> t(M,N) = true .
44:  -- induction hypothesis of WFI
45:  cq[oo><=ih]: oo><=(M:Pnat,N:Pnat) = true
46:               if t(m,n) wf> t(M,N) .
47:  -- fresh constants for refining m and n
48:  ops m$1 n$1 : -> Pnat . }
49:  --> ---------------------------------------------------------
50:  --> executing proof with TPcalc via term refinements
51:  --> ---------------------------------------------------------
52:  select OO><=wfi .
53:  :goal{eq oo><=(m,n) = true .}
54:  :def ih = :init [oo><=ih]
55:            by {M:Pnat.PNAT <- M:Pnat.PNAT;}
```

[7] The problem of verifying `oo><=` is based on an example in [8].

```
56:  :def m = :csp{eq m = 0 . eq m = s m$1 .}
57:  :def n = :csp{eq n = 0 . eq n = s n$1 .}
58:  :apply(ih m n rd-) -- succeed
59:  --> ----------------------------------------------------------
60:  **> QED [oo><=]
61:  --> ----------------------------------------------------------
```

`54:-55:` gives name `ih` to the `:init` command that initializes the conditional equation `oo><=ih` in `45:-46:` that declares the induction hypothesis. An `:init` command initializes (or instantiates) an already declared equation for making the equation work as a reduction rule by instantiating variables in the equation, and the soundness of applying the initialized equation is clear.

The instantiation of a variable given in `55:` is a trivial one, and the purpose of the `:init` command in `54:-55:` is to make the left hand side `oo><=(M:Pnat,N:Pnat)` of the equation `oo><=ih` reduced form (normal form). Without making left hand side normal form as `54:-55:`, the application `58:` can not succeed in showing effectiveness of the proof tree created. This kind of initialization is necessary also for the proof score in **4.2** (i.e. **4.2**/`47:,48:`).

`56:` and `57:` give names `m` and `n` to two `:csp` commands that define two exhaustive case splits by refining the fresh constants `m` and `n` with the fresh constants `m$1` and `n$1` declared in `48:`.

`58:` applies `:apply(ih m n rd-)` to the root node (a leaf node), and does the followings successively (See **3.5**). (1) Applies `:apply(ih)` and adds the initialized equation to the root node. (2) Applies `:apply(m)` to the root node and creates 2 child nodes (i.e. leaf nodes) by adding each of the 2 equations in `56:` respectively. (3) Applies `:apply(n)` to each of the 2 leaf nodes and creates two child nodes for each of the 2 leaf nodes by adding each of the two equations in `57:` respectively, and creates 4 leaf nodes in total. (4) Applies `:apply(rd-)` to each of the 4 leaf nodes and proves that they are effective. Hence the proof score `27:-58:` proves to construct an effective proof tree.[8]

The above inference can be described using notations $Mod \models prp$ and $Mod \vdash_{\mathrm{r}} prp$ as in **2.4**. You can write `:apply(ih rd- m rd- n rd-)` instead of `:apply(ih m n rd-)` (`58:`) if you want to do an effectiveness check as soon as a new equation is added.

4.2 Simultaneous WFI

The goal property is sometimes described as a conjunction of component propositions (i.e. ground Boolean terms) $p_1, p_2, \cdots, p_n$ and, if possible, each component is better to be proved independently. However, if the proof is with induction, p_i's proof may require the induction hypothesis of $p_j (i \neq j)$. In that case, we assume all induction hypotheses of $p_k (k \in \{1, 2, \cdots, n\})$ and prove p_i for each $i \in \{1, 2, \cdots, n\}$. This kind of induction scheme is called *simultaneous induction*.

The following proof score `25:-50:` codifies the proof with simultaneous WFI.

```
01:  --> ----------------------------------------------------------
02:  --> PNAT with even and odd predicates
```

[8] The behavior of the PTcalc subsystem when executing `:apply(ih m n rd-)` (`58:`) is best seen by looking into the comment `#|...|#` in the file `pnat-gt-eq-oo-wfi-ps.cafe` posted at the web page indicated in the footnote 3.

```
03:  --> ----------------------------------------------------------
04:  mod! PNATeo {
05:  pr(PNAT)
06:  -- even,odd predicates on Pnat
07:  preds even odd : Pnat .
08:  eq even(0) = true .
09:  eq even(s 0) = false .
10:  eq even(s s N:Pnat) = even(N) .
11:  eq odd(0) = false .
12:  eq odd(s 0) = true .
13:  eq odd(s s N:Pnat) = odd(N) . }
14:  --> ==========================================================
15:  --> Proof score for proving
16:  -->   op n : -> Pnat . -- fresh constant
17:  -->   eq[e=o=e]: ((even(n) iff odd(s n)) and
18:  -->               (odd(n) iff even(s n))) = true .
19:  --> at module PNATeo
20:  --> by simultaneous WFI (SWFI) on n.
21:  --> ==========================================================
22:  --> ----------------------------------------------------------
23:  --> module for proof by SWFI on Pnat
24:  --> ----------------------------------------------------------
25:  mod PNATeo-swfi {
26:  -- base module on which the proof applies
27:  pr(PNATeo)
28:  -- 2 goal predicates for expressing the goal proposition
29:  preds e=o o=e : Pnat .
30:  eq e=o(N:Pnat) = (even(N) iff odd(s N)) .
31:  eq o=e(N:Pnat) = (odd(N) iff even(s N)) .
32:  -- fresh constants for defining the proof goal
33:  -- :goal{eq e=o(n) = true . eq o=e(n) = true .} (46:)
34:  op n : -> Pnat .
35:  -- a well-founded binary relation on Pnat .
36:  pred _wf>_ : Pnat Pnat . eq s N:Pnat wf> N = true .
37:  -- two induction hypotheses of SWFI
38:  cq[e=o-ih]: e=o(N:Pnat) = true if n wf> N .
39:  cq[o=e-ih]: o=e(N:Pnat) = true if n wf> N .
40:  -- fresh constant for refining n
41:  op n$1 : -> Pnat . }
42:  --> ----------------------------------------------------------
43:  --> executing proof by SWFI via term refinement with PTcalc
44:  --> ----------------------------------------------------------
45:  select PNATeo-swfi .
46:  :goal{eq e=o(n) = true . eq o=e(n) = true .}
47:  :def eo = :init [e=o-ih] by {N:Pnat <- N:Pnat;}
48:  :def oe = :init [o=e-ih] by {N:Pnat <- N:Pnat;}
49:  :def n = :csp{eq n = 0 . eq n = s n$1 .}
50:  :apply(eo oe n rd-) -- succeed
51:  --> ==========================================================
52:  **> QED [e=o=e]
53:  --> ==========================================================
```

The goal property is a conjunction of `e=o(n)` (`30:`) and `o=e(n)` (`31:`), and their induction hypotheses `e=o-ih` and `o=e-ih` are declared in `38:-39:`. `:goal` command can declare many equations simultaneously as `46:`. By making the proof goal contain two equations as `46:`, each of `e=o(n)` and `o=e(n)` is reduced independently, and a leaf node is effective if the both of `e=o(n)` and `o=e(n)` are reduced to `true`. To make the proof tree constructed effective, the following two are needed. (1) Normalization of left hand sides of `[e=o-ih]` and `[o=e-ih]` (`47:-48:`). (2) Case split by refining `n` (`49:`).

4.3 WFI with Associative Constructor

The principle of structural induction was stated as follows in [2]. *If for some set of structures a structure has a certain property whenever all its proper constituents have that property then all the structures in the set have the property.* If constructors for the structures are associative, multiple constituent relations are

produced for an equal structure. For example, if f is a binary associative constructor, $f(a, f(b, c))$ and $f(f(a, b), c)$ are equal but with different constituent relations. Formalization of an induction step in the structural induction should take care of the multiple constituent relations.

The proposed WFI formalizes an induction step with the following **(i)** and **(ii)**, and does not need to consider the multiple constituent relations directly. **(i)** Conditional equation(s) conditioned by a well-founded binary relation `_wf>_`. **(ii)** Term refinements via a set of case sprits each element `:csp{...}` of that is exhaustive.

The proof score in the file `seq-revrev-wfi-ps.cafe` posted at the web page indicated in the footnote 3 codifies WFI with the associative constructor that defines sequences. Similar proof scores are possible for the associative and commutative constructor that define multi-sets.

5 Related Works

5.1 Theorm Provers and Proof Scores

Maude [21] is a sister language of CafeOBJ and both languages share many important features. Maude mainly focuses on sophisticated model checking with its powerful associative and/or commutative rewriting engine, and also provides the Maude Inductive Theorem Prover (MITP) [4] for functional modules, but MITP does not take the proof score approach.

The most notable theorem proving systems supporting induction are either first-order or higher-order. First-order such systems include Otter [23], ACL2 [1]. Higher-order such systems include Coq [6], Isabelle/HOL [19], PVS [25]. These lists are no way comprehensive because there are quite a few theorem proving systems developed or being developed.

The CafeOBJ's proof score method/system has the following characteristics compared to other theorem proving methods/systems including the above ones.

(1) The CafeOBJ language is a general purpose formal specification language for a wide variety of systems, based on Algebraic Abstract Data Types (ADT) that can model and describe specifications in an appropriate abstraction level. This characteristic is unique among other theorem proving languages that are based on other formalism than ADT.
(2) CafeOBJ is an executable formal specification language, and the execution is done via quite transparent mechanism of reduction/rewriting. A proof score is just an important sophisticated example of the executable CafeOBJ code. This is a novel and important feature of proof scores.
(3) In proof score approach, the proof rules of the form **2.1-(2)** or **3.3-(1)**, are formalized at the level of satisfaction assertions $SPEC \models prop$ (or $MOD \models prop$). This is another novel and important feature of proof scores.
(4) Automated parts of proofs with proof scores (i.e. $M \vdash_{\mathrm{r}} p$) is done solely by reduction, and interactive parts are formalized as PTcalc based on proof rules **2.1-(1)** and **2.1-(2)** (or **3.3-(1)**). This two layered structure can provide a simple, transparent, and powerful architecture for interactive verification.

5.2 WFI

Almost all of the theorem proving systems including ones mentioned in **5.1** seem to support WFI. For examples, Coq contains "Coq.Init.Wf" [7] and Isabelle/HOL contains "Isabelle/HOL/Wellfounded" [20].

It is natural to provide, say, lexicographic relations based on strict subterm relations on constructor terms as pre-defined well-founded relations. CafeOBJ's powerful parameterized modules could make it easy to use the pre-defined well-founded relations, even with renaming of sorts and operators. We think, however, it is also important to allow users to define their own well-founded relations succinctly as `_wf>_` in **4.1**`/42:-43:`, **4.3**`/28:-29:`. Because it is a nice kind of liberalness of the proof scores, and provides the possibility of inventing new induction schemes for easier and clearer proofs.

6 Concluding Remarks

CiMPA (CafeInMaude [27] Proof Assistant) and CiMPG (CafeInMaude Proof Generator) reported in [26] provide impressive automatic translation from a class of `open-close` style proof scores (**2.3**) to a class of PTcalc style proof scores (**3.5**)[9]. The motivation behind the automatic translation is to take both advantages of liberalness in `open-close` style and formality in PTcalc style. Induction is a main factor to increase proof score's sophistication, and WFI could ease the coordination CiMPA/CiMPG is aiming of `open-close` style and PTcalc style proof scores.

Proof score capability of CafeOBJ has been significantly enhanced with achievements including **(i)** formalization of induction and case split based on constructor-based specification calculus [14] and **(ii)** specification verification of transition systems with conditional transition rules by making use of built-in search predicates [12,13]. The WFI formaized as a method for constructing proof scores (WFIM) advances **(i)** significantly and has the following characteristics.

- Induction hypotheses are directly declared with conditional equations (i.e. conditional reduction-rules), and a wide variety of induction hypotheses could be defined in an executable way.
- The principle of WFI (**Proposition 1-(1)**) is coded almost directly into the CafeOBJ code like **2.4**`/11:-33:` and **4.1**`/27:-58:`.
- The case split with term refinement (**3.3**, **3.4**) lies in the core of PTcalc (**3.5**) and makes the full use of the WFI hypotheses defined in the conditional equations.

WFIM also has a potential to advances **(ii)** by formalizing sufficient conditions for the leads-to property (a liveness property[24]) of transition systems as constant Boolean grand terms with the built-in search predicates [13]. This would significantly improve specification verification of transition systems.

[9] In [26] "proof score" means only `open-close` style proof score and PTcalc styple proof score is called proof script.

References

1. ACL2: Web page. http://www.cs.utexas.edu/users/moore/acl2/ (2020 accessed)
2. Burstall, R.: Proving properties of programs by structural induction. Computer Journal **12(1)**, 41–48 (1969)
3. CafeOBJ: Web page. https://cafeobj.org/ (2020 accessed)
4. Clavel, M., Palomino, M., Riesco, A.: Introducing the ITP tool: a tutorial. J. UCS **12**(11), 1618–1650 (2006)
5. Cohn, P.M.: Universal Algebra. Harper and Row, New York and London (1965)
6. Coq: Web page. http://coq.inria.fr/ (2020 accessed)
7. Coq.Init.Wf: Web page. https://coq.inria.fr/library/Coq.Init.Wf.html (2020 accessed)
8. Diaconescu, R.: Structural induction in institutions. Inf. Comput. **209**(9), 1197–1222 (2011)
9. Diaconescu, R., Futatsugi, K.: CafeOBJ Report. World Scientific (1998)
10. Futatsugi, K.: Verifying specifications with proof scores in CafeOBJ. In: Proc. 21st IEEE/ACM ASE. pp. 3–10 (2006)
11. Futatsugi, K.: Fostering proof scores in CafeOBJ. In: Proc. 12th ICFEM. LNCS, vol. 6447, pp. 1–20. Springer (2010)
12. Futatsugi, K.: Generate & check method for verifying transition systems in CafeOBJ. In: Software, Services, and Systems. LNCS, vol. 8950, pp. 171–192. Springer (2015)
13. Futatsugi, K.: Introduction to Specification Verification in CafeOBJ (in Japanese). Saiensu-Sha, Tokyo (2017)
14. Futatsugi, K., Gâinâ, D., Ogata, K.: Principles of proof scores in CafeOBJ. Theor. Comput. Sci. **464**, 90–112 (2012)
15. Futatsugi, K., Nakagawa, A.T.: An overview of CAFE specification environment - an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In: Proc. First IEEE ICFEM. pp. 170–182 (1997)
16. Gâinâ, D., Zhang, M., Chiba, Y., Arimoto, Y.: Constructor-based inductive theorem prover. In: Proc. 5th CALCO. LNCS, vol. 8089, pp. 328–333. Springer (2013)
17. Goguen, J.: Theorem Proving and Algebra. [Unpublished Book Draft] (2006)
18. Hrbacek, K., Jech, T.: Introduction to Set Theory, 3rd ed. Marcel Dekker (1999)
19. Isabelle/HOL: Web page. https://isabelle.in.tum.de/dist/library/HOL/ (2020 accessed)
20. Isabelle/HOL/Wellfounded: Web page. https://isabelle.in.tum.de/dist/library/HOL/HOL/Wellfounded.html (2020 accessed)
21. Maude: Web page. http://maude.cs.uiuc.edu/ (2020 accessed)
22. Ogata, K., Futatsugi, K.: Proof scores in the OTS/CafeOBJ method. In: Proc. 6th IFIP WG 6.1, FMOODS 2003. pp. 170–184 (2003)
23. Otter: Web page. http://www.cs.unm.edu/~mccune/otter/ (2020 accessed)
24. Preining, N., Ogata, K., Futatsugi, K.: Liveness properties in CafeOBJ. In: Proc. 24th LOPSTR, 2014. LNCS, vol. 8981, pp. 182–198. Springer (2015)
25. PVS: Web page. http://pvs.csl.sri.com/ (2020 accessed)
26. Riesco, A., Ogata, K.: Prove it! inferring formal proof scripts from CafeOBJ proof scores. ACM Trans. Softw. Eng. Methodol. **27**(2), 6:1–6:32 (2018)
27. Riesco, A., Ogata, K., Futatsugi, K.: A maude environment for CafeOBJ. Formal Asp. Comput. **29**(2), 309–334 (2017)
28. Terese: Term Rewriting Systems. Cambridge University Press (2003)

A Rule-based System for Computation and Deduction in Mathematica*

Mircea Marin[1], Besik Dundua[2], and Temur Kutsia[3]

[1] Faculty of Mathematics and Informatics
Department of Computer Science
West University of Timişoara, Timişoara, Romania
mircea.marin@e-uvt.ro
[2] Ilia Vekua Institute of Applied Mathematics
Ivane Javakhishvili Tbilisi State University, Tbilisi, Georgia
bdundua@gmail.com
[3] Research Institute for Symbolic Computation
Johannes Kepler University, Linz, Austria
kutsia@risc.jku.at

Abstract. ρLog is a system for rule-based programming implemented in Mathematica, a state-of-the-art system for computer algebra. It is based on the usage of (1) conditional rewrite rules to express both computation and deduction, and of (2) patterns with sequence variables, context variables, ordinary variables, and function variables, which enable natural and concise specifications beyond the expressive power of first-order logic. Rules can be labeled with various kinds of strategies, which control their application. Our implementation is based on a rewriting-based calculus proposed by us, called ρLog too. We describe the capabilities of our system, the underlying ρLog calculus and its main properties, and indicate some applications.

Keywords: Rewriting-based calculi · Strategies · Constrained Rewriting.

1 Introduction

In this paper we present our main contributions to the design and implementation of a system for rewriting-based declarative programming with rewriting strategies. The system is called ρLog, and is implemented as an add-on package on top of the rewriting and constraint solving capabilities of Mathematica [16]. It provides (1) a logical framework to reason in theories whose deduction rules can be specified by conditional rewrite rules of a very general kind, and (2) a semantic framework where computations are sequences of state transitions modelled as rewrite steps controlled by strategies.

ρLog has some outstanding capabilities:

* This work was supported by Shota Rustaveli National Science Foundation of Georgia under the grant no. FR17 439 and by the Austrian Science Fund (FWF) under project 28789-N32.

1. It has a specification language which, in addition to term variables, allows the use of sequence variables, function variables, and context variables. Sequence variables are placeholders for sequences of terms; function variables are placeholders for function symbols; and context variables are placeholders for functions of the form $\lambda x.t$ where t is a term with a single occurrence of the term variable x. These new kinds of variables enable natural and concise specifications beyond the expressive power of first-order logic. For example, solving equations involving sequence variables has applications in AI, knowledge management, rewriting, XML processing, and theorem proving.
2. It is based on a rewrite-based calculus proposed by us [14, 15] which integrates novel matching algorithms for the kinds of variables mentioned above, and is sound and complete if we impose some reasonable syntactic restrictions [12].
3. It is seamlessly integrated with the constraint solving capabilities of Mathematica, a state of the art system with nearly 5000 built-in functions covering all areas of technical computing. As a result, we can use ρLog to tackle a wide range of applications.
4. It can generate human-readable traces of its computations and deductions. In particular, this capability can be turned into a tool to generate human-readable proof certificates for deduction.

The paper is structured as follows. Section 2 contains a brief description of ρLog and its core concepts: programs, strategies and queries. In Sect. 2.1 we present the rewriting calculus implemented by us. Section 3 indicates some applications. Section 4 concludes.

2 The ρLog system

A program consists of rule declarations

`DeclareRules[`$rule_1, \ldots, rule_m$`]`

where $rule_1, \ldots, rule_m$ are labeled conditional rewrite rules of the form

$$t \to_{stg} t' /; cond_1 \wedge \ldots \wedge cond_n \tag{1}$$

with the intended reading "t reduces to t' with strategy stg (notation $t \to_{stg} t'$) whenever $cond_1$ and ... and $cond_n$ hold." Such a rule is a partial definition for strategy stg. In the special case when $n = 0$, rules become unconditional: $t \to_{stg} t'$ has the intended reading "t is reducible to t' with strategy stg." Thus, the rewrite rules of ρLog differ from the usual rewrite rules of a rewrite theory because we label them with terms which we call *strategies*.

To illustrate how reduction works, consider the problem of extracting the smallest number from a non-empty list of numbers. We can achieve this by repeated application of the labeled rule

$\{\mathtt{x}\_, \mathtt{a}\_\_\_, \mathtt{y}\_, \mathtt{b}\_\_\_\} \to_{\text{"}\mathtt{swap}\text{"}} \{\mathtt{y}, \mathtt{a}, \mathtt{x}, \mathtt{b}\} /; (\mathtt{x} > \mathtt{y})$.

until the smallest element is moved at first position in the list, followed by one application of the labeled rule

$\{\mathtt{x}\_, \_\_\_\} \to_{\text{"}\mathtt{first}\text{"}} \mathtt{x}$.

We can declare these labeled rewrite rules as follows:

```
DeclareRules[{x_, a___, y_, b___} →"swap" {y, a, x, b}/; (x > y),
             {x_, ___} →"first" x]
```

If `L` is a nonempty list of numbers and we pose the query

```
ApplyRule[NF["swap"]∘"first",L]
```

then the interpreter of ρLog returns as answer the term `t` which satisfies the reducibility formula $\mathtt{L} \to_{\mathtt{NF}[\text{"}\mathtt{swap}\text{"}]\circ\text{"}\mathtt{first}\text{"}} \mathtt{t}$, and this term is the smallest number in list `L`. Note the following peculiarities of our specification language:

1. $\mathtt{a}, \mathtt{b}, \mathtt{x}, \mathtt{y}$ are variables. They are identified by suffixing their first occurrences[4] in the rule with _ or ___. The variables suffixed by _ are either term variables or function variables, whereas those suffixed by ___ are sequence variables. Thus, $\mathtt{a}, \mathtt{b}$ are sequence variables, and $\mathtt{x}, \mathtt{y}$ are term variables. Like in Prolog, we allow the use of anonymous variables: _ is a nameless placeholder for an element (a function symbol or a term), and ___ is a nameless placeholder for a sequence of terms.
2. `(x>y)` is a boolean condition that is properly interpreted by the constraint logic programming component (CLP) of ρLog.
3. `'NF'` and `'∘'` are predefined general-purpose strategy combinators: $t \to_{\mathtt{NF}[stg]} t'$ holds if t' is a normal form produced by repeated applications of $\to_{stg}$-reduction steps starting from t; and $t \to_{stg_1 \circ stg_2} t'$ holds if $t \to_{stg_1} t'' \to_{stg_2} t'$ holds for some intermediary term t''.

Sequence variables introduce nondeterminism in the reduction process. For example, there are 2 ways to reduce $\{4, 1, 5, 2\}$ with the labeled rule for `"swap"`: $\{4, 1, 5, 2\} \to_{\text{"}\mathtt{swap}\text{"}} \{1, 4, 5, 2\}$ with matcher $\{\mathtt{x} \to 4, \mathtt{a} \to \ulcorner\urcorner, \mathtt{y} \to 1, \mathtt{b} \to \ulcorner 5, 2 \urcorner\}$, $\{4, 1, 5, 2\} \to_{\text{"}\mathtt{swap}\text{"}} \{2, 1, 5, 4\}$ with matcher $\{\mathtt{x} \to 4, \mathtt{a} \to \ulcorner 1, 5 \urcorner, \mathtt{y} \to 2, \mathtt{b} \to \ulcorner\urcorner\}$. Here, $\ulcorner t_1, \ldots, t_n \urcorner$ represents the sequence of terms $t_1, \ldots, t_n$, in this order.

Sequence variables provide a simple way to traverse and process terms of any width. In contrast, context variables allow to traverse terms of any depth. For example, the parametric strategy `"rw"` defined by the rule

$\mathtt{C}\_^{\circ}[\mathtt{s}\_] \to_{\text{"}\mathtt{rw}\text{"}[\mathtt{r}\_]} \mathtt{C}^{\circ}[\mathtt{t}]/; (\mathtt{s} \to_{\mathtt{r}} \mathtt{t}\_)$

specifies term rewriting with rules corresponding to parameter `r`. Here, $\mathtt{C}^{\circ}$ is a context variable, and the pattern $\mathtt{C}^{\circ}[\mathtt{s}\_]$ matches a term t in all ways which bind s to a subterm t' of t, and $\mathtt{C}^{\circ}$ to the context in which t' occurs. If `"r"` is the strategy defined by the rule

[4] By 'first occurrence' in (1), we mean first occurrence in the sequence of expressions (stg, t), $cond_1, \ldots, cond_n, t'$.

$\texttt{f\_[f\_[x\_\_\_]]} \to_{\texttt{"r"}} \texttt{f[x]}$

then there are two ways to reduce the term $\texttt{t} = \texttt{a[a[b[b[1, 2]]]]}$ with the labeled rule for strategy `"rw"["r"]`:

- $t \to_{\texttt{"rw"}[\texttt{"r"}]} t_1 = \texttt{a[b[b[1, 2]]]}$ with matcher $\{\texttt{C}^\circ \to \lambda\texttt{x.x}, \texttt{s} \to t, \texttt{t} \to t_1\}$ because $t \to_{\texttt{"r"}} t_1$ with matcher $\{\texttt{f} \to \texttt{a}, \texttt{x} \to \ulcorner\texttt{b[b[1, 2]]}\urcorner\}$, and
- $t \to_{\texttt{"rw"}[\texttt{"r"}]} t_2 = \texttt{a[a[b[1, 2]]]}$ with matcher $\{\texttt{C}^\circ \to \lambda\texttt{x.a[a[x]]}, \texttt{s} \to \texttt{b[b[1, 2]]}, \texttt{t} \to \texttt{b[1, 2]}\}$ because $\texttt{b[b[1, 2]]} \to_{\texttt{"r"}} \texttt{b[1, 2]}$ with matcher $\{\texttt{f} \to \texttt{b}, \texttt{x} \to \ulcorner\texttt{1, 2}\urcorner\}$.

This strategy definition illustrates another feature of ρLog: variable `f` matches a function symbol. In first-order logic, variables are placeholders for terms only, but some functional programming languages, including Mathematica, go beyond this limitation and allow variables to match function symbols too.

Matching with sequence variables and context variables is finitary [9, 8]. Algorithms which enumerate all finitely many matchers with terms containing such variables are described in [11, 10], and are used by the interpreter of ρLog.

The constraints in the conditional part of a rule are of three kinds: (1) reducibility formulas $t \to_{stg} t'$, (2) irreducibility formulas $t \not\to_{stg} t'$, (3) any boolean formulas expressed in the host language of Mathematica.

ρLog is designed to work with three kinds of **strategies**:

1. Atomic strategies, designated by a string identifier `sId`, and defined by one or more labeled rules of the form

$$t \to_{\texttt{sId}} t'/; cond_1 \wedge \ldots \wedge cond_n.$$

   The following atomic strategies are predefined:
   `"Id"`: $t \to_{\texttt{"Id"}} t'$, abbreviated $t \equiv t'$, holds if and only if t' matches t.
   `"elem"`: $l \to_{\texttt{"elem"}} e$ holds if and only if e matches an element of list l.
   `"subset"`: $t \to_{\texttt{"subset"}} t'$ holds if and only if t' matches a subset of set t.
2. Parametric strategies, defined by rules of the form

$$t \to_{sId[s_1,\ldots,s_m]} t'/; cond_1 \wedge \ldots \wedge cond_n$$

   where sId is the strategy identifier (a string) and $s_1, \ldots, s_m$ are its parameters. The parameters provide syntactic material to be used in the conditional part and result of the rule application.
3. Composite strategies, built from other strategies with strategy combinators. In ρLog, the following combinators are predefined:
   composition: $t \to_{stg_1 \circ stg_2} t'$ holds if $t \to_{stg_1} t'' \to_{stg_2} t'$ holds for some intermediary term t''
   choice: $t \to_{stg_1 | stg_2} t'$ holds if either $t \to_{stg_1} t'$ holds or $t \to_{stg_2} t'$ holds.
   repetition: $t \to_{stg^*} t'$ holds if either $t \to_{\texttt{"Id"}} t'$ holds or there exist $u_1, \ldots, u_n$ such that $t \to_{stg} u_1 \to_{stg} \ldots \to_{stg} u_n \to_{stg} t'$ holds.
   first choice: $t \to_{\texttt{Fst}[stg_1,\ldots,stg_n]} t'$ holds if there exists $1 \leq i \leq n$ such that $t \to_{stg_i} t'$ and $t \to_{stg_j} t'$ hold for all $i < j \leq n$.
   normalization: $t \to_{\texttt{NF}[stg]} t'$ holds if $t \to_{stg^*} t'$ and $t' \not\to_{stg} \_$ hold.

Queries are formulas of the form $cond_1 \wedge \ldots \wedge cond_n$ where $cond_i$ must be of the same kind as the formulas from the conditional parts of rules. For $n = 0$ we obtain the vacuously true query, which we denote by $\top$.

We can submit to ρLog requests of the form

`Request[`$cond_1 \wedge \ldots \wedge cond_n$`]` or `RequestAll[`$cond_1 \wedge \ldots \wedge cond_n$`]`

They instruct the system to compute one (resp. all) substitution(s) for the variables in the formula $cond_1 \wedge \ldots \wedge cond_n$ for which it holds with respect to the current program. For example, if the current program contains the previous definition of the atomic strategy `"swap"`, then the request

`Request[`$\{4, 1, 5, 2\} \rightarrow$`"swap" x_]`

computes the substitution $\{\mathtt{x} \rightarrow \{1, 4, 5, 2\}\}$, whereas the request

`RequestAll[`$\{4, 1, 5, 2\} \rightarrow$`"swap" x_]`

computes the set of all substitutions $\{\{\mathtt{x} \rightarrow \{1, 4, 5, 2\}\}, \{\mathtt{x} \rightarrow \{2, 1, 5, 4\}\}\}$ for which the reducibility formula $\{4, 1, 5, 2\} \rightarrow$`"swap" x` holds.

Another use of ρLog is to compute a reduct of a term with respect to a strategy. The request

`ApplyRule[`stg, t`]`

instructs ρLog to compute one (if any) reduct of t with respect to strategy stg, that is, a term t' such that the reducibility formula $t \rightarrow_{stg} t'$ holds. ρLog reports `"no solution found."` if there is no reduct of t with stg. ρLog can also be instructed to find *all* reducts of a term with respect to a strategy, with

`ApplyRuleList[`stg, t`]`

More information about ρLog can be found at

`http://staff.fmi.uvt.ro/~mircea.marin/rholog/`

2.1 The ρLog calculus

The ρLog system is designed to solve problems of the following kind:

Given a rewrite theory represented by a program P, and a query Q,
Find one, or all, substitutions σ for which formula $\sigma(Q)$ holds in the rewrite theory represented by P.

There is no effective method to solve this problem in full generality because syntactic unification of terms with our kinds of variables is infinitary [8]. We can avoid this difficulty by imposing syntactic restrictions on the structure of programs and queries, that guarantee the possibility to use matching instead of unification (we already mentioned that matching with terms containing the kinds of variables recognized by ρLog is finitary). Therefore, the ρLog calculus is designed to solve the following restricted version of the previous problem:

Given a rewrite theory represented by a deterministic program P, and a deterministic query Q,

Find one, or all, substitutions σ for which formula $\sigma(Q)$ holds in the rewrite theory represented by P.

Here, the notion of determinism is defined as follows: if $vars(E)$ denotes the set of variables in a syntactic construct E then

- If X is a set of variables and *cond* is a component formula of a query or of the conditional part of a rule, then *cond* is X-*deterministic* if either
  - *cond* is $t \rightarrow_{stg} t'$ or $t \nrightarrow_{stg} t'$ with $vars(t) \cup vars(stg) \subseteq X$, or
  - *cond* is a formula in which all predicate symbols are predefined in Mathematica, and $vars(cond) \subseteq X$.
- a rule $t \rightarrow_{stg} t'/; cond_1 \wedge \ldots \wedge cond_n$ is *deterministic* if $vars(t') \subseteq vars(t) \cup vars(stg) \cup \bigcup_{i=1}^{n} vars(cond_i)$ and, for all $1 \leq i \leq n$, $cond_i$ is X_i-deterministic where $X_i = vars(t) \cup vars(stg) \cup \bigcup_{j=1}^{i-1} vars(cond_j)$.
- a query $cond_1 \wedge \ldots \wedge cond_n$ is *deterministic* if, for all $1 \leq i \leq n$, $cond_i$ is X_i-deterministic where $X_i = \bigcup_{j=1}^{i-1} vars(cond_j)$.

Our calculus is, in essence, SLDNF-resolution with leftmost literal selection: every rule $t \rightarrow_{stg} t'/; cond_1 \wedge \ldots \wedge cond_n$ is logically equivalent with the clause

$$t \rightarrow_{stg} \mathtt{x}/; cond_1 \wedge \ldots \wedge cond_n \wedge (t' \equiv \mathtt{x\_})$$

where `x` is a fresh term variable and we can use resolution with respect to this equivalent clauses. The main difference from resolution in first-order logic is that, instead of using the auxiliary function $mgu(t, t')$ to compute a most general unifier of two terms, we use the auxiliary function $mcsm(t, t')$ which computes the finitely many matchers between a term t and a ground term t'.

Inference rules. The calculus has inference rules for the judgment $Q \rightsquigarrow_{\sigma} Q'$ with intended reading "query Q is reducible to Q' if substitution σ is performed." We use the notation

$$\frac{H_1 \quad \ldots \quad H_n}{Q \rightsquigarrow_{\sigma} Q'}$$

for an inference rule that allows us to conclude that $Q \rightsquigarrow_{\sigma} Q'$ holds if the assumptions $H_1, \ldots, H_n$ hold. Also, we write

- $Q_0 \rightsquigarrow_{\sigma}^{*} Q_n$, or just $Q_0 \rightsquigarrow^{*} Q_n$ whenever we succeed to infer a sequence of judgments $Q_0 \rightsquigarrow_{\sigma_1} Q_1 \ldots \rightsquigarrow_{\sigma_n} Q_n$, and σ is the restriction of substitution $\sigma_1 \ldots \sigma_n$ to $vars(Q_0)$.
- $Q_0 \not\rightsquigarrow^{*} Q_n$ whenever we finitely fail to infer that $Q_0 \rightsquigarrow^{*} Q_n$ holds.

The inference rules of ρLog are shown in Fig. 1. They differ from those of the initial ρLog calculus in some important ways:

1. The current version allows to have unrestricted Mathematica constraints in the specification of queries and conditional parts of rules. In this way, we achieve full integration of the CLP component of our system with the constraint solving capabilities of Mathematica.
   The inference rule of ρLog for such constraints is the last one from Fig. 1.
2. We introduced parametric strategies, that is, strategies with arguments that get instantiated during the reduction process. They enable natural and concise specifications of many kinds of rules, like those for strict and lazy evaluation. The inference rule of ρLog for this feature is the first one from Fig. 1: it matches both the left side and the strategy of the selected rule with the left side and strategy of the reducibility formula selected from the query.

$$\frac{(t'_1 \to_{stg'} t'_2 / ; \bigwedge_{i=1}^{n} cond_i) \in P \quad \sigma \in mcsm((t'_1, stg'), (t_1, stg))}{(t_1 \to_{stg} t_2) \wedge Q \rightsquigarrow_\sigma (\bigwedge_{i=1}^{n} cond_i \wedge (t'_2 \equiv t_2) \wedge Q)}$$

$$\frac{\sigma \in mcsm(t', t)}{(t \equiv t') \wedge Q \rightsquigarrow_\sigma \sigma(Q)} \qquad \frac{\sigma \in mcsm(t', t)}{(t \to_{\texttt{"Id"}} t') \wedge Q \rightsquigarrow_\sigma \sigma(Q)}$$

$$\frac{(t \to_{stg} t') \not\rightsquigarrow^* \top}{(t \not\to_{stg} t') \wedge Q \rightsquigarrow_{\{\}} Q}$$

$$\frac{(t \to_{stg} t') \rightsquigarrow^*_\sigma \top}{(t \to_{\texttt{Fst}[stg,\ldots]} t') \wedge Q \rightsquigarrow_\sigma \sigma(Q)}$$

$$\frac{(t \to_{stg_1} t') \not\rightsquigarrow^* \top}{(t \to_{\texttt{Fst}[stg_1, stg_2 \ldots, stg_n]} t') \wedge Q \rightsquigarrow_{\{\}} (t \to_{\texttt{Fst}[stg_2, \ldots, stg_n]} t') \wedge Q}$$

$$\frac{(t \to_{stg} \_) \rightsquigarrow^* \top}{(t \to_{\texttt{NF}[stg]} t') \wedge Q \rightsquigarrow_{\{\}} (t \to_{stg \circ \texttt{NF}[stg]} t') \wedge Q}$$

$$\frac{(t \to_{stg} \_) \not\rightsquigarrow^* \top \quad \sigma \in mcsm(t', t)}{(t \to_{\texttt{NF}[stg]} t') \wedge Q \rightsquigarrow_\sigma \sigma(Q)}$$

$$\frac{cond \text{ is a valid Mathematica formula}}{cond \wedge Q \rightsquigarrow_{\{\}} Q}$$

Fig. 1. The inference rules of the ρLog calculus

The proper interpretation of the other composite strategies and predefined parametric strategies is guaranteed by assuming that P contains defining rules for them. For example, the rules for the strategy combinators of ρLog are:

$\texttt{x\_} \to_{\texttt{s1\_}\circ\texttt{s2\_}} \texttt{z} / ; (\texttt{x} \to_{\texttt{s1}} \texttt{y\_}) \wedge (\texttt{y} \to_{\texttt{s2}} \texttt{z\_})$.
$\texttt{x\_} \to_{\texttt{s1\_|s2\_}} \texttt{z} / ; (\texttt{x} \to_{\texttt{s1}} \texttt{z\_})$.
$\texttt{x\_} \to_{\texttt{s1\_|s2\_}} \texttt{z} / ; (\texttt{x} \to_{\texttt{s2}} \texttt{z\_})$.
$\texttt{x\_} \to_{\texttt{[s\_*]}} \texttt{y} / ; (\texttt{x} \to_{\texttt{"Id"}|(\texttt{s}\circ\texttt{s*})} \texttt{y\_})$.

The set of answers computed by ρLog for Q in a rewrite theory represented by a program P is is $Ans_P(Q) := \{\sigma \mid \text{there is an inference derivation } Q \rightsquigarrow^*_\sigma \top\}$.

Properties. ρLog is a sound calculus: for every $\sigma \in Ans_P(Q)$, the formula $\sigma(Q)$ holds in the rewrite theory presented by P. Unfortunately, ρLog is not complete: some substitutions that satisfy Q w.r.t. P may not be found because the leftmost selection strategy of query components is not fair, and some attempts to compute a derivation $Q \rightsquigarrow^* \top$ may run forever. The same phenomenon happens in most implementations of Prolog: SLDNF resolution with leftmost literal selection is incomplete for the same reason.

Implementations of logic programming languages usually adopt SLDNF resolution with leftmost literal selection, mainly for efficiency reasons; there are other, less efficient literal selection strategies, which preserve completeness. But for ρLog we have no other selection strategies, because this is the only way to avoid the problem of infinitary unification: by preserving determinism of queries.

3 Applications

3.1 Evaluation strategies

Suppose $P = \{t_i \to t'_i/; cond_{i,1} \wedge \ldots \wedge cond_{i,p_i} \mid 1 \leq i \leq n\}$ is a set of conditional rewrite rules that represent a functional program, and we wish to evaluate a term t with respect to P. A straightforward way to encode the rules of P in ρLog is to assign to all of them a common strategy identifier, say `"P"`, to indicate that they all belong to the same program. Thus, a ρLog program for P could be declared as follows:

`DeclareRules["P",`
$t_1 \to t'_1/; cond_{1,1} \wedge \ldots \wedge cond_{1,p_1}$,
$\ldots$
$t_n \to t'_n/; cond_{n,1} \wedge \ldots \wedge cond_{n,p_n}$`]`

The main evaluation strategies in programming language theory are: eager (or strict) and lazy. When confined to expressions consisting of nested function calls, strict evaluation corresponds to innermost rewriting, and lazy evaluation corresponds to outermost rewriting with the conditional rewrite rules of P. If we adopt the small-step operational style, the value of a term is the normal form produced by derivations consisting of innermost (resp. outermost) rewrite steps.

In ρLog, the small-step operational specification of these evaluation strategies is straightforward:

strict evaluation: $t \to_{\texttt{"strict"[s]}} t'$ holds if if t' is obtained by reducing with s an innermost subterm of t. It can be defined by mutual recursion as follows:

```
DeclareRules[
    x_ →"strict"[s_] y/; (x →Fst["sAux"[s],s] y_),
    f_[as___, x_, bs___] →"sAux"[s_] f[as, y, bs]/; (x →"strict"[s] y_)]
```

lazy evaluation: $t \to_{\texttt{"lazy"[s]}} t'$ holds if t' is obtained by reducing with s an outermost subterm of t. It can be defined by mutual recursion as follows:

```
DeclareRules[
    x_ →"lazy"[s_] y/; (x →Fst[s,"lAux"[s]] y_),
    f_[as___, x_, bs___] →"lAux"[s_] f[as, y, bs]/; (x →"lazy"[s] y_)]
```

The strict value of a term t is returned by `ApplyRule[`t`,NF["strict"["P"]]]`, and the lazy value of term t is returned by `ApplyRule[`t`,NF["lazy"["P"]]]`.

3.2 Natural deduction

In logic and proof theory, natural deduction is a proof system whose inference rules are closely related to the "natural" way of reasoning. The general form of an inference rule is

$$\frac{J_1 \quad \ldots \quad J_n}{J}\ (name)$$

where $J_1, \ldots, J_n, J$ are judgments (that is, representations of something that is knowable), and *name* is the name of the inference rule. The judgments above the line are called *premise*s and that below the line is called *conclusion*. For example, Gentzen's proof system LK for natural deduction has inference rules for judgments of the form $L \vdash R$ where L and R are finite (possibly empty) sequences of formulas in first-order logic. Such a judgment is called sequent, and its intended reading is "A formula in R holds if all formulas in L hold." For example, the inference rules of system LK that pertain to the propositional fragment of first-order logic are those shown in Fig. 2, where the metavariables L, L_1, L_2, R, R_1, R_2 denote sequences of formulas, and A, B denote formulas.

$$\frac{}{L_1, A, L_2 \vdash R_1, A, R_2}\ (I)$$

$$\frac{L_1, A, B, L_2 \vdash R}{L_1, A \wedge B, L_2 \vdash R}\ (\wedge L) \qquad \frac{L \vdash R_1, A, R_2 \quad L \vdash R_1, B, R_2}{L \vdash R_1, A \wedge B, R_2}\ (\wedge R)$$

$$\frac{L_1, A, L_2 \vdash R \quad L_1, B, L_2 \vdash R}{L_1, A \vee B, L_2 \vdash R}\ (\vee L) \qquad \frac{L \vdash R_1, A, B, R_2}{L \vdash R_1, A \vee B, R_2}\ (\vee R)$$

$$\frac{L_1, L_2 \vdash A, R \quad B, L_1, L_2 \vdash R}{L, A \Rightarrow B, L_2 \vdash R}\ (\Rightarrow L) \qquad \frac{A, L \vdash R_1, B, R_2}{L \vdash R_1, A \Rightarrow B, R_2}\ (\Rightarrow R)$$

$$\frac{L_1, L_2 \vdash A, R}{L_1, \neg A, L_2 \vdash R}\ (\neg L) \qquad \frac{A, L \vdash R_1, R_2}{L \vdash R_1, \neg A, R_2}\ (\neg R)$$

Fig. 2. System LK: Inference rules for propositional formulas.

System LK is sound and complete. This implies that $L \vdash R$ holds iff it can be derived from the above rules.

Often, we can translate an inference rule $\frac{J_1 \quad \ldots \quad J_n}{J}\ (name)$ into a corresponding rule of ρLog, as follows:

$J \rightarrow_{name}$ `True`$/; (J_1 \rightarrow_{stg}$ `True`$) \wedge \ldots \wedge (J_n \rightarrow_{stg}$ `True`$)$.

where strategy *stg* can be defined such that

> $J \to_{stg} \texttt{True}$ holds iff J can be derived using the inference rules of the proof system under consideration.

This translation technique works well for Gentzen's proof system illustrated in Fig. 2: We obtain the program consisting of the following nine rule declarations:

$$
\begin{aligned}
&\texttt{DeclareRules}[\\
&\quad \{\_\_\_, \texttt{A}\_, \_\_\_\} \vdash \{\_\_\_, \texttt{A}\_, \_\_\_\} \to_{\texttt{"I"}} \texttt{True},\\
&\quad \{\texttt{L1}\_\_\_, \texttt{A}\_ \wedge \texttt{B}\_, \texttt{L2}\_\_\_\} \vdash \texttt{R}\_ \to_{\texttt{"}\wedge\texttt{L"}} \texttt{True}/; (\{\texttt{L1}, \texttt{A}, \texttt{B}, \texttt{L2}\} \vdash \texttt{R} \to_{\texttt{s}} \texttt{True}),\\
&\quad \texttt{L}\_ \vdash \{\texttt{R1}\_\_\_, \texttt{A}\_ \wedge \texttt{B}\_, \texttt{R2}\_\_\_\} \to_{\texttt{"}\wedge\texttt{R"}} \texttt{True}/; (\texttt{L} \vdash \{\texttt{R1}, \texttt{A}, \texttt{R2}\} \to_{\texttt{s}} \texttt{True}) \wedge\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\texttt{L} \vdash \{\texttt{R1}, \texttt{B}, \texttt{R2}\} \to_{\texttt{s}} \texttt{True}),\\
&\quad \{\texttt{L1}\_\_\_, \texttt{A}\_ \vee \texttt{B}\_, \texttt{L2}\_\_\_\} \vdash \texttt{R}\_ \to_{\texttt{"}\vee\texttt{L"}} \texttt{True}/; (\{\texttt{L1}, \texttt{A}, \texttt{L2}\} \vdash \texttt{R} \to_{\texttt{s}} \texttt{True}) \wedge\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\{\texttt{L1}, \texttt{B}, \texttt{L2}\} \vdash \texttt{R} \to_{\texttt{s}} \texttt{True}),\\
&\quad \texttt{L}\_ \vdash \{\texttt{R1}\_\_\_, \texttt{A}\_ \vee \texttt{B}\_, \texttt{R2}\_\_\_\} \to_{\texttt{"}\vee\texttt{R"}} \texttt{True}/; (\texttt{L} \vdash \{\texttt{R1}, \texttt{A}, \texttt{B}, \texttt{R2}\} \to_{\texttt{s}} \texttt{True}),\\
&\quad \{\texttt{L1}\_\_\_, \texttt{A}\_ \Rightarrow \texttt{B}\_, \texttt{L2}\_\_\_\} \vdash \{\texttt{R}\_\_\_\} \to_{\texttt{"}\Rightarrow\texttt{L"}} \texttt{True}/; (\{\texttt{L1}, \texttt{L2}\} \vdash \{\texttt{A}, \texttt{R}\} \to_{\texttt{s}} \texttt{True}) \wedge\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\{\texttt{B}, \texttt{L1}, \texttt{L2}\} \vdash \{\texttt{R}\} \to_{\texttt{s}} \texttt{True}),\\
&\quad \{\texttt{L}\_\_\_\} \vdash \{\texttt{R1}\_\_\_, \texttt{A}\_ \Rightarrow \texttt{B}\_, \texttt{R2}\_\_\_\} \to_{\texttt{"}\Rightarrow\texttt{R"}} \texttt{True}/; (\{\texttt{A}, \texttt{L}\} \vdash \{\texttt{B}, \texttt{R1}, \texttt{R2}\} \to_{\texttt{s}} \texttt{True}),\\
&\quad \{\texttt{L1}\_\_\_, \neg\texttt{A}\_, \texttt{L2}\_\_\_\} \vdash \{\texttt{R}\_\_\_\} \to_{\texttt{"}\neg\texttt{L"}} \texttt{True}/; (\{\texttt{L1}, \texttt{L2}\} \vdash \{\texttt{A}, \texttt{R}\} \to_{\texttt{s}} \texttt{True}),\\
&\quad \{\texttt{L}\_\_\_\} \vdash \{\texttt{R1}\_\_\_, \neg\texttt{A}\_, \texttt{R2}\_\_\_\} \to_{\texttt{"}\neg\texttt{R"}} \texttt{True}/; (\{\texttt{L}, \texttt{A}\} \vdash \{\texttt{R1}, \texttt{R2}\} \to_{\texttt{s}} \texttt{True})]
\end{aligned}
$$

where `s` is an identifier which should be instantiated with strategy *stg*. It could be the choice

$$\texttt{"}\wedge\texttt{L"} \mid \texttt{"}\wedge\texttt{R"} \mid \texttt{"}\vee\texttt{L"} \mid \texttt{"}\vee\texttt{R"} \mid \texttt{"}\Rightarrow\texttt{L"} \mid \texttt{"}\Rightarrow\texttt{R"} \mid \texttt{"}\neg\texttt{L"} \mid \texttt{"}\neg\texttt{R"} \mid \texttt{"I"}$$

but we can do better than that: We can use heuristics in the definition of `s` to reduce the search space for a proof. For example

$$\texttt{Fst}[\texttt{"I"}, \texttt{"}\wedge\texttt{L"} \mid \texttt{"}\wedge\texttt{R"} \mid \texttt{"}\vee\texttt{L"} \mid \texttt{"}\vee\texttt{R"} \mid \texttt{"}\Rightarrow\texttt{L"} \mid \texttt{"}\Rightarrow\texttt{R"} \mid \texttt{"}\neg\texttt{L"} \mid \texttt{"}\neg\texttt{R"}]$$

would be a better specification for `s` because it gives highest priority to rule `"I"` which, if applicable, detects immediately a proof for a sequent.

ρLog can generate a trace of its computation:

`Request[`Q`,Trace→True]`

instructs ρLog to generate and open a Mathematica notebook with human-readable explanations of the rule-based computations that produced an answer or ended with failure. For example

$$
\begin{aligned}
&\texttt{s = Fst}[\texttt{"I"}, \texttt{"}\wedge\texttt{L"} \mid \texttt{"}\wedge\texttt{R"} \mid \texttt{"}\vee\texttt{L"} \mid \texttt{"}\vee\texttt{R"} \mid \texttt{"}\Rightarrow\texttt{L"} \mid \texttt{"}\Rightarrow\texttt{R"} \mid \texttt{"}\neg\texttt{L"} \mid \texttt{"}\neg\texttt{R"}];\\
&\texttt{Request}[\{\} \vdash \{(\texttt{P} \Rightarrow \texttt{Q}) \Rightarrow ((\neg\texttt{Q}) \Rightarrow \neg\texttt{P})\} \to_{\texttt{s}} \texttt{True,Trace→True}]
\end{aligned}
$$

generates a notebook with explanations that certify that the sequent

$$\{\} \vdash \{(\texttt{P} \Rightarrow \texttt{Q}) \Rightarrow ((\neg\texttt{Q}) \Rightarrow \neg\texttt{P})$$

can be derived with the inference rules of system LK. A snapshot of this trace is shown in Fig. 3.

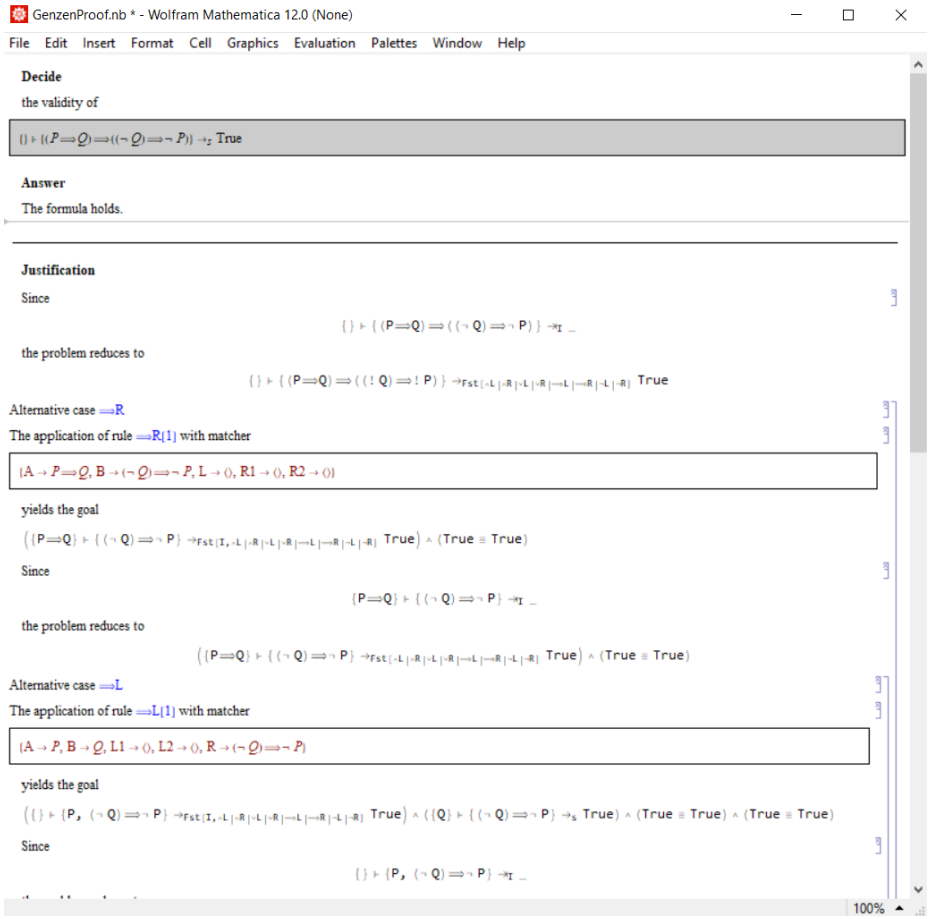


Fig. 3. A human-readable trace of a rule-based deduction.

3.3 Specification and safety-check of access-control models

A fundamental security problem in computing environments is to control the access of subjects to system abstractions (e.g., processes, files, or ports) or resources (e.g., printers). Attribute-based access control (ABAC) is the most recent authorization model, where access rights are granted to users through the use of policies which combine together the attributes of the participating entities (user/subject or subject/object), operations, and the environment relevant to a request. To better understand the power and limitations of this logical access control paradigm, several foundational models have been proposed [7, 6]: They provide minimal set of capabilities that suffice to configure many access control models of practical interest.

Of particular importance is ABAC$_\alpha$ [7], a foundational model for ABAC with a minimal set of capabilities to configure the dominant traditional access control models: discretionary (DAC), mandatory (MAC), and role-based (RBAC).

A system with an ABAC$_\alpha$ access control model can be viewed as a state transition system whose states are triples $\{U, S, O\}$ consisting of the existing users (U), subjects (S), and objects (O), and whose transitions correspond to the six operations from the functional specification of ABAC$_\alpha$: subject creation/deletion/modification, object creation/deletion, and authorized access.

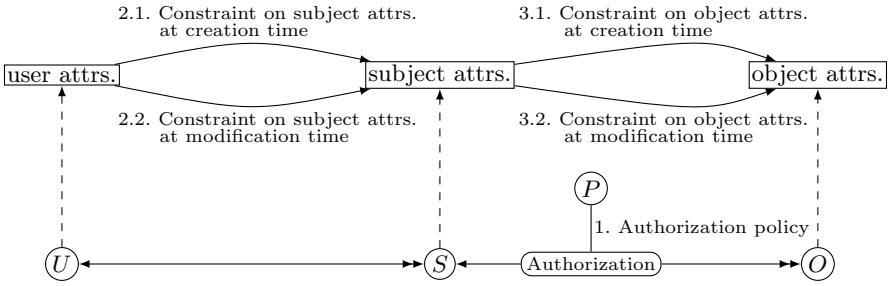


Fig. 4. The structure of ABAC$_\alpha$ model (adapted from [6])

The entities of ABAC$_\alpha$ are identityless—their credentials depend only on their attribute values,—therefore we can represent them as flat terms of the form

$$E[\mathit{attr\text{-}name}_1[\mathit{val}_1], \ldots, \mathit{attr\text{-}name}_n[\mathit{val}_n]]$$

where the term constructor $E \in \{\bullet\mathtt{U}, \bullet\mathtt{S}, \bullet\mathtt{O}\}$ indicates the entity type (user, subject, or object). In [13] we have shown that ρLog is a suitable framework to specify the operational model of ABAC$_\alpha$, because:

1. With parametric strategies, we can separate the declaration of configuration-specific policies from the declaration of policies characteristic to ABAC$_\alpha$. For example, the configuration point which grants to users the right to create a subject has a policy which depends on the specific configuration type (e.g., DAC, MAC, or RBAC). This type-specific policy can be defined with a labeled conditional rewrite rule of the form

   $\mathtt{ConstrSub}[\mathit{uPatt}, \mathit{sPatt}] \rightarrow_{\mathit{cfgTypeId}} \mathtt{True/;} \langle \mathit{CPL\text{-}formula}_1 \rangle .$

   where *uPatt* and *sPatt* are patterns for the interacting user and subject entities, *cfgTypeId* is a configuration type identifier, and $\langle \mathit{CPL\text{-}formula} \rangle_1$ is a boolean formula which expresses the policy as a constraint between the attribute values of user and subject in an instance of CPL, the Common Policy Language [1].
   The other configuration points are for subject modification, object creation/-modification, and granting an access right p for a subject to an object. Similarly, they have rule-based definitions of the form

$\texttt{ConstrModSub}[uPatt, sPatt_1, sPatt_2] \rightarrow_{cfgTypeId} \texttt{True/;} \langle \textit{CPL-formula}_2 \rangle$.
$\texttt{ConstrObj}[sPatt, oPatt] \rightarrow_{cfgTypeId} \texttt{True/;} \langle \textit{CPL-formula}_3 \rangle$.
$\texttt{ConstrModObj}[sPatt, oPatt_1, oPatt_2] \rightarrow_{cfgTypeId} \texttt{True/;} \langle \textit{CPL-formula}_4 \rangle$.
$\texttt{Auth}[p_1, sPatt, oPatt] \rightarrow_{cfgTypeId} \texttt{True/;} \langle \textit{CPL-formula}_{5,1} \rangle$.
...
$\texttt{Auth}[p_n, sPatt, oPatt] \rightarrow_{cfgTypeId} \texttt{True/;} \langle \textit{CPL-formula}_{5,n} \rangle$.

2. The constraint logic programming component of ρLog is based on the constraint solving capabilities of Mathematica and can interpret correctly all boolean constraints expressed in the CPL instances of ABAC_α.
3. The transitions of the operational model of ABAC_α can be defined as parametric strategies whose parameters get bound to configuration-specific identifiers, and can be used to enforce the application of the rule-based specifications of the configuration points.

A fundamental problem for any access control model is safety analysis. For ABAC_α, the safety problem can be stated as follows:

Given an initial state $St_0 = \{U, S_0, O_0\}$ for a particular configuration of ABAC_α, a subject $s \in S_0$, an object $o \in O_0$, and a permission p

Decide if there is a scenario, that is, a sequence of ABAC_α state transitions from St_0 to a state $St_n = \{U, S_n, O_n\}$ where s and o have attribute values that authorize subject s to exercise permission p on object o.

This problem was recently shown to be decidable [1], but no practical algorithm was proposed. In [13], we described a rule-based implementation in ρLog of such an algorithm. Our algorithm decides the safety problem in two steps:

1. First, we compute all possible combinations of attribute values that s may get, if the initial state is St_0. There are finitely many such combinations, which we call descendants of s, and collect them incrementally in a set $sDesc$.
   - We interleave the computation of $sDesc$ with the application of the labeled conditional rewrite rule

     $\texttt{Auth}[p, sPatt, oPatt] \rightarrow_{cfgTypeId} \texttt{True/;} \langle \textit{CPL-formula} \rangle$.

     to early detect if any descendant of s can exercise permission p on o. If yes, the configuration of ABAC_α is unsafe.
2. Next, we compute all possible combinations of attribute values that o may get by the operations exercised by all subjects that can be created and can modify their attribute values. There are finitely many such combinations, which we call *descendants* of o, and collect them incrementally in a set $oDesc$.
   - We interleave the computation of $oDesc$ with the application of the labeled conditional rewrite rule

     $\texttt{Auth}[p, sPatt, oPatt] \rightarrow_{cfgTypeId} \texttt{True/;} \langle \textit{CPL-formula} \rangle$.

     to early detect if any descendant of s can exercise permission p on and descendant of o. If yes, the configuration of ABAC_α is unsafe. If no such situation is detected, the configuration of ABAC_α is reported to be safe.

4 Conclusion

ρLog is a system for rule-based programming with strategies and labeled conditional rewrite rules. It is based on a rewriting calculus designed by us, which has the following characteristics:

1. It accepts specifications with term variables, function variables, sequence variables, and context variables. As a result, the specifications are more natural and concise. For instance, sequence variables and context variables permit matching to descend to arbitrary depth and width in a term represented as a tree. The ability to explore terms in two orthogonal directions in a uniform turned out be useful for querying data available as a large term, like XML documents [10].
2. It is based on sound and complete matching algorithms developed by us.
3. Its range of applications is enlarged significantly by its access to the constraint solving capabilities of Mathematica, its host language.

The ρLog calculus was also used to implement PρLog [4], an experimental tool that extends logic programming with strategic conditional transformation rules. PρLog combines Prolog with the ρLog calculus.

We already mentioned that ρLog can trace the computation of a derivation $Q \rightsquigarrow^*_\sigma \top$ as a human-readable proof that substitution σ satisfies query Q in the rewrite theory represented by a program P [14, 12]. Failed proof attempts $Q \not\rightsquigarrow^* \top$ can be traced too. This capability was inspired from Theorema [3, 5], a theorem prover which influenced the development of ρLog and with which our system shares many features.

We are currently investigating the possibility to extend our calculus with capabilities for approximate reasoning, by solving constraints over several similarity relations [2]. Similarity relations are reflexive, symmetric, and transitive fuzzy relations. They help to make approximate inferences, but pose challenges to constraint solving, since we can not rely on the transitivity property anymore.

References

1. Ahmed, T., Sandhu, R.: Safety of ABCAα Is Decidable. In: Yan, Z., Molva, R., Mazurczyk, W., Kantola, R. (eds.) Network and System Security. pp. 257–272. Springer International Publishing, Cham (2017)
2. Ait-Kaci, H., Pasi, G.: Fuzzy unification and generalization of first-order terms over similar signatures. In: Fioravanti, F., Gallagher, J.P. (eds.) Logic-Based Program Synthesis and Transformation – 27th International Symposium, LOPSTR 2017. LNCS, vol. 10855, pp. 218–234. Springer, Namur, Belgium (2017)
3. Buchberger, B., Jebelean, T., Kriftner, F., Marin, M., Tomuţa, E., Văsaru, D.: A Survey of the Theorema Project. In: Proceedings of ISSAC'97. pp. 384–391. Maui, Hawaii, USA (1997)
4. Dundua, B., Kutsia, T., Reisenberger-Hagmayer, K.: An Overview of PρLog. In: Y. Lierler and W. Taha (ed.) Proceedings of PADL 2017. LNCS, vol. 10137, pp. 34–49. Springer, Paris, France (2017)

5. Jebelean, T., Drămnesc, I.: Synthesis of list algorithms by mechanical proving. JSC **69**, 61–92 (2015)
6. Jin, X., Krishnan, R., Sandhu, R.: A unified attribute-based access control model covering DAC, MAC and RBAC. In: Cuppens-Boulahia, N., Cuppens, F., Garcia-Alfaro, J. (eds.) Data and Applications Security and Privacy XXVI. LNCS, vol. 7371, pp. 41–55. Springer, Berlin, Heidelberg (2012)
7. Jin, X.: Attribute-Based Access Control Models and Implementation in Cloud Infrastructure as a Service. Ph.D. thesis, University of Texas at San Antonio (2014)
8. Kutsia, T.: Solving equations with sequence variables and sequence functions. JSC **42**(3), 352–388 (2007)
9. Kutsia, T.: Solving equations involving sequence variables and sequence functions. In: Buchberger, B., Campbell, J.A. (eds.) Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC 2004, Linz, Austria, September 22-24, 2004, Proceedings. LNCS, vol. 3249, pp. 157–170. Springer (2004). https://doi.org/10.1007/978-3-540-30210-0_14, https://doi.org/10.1007/978-3-540-30210-0_14
10. Kutsia, T., Marin, M.: Can Context Sequence Matching be Used for Querying XML? In: Vigneron, L. (ed.) Proceedings of the 19th International Workshop on Unification (UNIF'05). pp. 77–92. Nara, Japan (2005)
11. Kutsia, T., Marin, M.: Matching with regular constraints. In: Sutcliffe, G., Voronkov, A. (eds.) Proceedings of LPAR 2005. LNAI, vol. 3835, pp. 215–229. Montego Bay, Jamaica (2005)
12. Marin, M., Kutsia, T.: Foundations of the rule-based system ρLog. Journal of Applied Non-Classical Logics **16**(1-2), 151–168 (2006)
13. Marin, M., Kutsia, T., Dundua, B.: A Rule-Based Approach to the Decidability of Safety of ABAC_α. In: Proceedings of the 24th ACM Symposium on Access Control Models and Technologies. pp. 173–178. SACMAT 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3322431.3325416
14. Marin, M., Piroi, F.: Deduction and Presentation in ρLog. In: Proceedings of MKM. ENTCS, vol. 93, pp. 161–182 (2004)
15. Marin, M., Piroi, F.: Rule-Based Programming with Mathematica. In: Proceedings of International Mathematica Symposium (IMS 2004). Banff, Canada (2004)
16. Wolfram, S.: The Mathematica Book. Wolfram Media, 5 edn. (2003)

Compositional Verification in Rewriting Logic (Work in Progress)

Óscar Martín, Alberto Verdejo, and Narciso Martí-Oliet

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

Abstract. Our goal is finding ways to perform compositional verification of systems specified in rewriting logic and Maude. In previous work, we described a means for compositional specification in rewriting logic; this paper continues that work and is based on it. Although some theoretical developments have been needed, our main goal is not creating new techniques, but showing how to adapt existing ones to our setting. We study, in particular, the assume/guarantee technique, and component-wise simulation and equational abstraction. We hope these show the way to adapting other techniques that may be more suitable in each case. A toy example shows how they can be used in practice. This is work in progress.

Keywords: Compositionality · Verification · Rewriting logic · Maude · Model checking · Simulation · Equational abstraction · Assume/guarantee

1 Introduction

Rewriting logic, introduced in [12], is well-suited for the specification of concurrent and non-deterministic systems, among other purposes. Maude, now in its version 3.0 [4,5], is a language based on rewriting logic. It includes in its toolkit an efficient model checker to verify rewriting logic specifications. However, up to now, the compositional verification of such specifications has not been studied.

A compositional verification requires a compositional specification to work on. Thus, in [11], we developed the theoretical basis for compositional specification in rewriting logic and Maude. We summarize that paper in Sect. 2 below.

We consider two techniques that take advantage of compositionality for verification, and show how to adapt them to our setting for rewriting logic. The first, which is the main theoretical novelty of the paper, is the component-wise use of abstraction: abstracting a component induces the abstraction of the whole system, with a potentially reduced effort. The second is the assume/guarantee technique, that allows the verification of isolated individual components, ensuring the result holds for whatever environment the component is to be placed on.

Pending or ongoing work. This paper shows a work in progress. We use paragraphs like this one to mention ways in which our work will be continued.

The interface technique of [3], roughly speaking, works by using a so-called *interface component*, that summarizes the behaviour expected from the whole environment. We will address this technique in the future. □

We are interested in verifying rewrite specifications. However, satisfaction of temporal formulas is usually defined on its semantics, as given by transition structures. That is the reason why the following pages deal more with transition structures than directly with rewrite systems. In [11] we defined egalitarian transition structures as semantics for egalitarian rewrite systems—see Sect. 2.

A composed system is a heterogeneous set of interacting atomic ones. Such a compound can be seen as modelling a distributed system, in which maybe only one of the components is under our control. In this case, global states are meaningless: when we connect a new device to a network, we do not care about the global state of the network. In this view, each component should be analysed or verified individually, to ensure that the component behaves appropriately in an appropriate environment. Conclusions about the behaviour of the whole system must be drawn from the behaviour of each component. This is where compositional verification techniques are appropriate (like the assume/guarantee technique of Sect. 6).

In contrast, what we may call a *compact* system has typically all its components in the same location, interacting by direct contact. A compact system can be compositionally specified because it makes sense for modular engineering, but global states also make sense. In this case, we may prefer to transform a compositionally specified system into a monolithic one and verify the result. The *split* operation, summarized in Sect. 2.3, performs this transformation, and produces what we call *plain* systems, which are quite standard ones.

The difference between both views, the distributed and the compact ones, is not clear cut, with all shades of grey being possible. A compact system can be complex enough that a compositional verification is a good choice. Or the specification of a distributed algorithm can be simple enough that a monolithic verification performs better, counterintuitive as the process may be. Therefore, an essential result is the equivalence of verification methods, that is, the temporal properties of a given system that can be proved by compositional means are the same that can be proved by standard, monolithic means applied to the split system.

This is the plan of the paper. In Sect. 2 we summarize our theoretical basis for compositional specification in rewriting logic. In Sect. 3, we describe the temporal logic against which we verify our systems. In Sect. 4, we define satisfaction of temporal formulas in systems. In Sect. 5, we consider component-wise simulation and equational abstraction. In Sect. 6, we consider the assume/guarantee paradigm and show how it can be adapted to our setting. In Sect. 7 we show a simple illustrative example. Some closing remarks are in Sect. 8.

2 Compositional Specification in Rewriting Logic

The present paper builds on the theoretical basis for compositional specification set in [11]. This section contains a quick introduction to the minimum needed to understand the rest of the paper.

2.1 Egalitarianism

The convenience of putting states and transitions (or actions, or events) at the same level has been claimed many times; see, for instance, the explanations and examples in [9,14]. The specifications of systems and of their temporal properties get clearer or easier when we can refer to both states and transitions, or to propositions defined on them. We have shown in [11] that the same is true for synchronization between components. We proposed there *egalitarian* systems and structures. In them, transitions have meaning by themselves, and the same transition can have several origin and destination states. We introduced the word *stage* to refer to either states or transitions.

Egalitarian rewrite rules include origin and destination state terms and the transition term, and they can also include a condition: $s \multimap\!\!\left[\, t \,\right]\!\!\to s'$ `if` C. For instance, the rule `o` $\multimap\!\!\left[\,\texttt{tick}\,\right]\!\!\to$ `o` can be seen as modelling the ticking of a clock. Here, `tick` is not just a label, but a term. The choice of the transition term and, in particular, the variables in it, has semantic consequences, as explained in [11].

An atomic egalitarian rewrite system is given by a tuple $(D, \leq, \Omega, M, E, R)$, where: $(D, \leq)$ is a kind-complete poset of sorts, in which we require to be included a sort `Stage` with two subsorts `State` and `Trans`; Ω is the signature of operators, including transition constructors and properties (see below); and M, E and R are the sets of membership axioms, equations, and egalitarian rules. (We used S and Σ instead of D and Ω in [11], but here we are using the former for other purposes.) We call these *atomic*, because we compose them to form larger systems as explained below.

The semantics of an atomic egalitarian rewrite system is given by an atomic egalitarian transition structure. This is formalized as a tuple $(Q, T, \to, P, g^0)$, where: Q is the set of states; T is the set of transitions; $\to$ is the bipartite adjacency relation; P is the set of properties, each one declared as $p : Q \cup T \to C_p$ for some codomain C_p; and $g^0 \in Q \cup T$ is the initial stage.

2.2 Properties and Synchronization

When the time comes for specifying the criteria to synchronize components, we do not want them to depend on the internal details of the implementations; instead, we want to use *properties*. These are functions defined on the component's stages. Their return data type can be whatever is appropriate in each case. Boolean-valued properties, in particular, are equivalent to Boolean propositions, typically used in labelled transition structures. A system's properties constitute its interface. They can be thought of as (typed) ports or connectors.

We use the syntax $\mathcal{S}_1 \|_Y \mathcal{S}_2$ for the synchronous composition of two systems. The set Y contains the synchronization criteria, pairs (p_1, p_2), with p_i being a property defined on the stages of system $\mathcal{S}_i$. It means that the two systems have to evolve simultaneously, behaving in such a way that the equality $p_1(g_1) = p_2(g_2)$ is held at all times, with g_1 and g_2 being the current stages. This syntax is used to compose either rewrite systems or transition structures.

More than two systems can be composed, either hierarchically or at once. In [11] we show some equivalences that amount to commutativity and associativity of composition. Thus, we often use the composition operator as a n-ary one, writing, for example, $\mathcal{S}_1 \|_Y (\mathcal{S}_2 \|_Z \mathcal{S}_3)$ as $\|_{Y \cup Z} \mathcal{S}_i$.

Properties get values at states and at transitions, and it can very well happen that a state in a component system needs to be visited at the same time as a transition at the other. Thus, in composed systems, we can rarely talk about pure states or pure transitions, and the name *stage* then refers to any combination of individual states or transitions from different atomic components.

Properties are allowed to be only partially defined. A property that is not meaningful at a given stage can be left undefined there. For a naïve example, suppose a clock is able to tick in two different ways. Then, `typeOfTick` may be a useful property, but it is only meaningful on stages where a tick is actually taking place. Technically, in our setting, which is based on membership equational logic, being partially defined means that some expressions of the form $p(g)$ can be assigned a kind but not a sort.

2.3 The Split

As discussed in the introduction, for practical or theoretical reasons, in some cases, we may prefer to deal with compositional specifications of monolithic systems. For example, the execution engine and other tools that are part of Maude cannot be used on specifications containing synchronous composition or egalitarian features. In [11] we presented an operation, that we call *split*, that transforms a composed egalitarian specification into an equivalent monolithic, standard, *plain* one. The precise sense of that equivalence is described in the cited paper. In short, the computation of split($\|_Y \mathcal{R}_i$) works by:

- splitting, in each component, each egalitarian rule, $s \dashv\!\!\lfloor t \rfloor\!\!\to s'$, into two standard ones, $s \to t$ and $t \to s'$;
- performing a sort of Cartesian product of the previously split rewrite rules, to generate global rules: $\langle s_1, \ldots, s_n \rangle \to \langle s'_1, \ldots, s'_n \rangle$;
- adding conditions to ensure the satisfaction of synchronization criteria.

Thus, all resulting rules have the shape:

$$\langle s_1, \ldots, s_n \rangle \to \langle s'_1, \ldots, s'_n \rangle \;\; \text{if} \bigwedge_{(p_i, p_j) \in Y} p_i(s'_i) = p_j(s'_j)$$

assuming p_i and p_j are properties of $\mathcal{R}_i$ and $\mathcal{R}_j$, respectively.

3 Our Linear Temporal Logic

The temporal logic we use in this work is LTL with two deviations from the standard that we discuss below. LTL is appropriate for compositional verification because its formulas are implicitly universally quantified over execution paths. Thus, when the possible executions of a system are restricted by its interaction with the environment, the remaining ones still satisfy whatever LTL formulas were satisfied in isolation. ACTL* is a superset of LTL that shares this universality property, but LTL is the logic usually studied in relation to rewriting logic, and there exist model checkers and other tools for LTL and Maude.

The first difference between our logic, that we call $\mathrm{LTL}_{\bigcirc}(\Sigma, \Pi)$, and standard LTL is that we avoid the use of the *next* temporal operator represented by $\bigcirc$ (or, alternatively, N, or X). The resulting logic is still quite common in the literature. The reason for avoiding $\bigcirc$ is that its reference (the next stage) is not preserved by composition, nor by refinement. Also, its semantics is not clear when we have to evaluate it at both states and transitions.

The second difference between $\mathrm{LTL}_{\bigcirc}(\Sigma, \Pi)$ and standard LTL is that, instead of atomic propositions, we use in our formulas *properties* and terms involving them—that is what Π is for. We decided that properties are the interfaces of systems, and that they are all that is to be known by the external world. Thus, it makes sense to use them in formulas. For instance, $\Diamond(p = 5)$ and $(p_1 + p_2 < p_3)\,\mathcal{U}\,(p_4 = \texttt{true})$ are valid temporal formulas for us, interpretable on structures in which the respective properties, p, p_i, are defined. Using properties instead of propositions does not increase the power of our formulas, but properties fit better in our setting.

When we get to semantics below, we will need a means to evaluate the expressions involving properties. For now, from a merely syntactic point of view, we need a signature on which such expressions are built. Following [11], we use in this paper membership equational logic, or *MEL* for short. A MEL signature is a tuple $\Sigma = (K, \Sigma, S)$, where K is a set of kinds, $\Sigma = \{\Sigma_{\overline{k},k'} \mid \overline{k} \in K^*, k' \in K\}$ is a family of sets of functions (including constants), and $S = \{S_k \mid k \in K\}$ is a family of sets of sorts. The subsort relation $\leq\, \subseteq S \times S$ is represented by MEL statements like "$t : s_2$ if $t : s_1$", thus, it does not need to be explicitly added to the MEL signature, but can be used as a shortcut. To such a signature we add Π, a set of K-kinded symbols (disjoint from Σ), to represent properties. We denote by $p : k$ the fact that $p \in \Pi$ is given kind $k \in K$.

Definition 1. *Let $\Sigma = (K, \Sigma, S)$ be a MEL signature. Let Π be a set of kinded symbols disjoint from Σ. A formula in $\mathrm{LTL}_{\bigcirc}(\Sigma, \Pi)$ is defined by:*

- *$t = t'$ and $t : s$ are formulas for any terms $t, t' \in T_\Sigma(\Pi)_k$ and sort $s \in S_k$ for some $k \in K$;*
- *if φ and ψ are formulas, then so are $\neg\varphi$, $\varphi \vee \psi$, and $\varphi\,\mathcal{U}\,\psi$.*

We define $\wedge$, $\rightarrow$, $\leftrightarrow$, $\Diamond$, $\Box$, $\mathcal{W}$, and $\mathcal{R}$ as the usual abbreviations.

4 Basic Satisfaction Relations

We need two structures to jointly provide a basis to evaluate the satisfaction of $\mathrm{LTL}_{\diamond}(\Sigma, \Pi)$-formulas. One is a membership algebra on which to evaluate terms in $T_\Sigma(\Pi)$ (like "$p_1 + p_2 < p_3$"). Membership equational logic and membership algebras are described in [13,1]. The other is a transition structure on which temporal formulas make sense: we are using *egalitarian transition structures* and *plain* ones, as defined in [11]. Thus, we are dealing with satisfaction relations of the form $\mathcal{T}, \mathcal{A} \models \varphi$, where $\mathcal{T}$ is a transition structure, $\mathcal{A}$ is a membership algebra, and φ is a temporal formula.

An added complexity for the definition of satisfaction is that properties are partial functions, that is, they are allowed to be left undefined at some (or even all) states and transitions of a system. Thus, the satisfaction of formulas involving such properties cannot be reduced to holding or not holding. In this paper, however, we restrict to the simpler case where all properties are defined at all stages. The most general case is left for future work.

Pending or ongoing work. The natural setting to evaluate satisfaction relations is again MEL. That is, we propose that sometimes $\mathcal{T}, \mathcal{A} \models \varphi$ cannot be further evaluated. We can consider it as *undefined*, and work in a three-valued logic. But better yet would be to take it as an expression in the kind of the Booleans but with no sort. As discussed in [11], allowing properties to be partially defined adds power and flexibility to our framework for composition. So this general case is not to be missed. □

The satisfaction relations studied in this section consider systems as closed entities, with no environment, no interaction with other systems. Sect. 5 and, specially, Sect. 6 deal with open, interacting systems.

Given a membership algebra $\mathcal{A}$ for the MEL signature $\Sigma = (K, \Sigma, S)$, we denote the interpretation of $k \in K$, $s \in S_k$, and $f \in \Sigma_{\overline{k},k'}$ in $\mathcal{A}$, respectively, as $k_\mathcal{A}$, $s_\mathcal{A}$, $f_\mathcal{A}$. In a similar way, given a transition structure $\mathcal{T} = (Q, T, \to, P, g^0)$ and the set of kinded properties Π, we interpret each symbol $p \in \Pi$ as a function $p_\mathcal{T} \in P$. If $p : k$ (that is, $p \in \Pi$ has been given kind $k \in K$), we require that $p_\mathcal{T} : Q \cup T \to k_\mathcal{A}$ (that is, the interpretation $\Pi \to P$ is kind-preserving). In what follows, we call *Π-transition structures* to the ones that satisfy this requirement of including kind-preserving interpretations for the symbols in Π (technically, we should say *(Σ, Π)-transition structures*, because kinds are declared in Σ, but let's keep the notation a little simpler). And, as usual, a *Σ-algebra* is a membership algebra for the MEL signature Σ.

Syntactically speaking, the role of Π in $T_\Sigma(\Pi)$ is equivalent to that of typed variables in standard term algebras. The mapping $p \mapsto p_\mathcal{T}(g^0)$ can be seen as an assignment for the pseudo-variables in Π. Thus, there is a unique homomorphism from the term algebra $T_\Sigma(\Pi)$ to the algebra $\mathcal{A}$ that extends that assignment (see [6, p. 21], or [8, p. 32]). For $t \in T_\Sigma(\Pi)$, we denote as $t_{\mathcal{T},\mathcal{A}}$ the image of t under that homomorphism, that is, the interpretation of the term t in $\mathcal{T}$ and $\mathcal{A}$.

Definition 2. *Consider the following given:*

– *a MEL signature* $\Sigma = (K, \Sigma, S)$,
– *a set* Π *of* K*-kinded symbols, disjoint from* Σ,
– *a membership* Σ*-algebra* $\mathcal{A}$*, and*
– *an atomic egalitarian* Π*-transition structure* $\mathcal{T} = (Q, T, \to, P, g^0)$.

Let $t, t' \in T_\Sigma(\Pi)_k$ *and* $s \in S_k$ *for some* $k \in K$*, and let* φ, ψ *be formulas in* $\mathrm{LTL}_\circledcirc(\Sigma, \Pi)$*. Also, let* $\overline{g}$ *denote a path in* $\mathcal{T}$*, that is, a sequence of stages* $g^0 \to g^1 \to \ldots$ *starting at* $\mathcal{T}$*'s initial stage. The satisfaction relation* $\mathcal{T}, \mathcal{A} \models \varphi$ *is defined by:*

– $\mathcal{T}, \mathcal{A} \models t = t'$ *iff* $t_{\mathcal{T},\mathcal{A}} = t'_{\mathcal{T},\mathcal{A}}$;
– $\mathcal{T}, \mathcal{A} \models t : s$ *iff* $t_{\mathcal{T},\mathcal{A}} \in s_{\mathcal{A}}$;
– *otherwise,* $\mathcal{T}, \mathcal{A} \models \varphi$ *iff for each path* $\overline{g}$ *in* $\mathcal{T}$ *starting at its initial stage, we have* $\mathcal{T}, \mathcal{A}, \overline{g} \models \varphi$.

Satisfaction of a formula by a path is defined by:

– $\mathcal{T}, \mathcal{A}, \overline{g} \models t = t'$ *iff* $\mathcal{T}, \mathcal{A} \models t = t'$;
– $\mathcal{T}, \mathcal{A}, \overline{g} \models t : s$ *iff* $\mathcal{T}, \mathcal{A} \models t : s$;
– $\mathcal{T}, \mathcal{A}, \overline{g} \models \neg\varphi$ *iff not* $\mathcal{T}, \mathcal{A}, \overline{g} \models \varphi$;
– $\mathcal{T}, \mathcal{A}, \overline{g} \models \varphi \vee \psi$ *iff* $\mathcal{T}, \mathcal{A}, \overline{g} \models \varphi$ *or* $\mathcal{T}, \mathcal{A}, \overline{g} \models \psi$;
– $\mathcal{T}, \mathcal{A}, \overline{g} \models \varphi\ \mathcal{U}\ \psi$ *iff there is some* $i \geq 0$ *such that* $\mathcal{T}(g^i), \mathcal{A}, \overline{g}^i \models \psi$*, and for all* $j < i$*, we have* $\mathcal{T}(g^j), \mathcal{A}, \overline{g}^j \models \varphi$*, where* $\mathcal{T}(g^i)$ *is* $\mathcal{T}$ *with its initial stage replaced by* g^i*, that is,* $\mathcal{T}(g^i) = (Q, T, \to, P, g^i)$*, and* $\overline{g}^i$ *is the result of removing from* $\overline{g}$ *the first* i *stages.*

If $\mathcal{T} = (Q, \to, P, q^0)$ *is instead a plain transition structure, the definition is the same as above, just replacing "stage" by "state", and each* g^i *by a* q^i.

Finally, if $\mathcal{T}$ *is a non-atomic egalitarian transition structure, we define* $\mathcal{T}, \mathcal{A} \models \varphi$ *as equivalent to* $\mathrm{split}(\mathcal{T}), \mathcal{A} \models \varphi$.

Boolean properties can be viewed as atomic propositions in disguise. Thus, in the particular case in which all properties are Boolean (and completely defined) and all atomic formulas are of the form $p = \texttt{true}$ for some $p \in \Pi$, our definition agrees with the standard one for LTL.

Proposition 1. *Let* $\mathcal{T}$ *be an atomic egalitarian transition structure,* $\mathcal{A}$ *a membership algebra, and* φ *an* $\mathrm{LTL}_\circledcirc(\Sigma, \Pi)$ *formula. We have* $\mathcal{T}, \mathcal{A} \models \varphi$ *iff* $\mathrm{split}(\mathcal{T}), \mathcal{A} \models \varphi$.

Pending or ongoing work. The extended version of this paper will include a proof that equivalent structures (that is, similar but different arrangements of the same components) satisfy exactly the same formulas. It will also include a discussion of how this generalizes the standard setting for Kripke structures. □

In [11], we defined a semantics function to assign an egalitarian or plain transition structure to a corresponding rewrite system: $\mathcal{R} \mapsto \mathrm{sem}(\mathcal{R})$. A rewrite system specification, moreover, includes in itself a membership equational specification, from which a canonical initial membership algebra can be obtained (as

proved in [13,1]): $\mathcal{R} \mapsto \mathcal{A}(\mathcal{M}(\mathcal{R}))$. This algebra $\mathcal{A}(\mathcal{M}(\mathcal{R}))$ is larger than strictly needed. For instance, it contains interpretations for state and transition sorts, while we only need it for evaluating expressions involving values of properties. A mere Boolean algebra would do in the extreme case when all properties are Boolean. A larger algebra, however, is no problem for our needs. Therefore, the two ingredients on which to define a satisfaction relation can be obtained from a single rewrite specification.

The MEL signature $\Sigma = (K, \Sigma, S)$ for which $\mathcal{A}(\mathcal{M}(\mathcal{R}))$ is a Σ-algebra (for $\mathcal{R}$ an atomic or plain rewrite system) is determined by its sets of kinds, non-property operators, and sorts. The set of property symbols Π for which $\mathrm{sem}(\mathcal{R})$ is a Π-transition structure is the set of property operators declared in $\mathcal{R}$.

Definition 3. *Let $\mathcal{R}$ be an atomic egalitarian rewrite system, or a plain one. Let φ be an $\mathrm{LTL}_{\mathbb{G}}(\Sigma, \Pi)$-formula, for Σ and Π induced by $\mathcal{R}$. We define $\mathcal{R} \models \varphi$ by $\mathrm{sem}(\mathcal{R}), \mathcal{A}(\mathcal{M}(\mathcal{R})) \models \varphi$. Also, for $\mathcal{R}$ non-atomic, we define $\mathcal{R} \models \varphi$ by $\mathrm{split}(\mathcal{R}) \models \varphi$.*

An effect of these definitions is that it is possible to specify a system compositionally, as a set of interacting atomic ones, apply then the split operation to generate an equivalent plain system, and study satisfaction of temporal formulas in it. In short: specify compositionally, verify monolithically. Our goal, however, is to take advantage of compositionality also for verification. The next section starts exploring in this direction.

5 Component-wise Simulation and Equational Abstraction

Abstracting a system means producing a smaller one that behaves in some way the same as the original, so that we can analyse the small one and draw conclusions valid for both—see [2], for example. It is a particular case of simulation. We show in this section how abstraction and simulation can be performed compositionally, that is, abstracting one component produces also an abstraction for the result of the composition. Equational abstraction is described and studied in [15]. It is the best studied kind of abstraction in rewriting logic; we consider it below.

In this section we still consider only, for simplicity, the case in which properties are defined at all states and transitions.

Definition 4. *Consider given Σ, Π, and two atomic egalitarian Π-transition structures $\mathcal{T}_i = (Q_i, T_i, \to_i, P_i, g_i^0)$ for $i = 1, 2$. A* simulation $\mathsf{S} : \mathcal{T}_1 \to \mathcal{T}_2$ *is a relation $\mathsf{S} \subseteq (Q_1 \cup T_1) \times (Q_2 \cup T_2)$ such that:*

- $g_1^0 \mathsf{S} g_2^0$;
- *if $g_1 \mathsf{S} g_2$ then for all $p \in \Pi$ we have $p_{\mathcal{T}_1}(g_1) = p_{\mathcal{T}_2}(g_2)$;*
- *if $g_1^1 \mathsf{S} g_2^1$ and $g_1^1 \to_1 g_1^2$ then there exists a path in $\mathcal{T}_2$, $g_2^1 \to_2 \cdots \to_2 g_2^n$, with $n \geq 1$, such that $g_1^1 \mathsf{S} g_2^i$ for $i = 1, \ldots, n-1$ and $g_1^2 \mathsf{S} g_2^n$.*

If both S *and* S^{-1} *are simulations, we say that* S *is a* bisimulation.

The definition for plain transition structures is a straightforward adaptation of the above.

In particular, the third item above allows $n = 1$, so that the requirement becomes $g_1^1 \mathsf{S} g_2^1$ and $g_1^2 \mathsf{S} g_2^1$—so to speak, $\mathcal{T}_1$ advances while $\mathcal{T}_2$ waits.

Implicit in the definition of simulation are the interpretations of the symbols in Π, that is, the maps $\Pi \to P_i$. When we want to make explicit that $p_1 \in P_1$ and $p_2 \in P_2$ are interpretations of the same symbol in Π as used for the simulation S, we write $p_1 \mathsf{S} p_2$.

The concept defined above is usually called *stuttering (bi)simulation*. However, we decided to avoid the use of the *next* operator in our temporal logic, and this supposes that we are always working in a stuttering way. Also our related decision that only the values of properties are important, and whether the system internally moves stage or not is irrelevant from the outside amounts to the same. So, we drop the adjective and call our concept just *(bi)simulation*.

Proposition 2. *Consider* Σ, Π, *a* Σ*-algebra* $\mathcal{A}$, *and two atomic egalitarian* Π*-transition structures* $\mathcal{T}_1$ *and* $\mathcal{T}_2$. *If there exists a simulation* $\mathsf{S} : \mathcal{T}_1 \to \mathcal{T}_2$, *then for every* $\mathrm{LTL}_{\bigcirc}(\Sigma, \Pi)$*-formula* φ *we have that* $\mathcal{T}_2, \mathcal{A} \models \varphi$ *implies* $\mathcal{T}_1, \mathcal{A} \models \varphi$. *If* S *is a bisimulation, then* $\mathcal{T}_1, \mathcal{A} \models \varphi$ *iff* $\mathcal{T}_2, \mathcal{A} \models \varphi$.

The result for plain transition structures is a straightforward adaptation of the above.

The following theorem is our main result about simulations, stating that component-wise simulations induce global ones. It can be seen as an adaptation of [2, Chap. 12]. First, a lemma and an auxiliary definition.

Lemma 1. *If* A_i *is an atomic egalitarian* Π_i*-transition structure for each* i, *then* $\mathrm{split}(\|_Y A_i)$ *is a plain* Π*-transition structure for* $\Pi = \biguplus_i \Pi_i$.

Definition 5. *Let* A_i *and* B_i *be atomic egalitarian transition structures such that there are simulations* $\mathsf{S}_i : A_i \to B_i$ *for* $i = 1, \ldots, n$. *Consider the compositions* $\|_Y A_i$ *and* $\|_Z B_i$. *We say that* Y *and* Z *are* corresponding sets of synchronization criteria, *if: (i) for each* $\langle p_i, p_j \rangle \in Y$ *with* $p_i \in P_{A_i}$ *and* $p_j \in P_{A_j}$, *there is* $\langle q_i, q_j \rangle \in Z$ *with* $q_i \in P_{B_i}$ *and* $q_j \in P_{B_j}$ *such that* $p_i \mathsf{S}_i q_i$ *and* $p_j \mathsf{S}_j q_j$; *(ii) the converse, that is, the same swapping the roles of* Y *and* Z.

Theorem 1. *Let* $A_i = (Q_{A_i}, T_{A_i}, \to_{A_i}, P_{A_i}, g^0_{A_i})$ *and* $B_i = (Q_{B_i}, T_{B_i}, \to_{B_i}, P_{B_i}, g^0_{B_i})$ *be atomic egalitarian transition structures such that there are simulations* $\mathsf{S}_i : A_i \to B_i$ *for* $i = 1, \ldots, n$. *(The identity is a (bi)simulation, so this includes the case that* $A_i = B_i$ *for some or all* i.*) Consider* $\|_Y A_i$ *and* $\|_Z B_i$ *with* Y *and* Z *corresponding sets of synchronization criteria. Then, there is a simulation* $\mathsf{S} : \mathrm{split}(\|_Y A_i) \to \mathrm{split}(\|_Z B_i)$ *(as plain* $(\biguplus_i \Pi_i)$*-transition structures). If all* S_i *are bisimulations,* S *can be taken to be as well.*

A well-known way to implement simulations is by equational abstraction in a rewrite system. Detailed explanations can be found in [15]. In short, on an atomic

egalitarian or plain rewrite system $\mathcal{R} = (D, \leq, \Omega, E, M, R)$ we can perform equational abstraction by adding equations E' to obtain $\mathcal{R}^* = (D, \leq, \Omega, E \cup E', M, R)$, so that states satisfying certain conditions are now equated and considered the same. Then, there is an induced simulation $\mathsf{S} : \mathrm{sem}(\mathcal{R}) \to \mathrm{sem}(\mathcal{R}^*)$.

Theorem 2. *Let A_i and A_i^* be atomic egalitarian rewrite systems such that each A_i^* is an equational abstraction of the corresponding A_i. Consider $\|_Y A_i$ and $\|_Y A_i^*$ for some set of synchronization criteria Y. Then,* $\mathrm{split}(\|_Y A_i^*)$ *can be obtained as an equational abstraction of* $\mathrm{split}(\|_Y A_i)$.

The usual questions about executability hold here, that is, we must ensure: that the new set of equations is ground Church-Rosser and terminating; that the rules are still ground coherent with respect to the new set of equations; and that all properties are *preserved*, that is, $t \equiv_{E \cup E'} t' \implies p(t) = p(t')$. The following result shows that checking whether the global abstraction is executable can be done component-wise.

Theorem 3. *Let A_i be atomic egalitarian rewrite systems. If each A_i is executable (in the sense of the previous paragraph), then so is* $\mathrm{split}(\|_Y A_i)$*, for any set of synchronization criteria Y.*

6 The Assume/Guarantee Technique

Reactive systems are *open*, in the sense that they must be ready to interact with other systems in their environment. This interaction may restrict its possible actions, filtering out the ones not allowed by neighbouring systems. The classical satisfaction relation between a system $\mathcal{S}$ and a temporal formula φ, that we write $\mathcal{S} \models \varphi$, considers the system as if run in isolation—as a non-interacting or *closed* system. For open systems this classical approach may not be suitable. Other techniques have been devised to verify that a component satisfies a given specification in whatever environment it is placed in. Well-known among such techniques is assume/guarantee, first proposed in [16]. This section is devoted to adapting this technique to our setting for verifying rewrite systems. Our simplifying assumption that properties are completely defined still holds.

Satisfaction, according to the assume/guarantee technique, involves two formulas: one stating what this component can assume from the environment; the other stating what it is ready to guarantee based on the assumption and on its own internal behaviour. The notation we are using, inspired from [7], is $\mathcal{T}, \mathcal{A} \models \alpha \blacktriangleright \gamma$, where α is the assumption and γ the guarantee, both of them formulas in $\mathrm{LTL}_{\mathbb{O}}(\Sigma, \Pi)$. Here, as above, $\mathcal{T}$ is a Π-transition structure and $\mathcal{A}$ a membership Σ-algebra.

Reading $\mathcal{T}, \mathcal{A} \models \alpha \blacktriangleright \gamma$ as "the satisfaction of γ is guaranteed if $\mathcal{T}$ is in an environment that satisfies α" is somewhat naïve. Two kinds of problems may arise. First, emerging deadlocks in the composed system that prevent it from satisfying γ, even if a component in isolation does. Second, the lack of *component fairness* can make $\mathcal{T}$ starve, again preventing it from satisfying γ. The precise

description of these problems and their possible solutions are out of the scope of this paper. So, in what follows we assume deadlock freeness and component fairness have been already dealt with.

Pending or ongoing work. Both component fairness and deadlock freeness deserve deeper consideration. □

We can now define satisfaction in the assume/guarantee setting.

Definition 6. *Consider given Σ, Π, $\mathcal{A}$, and an atomic egalitarian $\mathcal{T}$. We define the satisfaction relation $\mathcal{T}, \mathcal{A} \models \alpha \blacktriangleright \gamma$ for two formulas α and γ in $\mathrm{LTL}_{\mathbb{O}}(\Sigma, \Pi)$ by: each execution of $\mathcal{T}$ that satisfies α also satisfies γ.*

The following remark is a direct consequence of the definition.

Proposition 3. *In the conditions of Def. 6, $\mathcal{T}, \mathcal{A} \models \alpha \blacktriangleright \gamma$ iff $\mathcal{T}, \mathcal{A} \models \alpha \to \gamma$.*

This is a known result—see, for example, [10, Theor. 5.1]. And it is most welcome, because it means we can use standard verification tools to verify compositionally. It entails the following unsurprising result.

Proposition 4. *For $\mathcal{T}$, $\mathcal{A}$, φ as above, we have $\mathcal{T}, \mathcal{A} \models \mathtt{true} \blacktriangleright \varphi$ iff $\mathcal{T}, \mathcal{A} \models \varphi$.*

As in previous cases, the definition can be easily adapted for plain transition structures. And also for rewrite systems, in the expected way.

Definition 7. *Let $\mathcal{R}$ be an atomic egalitarian rewrite system, or a plain one. We define $\mathcal{R} \models \alpha \blacktriangleright \gamma$ by $\mathrm{sem}(\mathcal{R}), \mathcal{A}(\mathcal{M}(\mathcal{R})) \models \alpha \blacktriangleright \gamma$.*

6.1 Deduction Rules

Our main goal is not to propose a particular deduction rule for compositional verification, but rather to show how deduction rules and related tools appearing in the literature can (sometimes, at least) be adapted to our setting. We include as witness a simple deduction rule.

Proposition 5. *With the notational conventions used so far, the following deduction rule is correct:*

$$\frac{\mathcal{R}_1 \models \gamma_1 \qquad Y \models \gamma_1 \to \alpha_2 \qquad \mathcal{R}_2 \models \alpha_2 \blacktriangleright \gamma_2}{\mathcal{R}_1 \|_Y \mathcal{R}_2 \models \gamma_2}$$

A note is needed on the meaning of $Y \models \gamma_1 \to \alpha_2$. The formula γ_1 uses the properties and the signature from $\mathcal{R}_1$, while α_2 uses the ones from $\mathcal{R}_2$. The relation between these two syntaxes is given by Y. For example, if Y contains (p_1, p_2), the formulas $\gamma_1 \equiv \Box(p_1 = 5)$ and $\alpha_2 \equiv \Box(p_2 = 5)$ are saying exactly the same thing. We represent that by $Y \models \gamma_1 \leftrightarrow \alpha_2$; in general, the equivalence is not needed, and the implication shown in the rule is enough. A little more formally, $\gamma_1 \to \alpha_2$ is a formula in the signature of $\mathrm{split}(\mathcal{R}_1 \|_Y \mathcal{R}_2)$, and $Y \models \gamma_1 \to \alpha_2$ means that $\gamma_1 \to \alpha_2$ becomes a tautology in LTL by taking into account the equalities between properties implied by Y.

Given that we defined $\mathcal{R}_1 \|_Y \mathcal{R}_2 \models \gamma_2$ as $\mathrm{split}(\mathcal{R}_1 \|_Y \mathcal{R}_2) \models \gamma_2$, this result can be seen as establishing that distributed verification implies monolithic one.

Pending or ongoing work. Instead of implication, we need to address equivalence of the two methods for verification. This requires completeness of the deduction rule, which we have not addressed here, that is, whether it is always possible to find appropriate intermediate formulas so as to make the application of the rule possible. In the restricted case of completely-defined properties, we conjecture the rule is complete. A proof, however, is missing.

This was quite a simple deduction rule. It can be strengthened in several ways. Also, showing some additional rule would yield some variety. In particular, circular deduction rules (in which each component guarantees what the other one needs to assume) are different enough from the one above to deserve our attention. Moreover, some proposals for deduction rules have the nice feature that the intermediate formulas (γ_1 and α_2, in our case) can be obtained automatically. Adapting this to our setting may also be worthwhile. The paper [7] proposes a circular rule with automatic intermediate formula generation. It, and the references therein, may help. □

One more result is worth stating, even in the form of a deduction rule.

Proposition 6. *With the usual notational conventions, the following deduction rule is correct for any Y and φ:*

$$\frac{\mathcal{R}_1 \models \varphi}{\mathcal{R}_1 \|_Y \mathcal{R}_2 \models \varphi}.$$

7 An Example

Let us consider a very simple model of a train, with states representing stations:

```
crl atStation N =[ towards N + 1 ]=> atStation N + 1 if N < 9 .
```

There is a total of 10 stations, numbered 0 to 9. But the transit from station 9 to 0 is different, because it passes through a crossing:

```
rl atStation 9 =[ crossing ]=> atStation 0 .
```

Indeed, we have two trains, modelled exactly the same. Stations and transits are disjoint, except for the crossing, that is shared. We are interested in getting safety and fairness in the access to the crossing. So, we define for each train a Boolean property `isXing` to be true at the transition `crossing` and false everywhere else:

```
eq isXing @ crossing = true .
eq isXing @ (atStation N) = false .
eq isXing @ (towards N) = false .
```

The syntax `P @ G` is the one we use to represent the evaluation of property `P` at stage `G` (which in math we have been writing as `P(G)`). As this is the only property of interest, it makes sense to perform equational abstraction based on it. To that goal, we create in each system a new `Stage` constant, `elsewhere`, and add the equation:

```
ceq G = elsewhere if isXing @ G = false .
```

The result of this equational abstraction is then bisimilar to this:

```
crl elsewhere =[ crossing ]=> elsewhere .
eq isXing @ crossing = true .
eq isXing @ elsewhere = false .
```

Because of Theor. 2, we can use this specification in composed systems instead of the original one, and draw conclusions on it. Let us call the two trains with this abstracted specification `TRAIN1` and `TRAIN2`.

The mutex controller for safe access to the crossing is implemented by the two rules:

```
rl o =[ grant1 ]=> o .
rl o =[ grant2 ]=> o .
```

We call this system `MUTEX` and define in it the Boolean properties `isGranted1` and `isGranted2` to be true at the respective transitions and false everywhere else:

```
eq isGranted1 @ grant1 = true .
eq isGranted2 @ grant2 = true .
eq P @ G = false [owise] .
```

Now we perform the three-way synchronous composition:

```
sync TRAIN1 || TRAIN2 || MUTEX
   on TRAIN1.isXing = MUTEX.isGranted1
   /\ TRAIN2.isXing = MUTEX.isGranted2 .
```

Let us call this composed system `SAFE-TRAINS`. We can easily show

$$\texttt{MUTEX} \models \Box\,\neg(\texttt{isGranted1} \wedge \texttt{isGranted2}),$$

from which, using Prop. 6, we get

$$\texttt{SAFE-TRAINS} \models \Box\,\neg(\texttt{TRAIN1.isXing} \wedge \texttt{TRAIN2.isXing}). \tag{1}$$

We add a final component to take care of fairness in the access to the crossing. Let us call this one `FAIR`. It consists of a single terse rule:

```
crl idle | Past1m2 =[ Now1m2 | Past1m2 ]=> idle | Past1m2 + Now1m2
   if Past1m2 + Now1m2 == 0 or Past1m2 + Now1m2 == 1 .
```

We are implementing a strict kind of fairness that only allows for either the number of crossings to be equal through both railways or to be one unit larger for railway 1. As we are only interested in the differences, not absolute values, we are only storing differences in state and transition terms. Thus, the variable `Past1m2` is a (positive or negative) integer that stores the number of crossings through railway 1 minus the number of them through railway 2. At each transition, the variable `Now1m2` stores that difference for the trains currently crossing. Each state and transition term keeps trace of what is happening at the time, and the past count difference. The transition `0 | Past1m2` represents a positive and equal number of trains passing each way, while the state `idle | Past1m2` represents no trains crossing at the time.

The system `FAIR` only cares about fairness, and allows several trains to cross at a time, any number of them through each railway, indeed, as long as the condition on the difference is held. We want to show that the combined effect of `MUTEX` and `FAIR` on the trains entails the interleaving of crossings.

The properties needed in `FAIR` so that it can control the trains are these:

```
eq some1 @ (Now1m2 | Past1m2) = Now1m2 >= 0 .
eq some2 @ (Now1m2 | Past1m2) = Now1m2 <= 0 .
eq some1 @ (idle | Past1m2) = false .
eq some2 @ (idle | Past1m2) = false .
```

The final system, that we call `FAIR-SAFE-TRAINS`, is:

```
sync SAFE-TRAINS || FAIR
   on TRAIN1.isXing = FAIR.some1
   /\ TRAIN2.isXing = FAIR.some2 .
```

A composed system inherits the properties from its components, so we are allowed to use the properties `isXing` from the trains. Defining

$$\texttt{none} := \neg\texttt{FAIR.some1} \wedge \neg\texttt{FAIR.some2},$$

the formula we want to prove for `FAIR-SAFE-TRAINS` is

$$\texttt{interleaving} := \Box\,\neg(\texttt{FAIR.some1} \wedge (\texttt{FAIR.some1}\;\mathcal{U}\;(\texttt{none} \wedge (\texttt{none}\;\mathcal{U}\;\texttt{FAIR.some1})))).$$

This expresses that two consecutive railway 1 crossings are not possible. Also, we have:

$$Y \models \Box\,\neg(\texttt{TRAIN1.isXing} \wedge \texttt{TRAIN2.isXing}) \leftrightarrow \Box\,\neg(\texttt{FAIR.some1} \wedge \texttt{FAIR.some2}), \quad (2)$$

for Y the synchronization criteria used for `FAIR-SAFE-TRAINS`. Finally, we can check, using, for example, Maude's model checker, that

$$\texttt{FAIR} \models \Box\,\neg(\texttt{FAIR.some1} \wedge \texttt{FAIR.some2}) \rightarrow \texttt{interleaving}. \quad (3)$$

From Statements (1), (2), and (3), and using Prop. 3 and 5, we get the desired result:

$$\texttt{FAIR-SAFE-TRAINS} \models \texttt{interleaving}.$$

8 Closing Remarks

Our proposal for compositionality in rewriting logic is set down in [11]. The current paper shows how verification can be made compositional based on that proposal. Our aim here was not to produce novel theory, but rather to show how methods for compositional verification proposed in the literature for other frameworks can be adapted to ours. This includes the component-wise use of simulation or abstraction, and the assume/guarantee technique. "Adapting" may be a misleading word, as it seems to imply a straightforward procedure. This is not necessarily the case, and indeed some theoretical developments and clarifications have been needed, and surely more will be needed in the future. We hope the work done here makes future adaptations easier. An extended version of this paper is expected to include several generalizations, a discussion of the interface technique and of the possibility of automatically generating intermediate formulas for the assume/guarantee technique. We are also working on an implementation of an extension of Maude's syntax that allows the use of compositional specification and verification along the lines described in our papers.

References

1. Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and Proof in Membership Equational Logic. *TCS*, 236(1-2):35–132, 2000.
2. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2001.
3. Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional Model Checking. In *LICS'89 Proceedings*, pages 353–362. IEEE Computer Society, 1989.
4. Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn Talcott. *Maude Manual (Version 3.0)*, 2019.
5. Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn L. Talcott. Programming and Symbolic Computation in Maude. *JLAMP*, 110, 2020.
6. Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer Berlin Heidelberg, 1985.
7. Karam Abd Elkader, Orna Grumberg, Corina S. Pasareanu, and Sharon Shoham. Automated Circular Assume-Guarantee Reasoning. *Formal Aspects of Computing*, 30(5):571–595, 2018.
8. Matthew Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
9. Ekkart Kindler and Tobias Vesper. ESTL: A Temporal Logic for Events and States. In Jörg Desel and Manuel Silva Suárez, editors, *ICATPN'98 Proceedings*, volume 1420 of *LNCS*, pages 365–384. Springer, 1998.
10. Orna Kupferman and Moshe Y. Vardi. An Automata-Theoretic Approach to Modular Model Checking. *ACM Trans. Program. Lang. Syst.*, 22(1):87–128, 2000.
11. Óscar Martín, Alberto Verdejo, and Narciso Martí-Oliet. Compositional Specification in Rewriting Logic. *TPLP*, 20:44–98, 2020.
12. José Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *TCS*, 96(1):73–155, 1992.
13. José Meseguer. Membership Algebra as a Logical Framework for Equational Specification. In Francesco Parisi-Presicce, editor, *WADT'97 Selected Papers*, volume 1376 of *LNCS*, pages 18–61. Springer, 1997.
14. José Meseguer. The Temporal Logic of Rewriting: A Gentle Introduction. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 354–382. Springer, 2008.
15. José Meseguer, Miguel Palomino, and Narciso Martí-Oliet. Equational Abstractions. *TCS*, 403(2-3):239–264, 2008.
16. Amir Pnueli. In Transition From Global to Modular Temporal Reasoning about Programs. In *Logics and Models of Concurrent Systems*, pages 123–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985.

Variants in the Infinitary Unification Wonderland

José Meseguer

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

Abstract. So far, results about variants, the finite variant property (FVP), and variant unification have been developed for equational theories $E \cup B$ where B is a set of axioms having a finitary unification algorithm, and the equations E, oriented as rewrite rules $\vec{E}$, are convergent modulo B. The extension to the case when B has an infinitary unification algorithm, for example because of non-commutative symbols having associative axioms, seems undeveloped. This paper takes a first step in developing such an extension. In particular, the relationships between the FVP and the boundedness properties, the identification of conditions on $E \cup B$ ensuring FVP, and the effective computation of variants and variant unifiers are explored in detail. The extension from the finitary to the infinitary case includes both surprises and opportunities.

1 Introduction

The notions of variant and variant unifier, e.g., [5, 14], have many applications, for example to cryptographic protocol analysis, e.g., [5, 3, 13, 19], program termination [10], SMT solving, e.g., [22, 24], program transformation and symbolic model checking, e.g., [23], and theorem proving, e.g., [25]. There is, however, an important current limitation, namely, that known results about variants, the finite variant property (FVP), and variant unification have been developed for equational theories $E \cup B$ where B is a set of axioms having a finitary unification algorithm, and the equations E, oriented as rewrite rules $\vec{E}$, are convergent modulo B. This leaves out of the picture many applications involving specifications whose axioms B have an infinitary unification algorithm, for example because of non-commutative symbols having associative axioms. For example, important rule-based languages such as OBJ [17], ASF+SDF [7], Elan [1], Cafe-OBJ [15], and Maude [4], all support rewriting modulo associativity without commutativity as well as other axioms, so that proving properties about such rule-based programs with the help of variant-based techniques would be very useful, but is currently out of reach, because the extension of the FVP results and algorithms to the infinitary unification case has not yet been developed and is a *terra incognita*.

Paradoxically, in this case practice has run ahead of theory, since Maude 3 [8] already supports the computation of variants and variant unifiers also in the case of axioms B that can include associativity without commutativity. The problem, however, is that at present we do not have any criteria to know when a theory involving such axioms is FVP. This paper is all about extending all the results and algorithms about variants, the finite variant property (FVP), and variant unification to the infinitary unification case. This includes investigating in the infinitary unification case: (i) adequate criteria for a theory to be FVP; (ii) the exact relationship between FVP and the so-called boundedness property [5, 3, 2]; (iii) notions weaker than FVP, and (iv) variant unification algorithms for both FVP theories and theories enjoying weaker properties.

As usual in these cases, the extension from the finitary to the infinitary case includes both some interesting surprises (thus, the poetic license of talking about a "Wonderland"), and

also some important opportunities. In a usual situation, I would now proceed to spell out in more detail the technical content and contributions of the paper. However, since for me the investigation of these problems has been quite full of surprises, I fear that I would spoil the fun for you, the reader, by telling you more details at this point. What I have already said is enough to give you the gist of the paper without depriving you of the fun ahead. Enjoy!

2 Preliminaries

2.1 Background on Order-Sorted Algebra

I summarize the order-sorted algebra notions needed in the paper. The material, adapted from [21], extends ideas in [16]. It assumes the notions of many-sorted signature and many-sorted algebra, e.g., [11], which include unsorted signatures and algebras as a special case.

Definition 1. *An* order-sorted (OS) signature *is a triple* $\Sigma = ((S,\leqslant),\Sigma)$ *with* $(S,\leqslant)$ *a poset and* (S,Σ) *a many-sorted signature.* $\widehat{S} = S/\equiv_{\leqslant}$*, the quotient of* S *under the equivalence relation* $\equiv_{\leqslant} = (\leqslant \cup \geqslant)^+$*, is called the set of* connected components *of* $(S,\leqslant)$*. The order* $\leqslant$ *and equivalence* $\equiv_{\leqslant}$ *are extended to sequences of same length in the usual way, e.g.,* $s'_1...s'_n \leqslant s_1...s_n$ *iff* $s'_i \leqslant s_i$*,* $1 \leqslant i \leqslant n$*.* Σ *is called* sensible *if for any two* $f : w \to s, f : w' \to s' \in \Sigma$*, with* w *and* w' *of same length, we have* $w \equiv_{\leqslant} w' \Rightarrow s \equiv_{\leqslant} s'$*.* A many-sorted signature Σ *is the special case where the poset* $(S,\leqslant)$ *is discrete, i.e.,* $s \leqslant s'$ *iff* $s = s'$*.* $\Sigma = ((S,\leqslant),\Sigma)$ *is a* subsignature *of* $\Sigma' = ((S',\leqslant'),\Sigma')$*, denoted* $\Sigma \subseteq \Sigma'$*, iff* $S \subseteq S'$*,* $\leqslant \subseteq \leqslant'$*, and* $\Sigma \subseteq \Sigma'$*.*

For connected components $[s_1],...,[s_n],[s] \in \widehat{S}$

$$f^{[s_1]...[s_n]}_{[s]} = \{f : s'_1...s'_n \to s' \in \Sigma \mid s'_i \in [s_i],\ 1 \leqslant i \leqslant n, s' \in [s]\}$$

denotes the family of "subsort polymorphic" operators f*.*

I will always assume that Σ's poset of sorts $(S,\leqslant)$ is *locally finite*, that is, that for any $s \in S$ its connected component $[s]$ is a finite set.

Definition 2. *For* $\Sigma = (S,\leqslant,\Sigma)$ *an OS signature, an* order-sorted Σ-algebra A *is a many-sorted* (S,Σ)*-algebra* A *such that:*
- *whenever* $s \leqslant s'$*, then we have* $A_s \subseteq A_{s'}$*, and*
- *whenever* $f : w \to s, f : w' \to s' \in f^{[s_1]...[s_n]}_{[s]}$ *and* $\overline{a} \in A^w \cap A^{w'}$*, then we have* $A_{f:w\to s}(\overline{a}) = A_{f:w'\to s'}(\overline{a})$*, where* $A^{\epsilon} = 1$ *(*ϵ *denotes the empty string and* $1 = \{0\}$ *is a singleton set), and* $A^{s_1...s_n} = A_{s_1} \times ... \times A_{s_n}$*.*

An order-sorted Σ-homomorphism $h : A \to B$ *is a many-sorted* (S,Σ)*-homomorphism such that whenever* $[s] = [s']$ *and* $a \in A_s \cap A_{s'}$*, then we have* $h_s(a) = h_{s'}(a)$*. We call* h injective*, resp.* surjective*, resp.* bijective*, iff for each* $s \in S$ h_s *is injective, resp. surjective, resp. bijective. We call* h *an* isomorphism *if there is another order-sorted* Σ*-homomorphism* $g : B \to A$ *such that for each* $s \in S$*,* $h_s;g_s = 1_{A_s}$*, and* $g_s;h_s = 1_{B_s}$*, with* $1_{A_s}, 1_{B_s}$ *the identity functions on* A_s, B_s*. This defines a category* $\mathbf{OSAlg}_{\Sigma}$*.*

Theorem 1. *[21] The category* $\mathbf{OSAlg}_{\Sigma}$ *has an initial algebra. Furthermore, if* Σ *is sensible, then the term algebra* T_{Σ} *with:*
- *if* $a : \epsilon \to s$ *then* $a \in T_{\Sigma,s}$*,*
- *if* $t \in T_{\Sigma,s}$ *and* $s \leqslant s'$ *then* $t \in T_{\Sigma,s'}$*,*
- *if* $f : s_1...s_n \to s$ *and* $t_i \in T_{\Sigma,s_i}$ $1 \leqslant i \leqslant n$*, then* $f(t_1,...,t_n) \in T_{\Sigma,s}$*,*

is initial, i.e., there is a unique Σ*-homomorphism from* T_{Σ} *to each* Σ*-algebra.*

T_Σ will (ambiguously) denote both the above-defined S-sorted set and the set $T_\Sigma = \bigcup_{s \in S} T_{\Sigma,s}$. For $[s] \in \widehat{S}$, $T_{\Sigma,[s]} = \bigcup_{s' \in [s]} T_{\Sigma,s'}$. An OS signature Σ is said to *have non-empty sorts* iff for each $s \in S$, $T_{\Sigma,s} \neq \emptyset$. Unless explicitly stated otherwise, *I will assume throughout that Σ has non-empty sorts*. An OS signature Σ is called *preregular* [16] iff for each $t \in T_\Sigma$ the set $\{s \in S \mid t \in T_{\Sigma,s}\}$ has a least element, denoted $ls(t)$. *I will assume throughout that Σ is preregular*.

An S-sorted set $X = \{X_s\}_{s \in S}$ of *variables*, satisfies $s \neq s' \Rightarrow X_s \cap X_{s'} = \emptyset$, and the variables in X are always assumed disjoint from all constants in Σ. The *Σ-term algebra* on variables X, $T_\Sigma(X)$, is the *initial algebra* for the signature $\Sigma(X)$ obtained by adding to Σ the variables X *as extra constants*. Since a $\Sigma(X)$-algebra is just a pair (A,α), with A a Σ-algebra, and α an *interpretation of the constants* in X, i.e., an S-sorted function $\alpha \in [X \to A]$, the $\Sigma(X)$-initiality of $T_\Sigma(X)$ can be expressed as the following corollary of Theorem 1:

Theorem 2. *(Freeness Theorem). If Σ is sensible, for each $A \in \mathbf{OSAlg}_\Sigma$ and $\alpha \in [X \to A]$, there exists a unique Σ-homomorphism, $\_\alpha : T_\Sigma(X) \to A$ extending α, i.e., such that for each $s \in S$ and $x \in X_s$ we have $x\alpha_s = \alpha_s(x)$.*

In particular, when $A = T_\Sigma(X)$, an interpretation of the constants in X, i.e., an S-sorted function $\sigma \in [X \to T_\Sigma(X)]$ is called a *substitution*, and its unique homomorphic extension $\_\sigma : T_\Sigma(X) \to T_\Sigma(X)$ is also called a substitution. Define $dom(\sigma) = \{x \in X \mid x \neq x\sigma\}$, and $ran(\sigma) = \bigcup_{x \in dom(\sigma)} vars(x\sigma)$.

The first-order language of *equational Σ-formulas* is defined in the usual way: its atoms are *Σ-equations* $t = t'$, where $t,t' \in T_\Sigma(X)_{[s]}$ for some $[s] \in \widehat{S}$ and each X_s is assumed countably infinite. The set $Form(\Sigma)$ of *equational Σ-formulas* is then inductively built from atoms by: conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and universal ($\forall x : s$) and existential ($\exists x : s$) quantification with sorted variables $x : s \in X_s$ for some $s \in S$. The literal $\neg(t = t')$ is denoted $t \neq t'$.

Given a Σ-algebra A, a formula $\varphi \in Form(\Sigma)$, and an assignment $\alpha \in [Y \to A]$, with $Y = fvars(\varphi)$ the free variables of φ, the *satisfaction relation* $A,\alpha \models \varphi$ is defined inductively as usual. Likewise, $A \models \varphi$ holds iff $A,\alpha \models \varphi$ holds for each $\alpha \in [Y \to A]$, where $Y = fvars(\varphi)$. We say that φ is *valid* (or *true*) in A iff $A \models \varphi$. We say that φ is *satisfiable* in A iff $\exists \alpha \in [Y \to A]$ such that $A,\alpha \models \varphi$, where $Y = fvars(\varphi)$. For a subsignature $\Omega \subseteq \Sigma$ and $A \in \mathbf{OSAlg}_\Sigma$, the *reduct* $A|_\Omega \in \mathbf{OSAlg}_\Omega$ agrees with A in the interpretation of all sorts and operations in Ω and discards everything in $\Sigma - \Omega$. If $\varphi \in Form(\Omega)$ we have the equivalence $A \models \varphi \Leftrightarrow A|_\Omega \models \varphi$.

An OS *equational theory* is a pair $T = (\Sigma,E)$, with E a set of Σ-equations. $\mathbf{OSAlg}_{(\Sigma,E)}$ denotes the full subcategory of $\mathbf{OSAlg}_\Sigma$ with objects those $A \in \mathbf{OSAlg}_\Sigma$ such that $A \models E$, called the *(Σ,E)-algebras*. $\mathbf{OSAlg}_{(\Sigma,E)}$ has an *initial algebra* $T_{\Sigma/E}$ [21]. Given $T = (\Sigma,E)$ and $\varphi \in Form(\Sigma)$, we call φ *T-valid*, written $E \models \varphi$, iff $A \models \varphi$ for each $A \in \mathbf{OSAlg}_{(\Sigma,E)}$. We call φ *T-satisfiable* iff there exists $A \in \mathbf{OSAlg}_{(\Sigma,E)}$ with φ satisfiable in A. Note that φ is *T-valid* iff $\neg\varphi$ is *T-unsatisfiable*.

The inference system in [21] is *sound and complete* for OS equational deduction, i.e., for any OS equational theory (Σ,E), and Σ-equation $u = v$ we have an equivalence $E \vdash u = v \Leftrightarrow E \models u = v$. Deducibility $E \vdash u = v$ is often abbreviated as $u =_E v$ and called *E-equality*. A preregular signature Σ is called *E-preregular* iff for each $u = v \in E$ and variable specialization ρ, $ls(u\rho) = ls(v\rho)$.

An *E-unifier* of a system of Σ-equations, i.e., of a conjunction $\phi = u_1 = v_1 \wedge \ldots \wedge u_n = v_n$ of Σ-equations, is a substitution σ such that $u_i\sigma =_E v_i\sigma$, $1 \leqslant i \leqslant n$. An *$E$-unification algorithm* for (Σ,E) is an algorithm generating a *complete set* of E-unifiers $Unif_E(\phi)$ for any system of Σ equations ϕ, where "complete" means that for any E-unifier σ of ϕ there is a $\tau \in Unif_E(\phi)$ and a substitution ρ such that $\sigma =_E (\tau\rho)|_{dom(\sigma) \cup dom(\tau)}$, where $=_E$ here means that for any variable

x we have $x\sigma =_E x(\tau\rho)|_{dom(\sigma)\cup dom(\tau)}$. The algorithm is *finitary* if it always terminates with a *finite set* $Unif_E(\phi)$ for any ϕ.

Given a set of equations B used for deduction modulo B, a preregular OS signature Σ is called *B-preregular*[1] iff for each $u = v \in B$ and substitutions ρ, $ls(u\rho) = ls(v\rho)$.

2.2 Convergent Theories, Constructors and Narrowing

Given an order-sorted equational theory $\mathcal{E} = (\Sigma, E \cup B)$, where B is a collection of associativity and/or commutativity and/or identity axioms and Σ is B-preregular, we can associate to it a corresponding *rewrite theory* [20] $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ by orienting the equations E as left-to right rewrite rules. That is, each $(u = v) \in E$ is transformed into a rewrite rule $u \to v$. The main purpose of the rewrite theory $\vec{\mathcal{E}}$ is to reduce the complex bidirectional reasoning with equations to the much simpler unidirectional reasoning with rules under suitable assumptions. I assume familiarity with the notion of subterm $t|_p$ of t at a term position p and of term replacement $t[w]_p$ of $t|_p$ by w at position p (see, e.g., [6]). The rewrite relation $t \to_{\vec{E},B} t'$ (which can be abbreviated to $t \to_{\vec{\mathcal{E}}} t'$) holds iff there is a subterm $t|_p$ of t, a rule $(u \to v) \in \vec{E}$ and a substitution θ such that $u\theta =_B t|_p$, and $t' = t[v\theta]_p$. We denote by $\to^*_{\vec{E},B}$ the reflexive-transitive closure of $\to_{\vec{E},B}$. The requirements on $\vec{\mathcal{E}}$ allowing us to reduce equational reasoning to rewriting are the following: (i) $vars(v) \subseteq vars(u)$; (ii) *sort-decreasingness*: for each substitution θ we must have $ls(u\theta) \geqslant ls(v\theta)$; (iii) *strict B-coherence*: if $t_1 \to_{\vec{E},B} t'_1$ and $t_1 =_B t_2$ then there exists $t_2 \to_{\vec{E},B} t'_2$ with $t'_1 =_B t'_2$; (iv) *confluence*: for each term t if $t \to^*_{\vec{E},B} v_1$ and $t \to^*_{\vec{E},B} v_2$, then there exist rewrite sequences $v_1 \to^*_{\vec{E},B} w_1$ and $v_2 \to^*_{\vec{E},B} w_2$ such that $w_1 =_B w_2$; (v) *termination*: the relation $\to_{\vec{E},B}$ is well-founded. If $\vec{\mathcal{E}}$ satisfies conditions (i)–(v) then it is called *convergent*. The key point is that then, given a term t, all terminating rewrite sequences $t \to^*_{\vec{E},B} w$ end in a term w, denoted $t!_{\vec{\mathcal{E}}}$, that is unique up to B-equality, and its called t's *canonical form*. Three major results then follow for the ground convergent case: (1) (Church-Rosser Theorem) for any terms t,t' we have $t =_{E\cup B} t'$ iff $t!_{\vec{\mathcal{E}}} =_B t'!_{\vec{\mathcal{E}}}$, (2) the canonical forms of ground terms are the elements of the *canonical term algebra* $C_{\Sigma/E,B}$, where for each $f : s_1 \ldots s_n \to s$ in Σ and canonical terms $t_1 \ldots t_n$ with $ls(t_i) \leqslant s_i$ the operation $f_{C_{\Sigma/E,B}}$ is defined by the identity: $f_{C_{\Sigma/E,B}}(t_1 \ldots t_n) = f(t_1 \ldots t_n)!_{\vec{\mathcal{E}}}$, and (3) we have an isomorphism $T_{\mathcal{E}} \cong C_{\Sigma/E,B}$.

Given a convergent rewrite theory $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ and a subsignature Ω on the same poset of sorts, a *constructor subspecification* is a convergent rewrite subtheory $\vec{\mathcal{E}}_\Omega = (\Omega, B_\Omega, \vec{E}_\Omega)$ of $\vec{\mathcal{E}}$ (i.e., we have an inclusion of convergent theories $(\Omega, B_\Omega, \vec{E}_\Omega) \subseteq (\Sigma, B, \vec{E})$) such that: (i) for each ground term t, $t!_{\vec{\mathcal{E}}} \in T_\Omega$, and (ii) $T_{\mathcal{E}}|_\Omega \cong T_{\mathcal{E}_\Omega}$. Furthermore, if $E_\Omega = \emptyset$, Ω is then called a signature of *free constructors modulo axioms* B_Ω. Furthermore, if $\vec{\mathcal{E}}_\Omega \subseteq \vec{\mathcal{E}}$ is a constructor subspecification we say that $\vec{\mathcal{E}}$ *sufficiently complete* w.r.t. Ω.

Whenever we have an inclusion of convergent theories $\vec{\mathcal{E}} \subseteq \vec{\mathcal{E}}'$, with respective signatures Σ and Σ', such that $T_{\mathcal{E}'}|_\Sigma \cong T_{\mathcal{E}}$, we say that $\vec{\mathcal{E}}'$ *protects* $\vec{\mathcal{E}}$. Therefore, condition (ii) above just states that $\vec{\mathcal{E}}$ protects $\vec{\mathcal{E}}_\Omega$.

Narrowing. Given a convergent $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ such that B has a (not necessarily finitary) unification algorithm, by replacing B-matching by B-unification, we can generalize the rewrite

[1] If $B = B_0 \uplus U$, with B_0 associativity and/or commutativity axioms, and U identity axioms, the B-preregularity notion can be *broadened* by requiring only that: (i) Σ is B_0-preregular in the standard sense that $ls(u\rho) = ls(v\rho)$ for all $u = v \in B_0$ and substitutions ρ; and (ii) the axioms U oriented as rules $\vec{U}$ are *sort-decreasing* in the sense explained in Section 2.2.

relation $t \rightarrow_{\vec{\mathcal{E}}} t'$ to the *narrowing relation* $t \rightsquigarrow_{\vec{\mathcal{E}}} t'$, often decorated as $t \overset{\theta}{\rightsquigarrow}_{\vec{\mathcal{E}}} t'$, which holds between Σ-terms t and t' iff there is a *non-variable* position p in t, a rewrite rule $(l \rightarrow r) \in \vec{E}$ (renamed if necessary so as not to share variables with t), and a B-unifier $\theta \in Unif_B(l = t|_p)$, with $ran(\theta)$ all fresh new variables "standardized apart," i.e., never generated before in the same narrowing process, such that $t' = (t[r]_p)\theta$. Likewise, in $t \overset{\theta}{\rightsquigarrow}{}^*_{\vec{\mathcal{E}}} t'$, the relation $\rightsquigarrow^*_{\vec{\mathcal{E}}}$ denotes the reflexive-transtive closure of $\rightsquigarrow_{\vec{\mathcal{E}}}$, and θ denotes the composition $\theta = \theta_1 ... \theta_n$ of the substitutions appearing in the n steps of the narrowing sequence, or the identity substitution when the sequence has length 0. What narrowing a term t with $\rightsquigarrow_{\vec{\mathcal{E}}}$, means is to *symbolically execute* t in the theory $\vec{\mathcal{E}}$. That is, even though t may be in $\vec{\mathcal{E}}$-canonical form and therefore not be executable by rewriting with $\rightarrow_{\vec{\mathcal{E}}}$, by instantiating t with $\rightsquigarrow_{\vec{\mathcal{E}}}$ in all possible "most general" ways, it *becomes* executable. Specifically, if $t \overset{\theta}{\rightsquigarrow}{}^*_{\vec{\mathcal{E}}} t'$ holds, then $t\theta \rightarrow^*_{\vec{\mathcal{E}}} t'$ also holds and, conversely, (the so-called "Lifting Lemma" [18]), if γ is an $\vec{\mathcal{E}}$-canonical substitution and $t\gamma \rightarrow^*_{\vec{\mathcal{E}}} u$ holds, then there is a narrowing sequence $t \overset{\theta}{\rightsquigarrow}{}^*_{\vec{\mathcal{E}}} t'$ of same length as $t\gamma \rightarrow^*_{\vec{\mathcal{E}}} u$, and with same term positions and rules at each step, and a substitution δ such that $t'\delta =_B u$.

2.3 Variants in a Nutshell

Given a convergent rewrite theory $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ associated to an equational theory $\mathcal{E}$ a *variant* of a term t is a pair (u, θ) such that θ is a substitution in canonical form, i.e., $\theta = \theta!_{\vec{\mathcal{E}}}$, and $u =_B (t\theta)!_{\vec{\mathcal{E}}}$. $\vec{\mathcal{E}}$ has the *finite variant property* (FVP) iff for any term t there is a finite set $[\![t]\!]_{\vec{\mathcal{E}}} = \{(u_1, \theta_1), ..., (u_n, \theta_n)\}$ of variants of t that are *most general possible* among all such variants, where the "most general" relation $\sqsupseteq$ between variants is defined by the equivalence: $(v, \alpha) \sqsupseteq (w, \beta) \Leftrightarrow \exists \gamma (\beta =_B \alpha\gamma \wedge v\gamma =_B w)$. Furthermore, if B has a finitary unification algorithm, a finite set of most general variants in $[\![t]\!]_{\vec{\mathcal{E}}}$ can be effectively computed for any t by *folding variant narrowing* [14].

There are two ways to understand variants and FVP. One is in terms of the notion of a *pattern*, i.e., a term u with variables describing *something*. The variants $\{(u_1, \theta_1), ..., (u_n, \theta_n)\}$ of a term t in an FVP theory $\vec{\mathcal{E}}$ are clearly patterns $u_1, ..., u_n$ (plus the added technical monkeys $\theta_1, ..., \theta_n$). But what do they describe? Obviously, up to B-equality, the infinite set of *all* patterns of the form $(t\theta)!_{\vec{\mathcal{E}}}$ for a given term t. But there is a second, equivalent way of understanding the FVP notion. Implicit in the idea that for *any* variant (w, β) of t there is a most general one $(u_1, \theta_1) \in [\![t]\!]_{\vec{\mathcal{E}}}$ lies the property, called the *boundedness property* [5], that *all variants* of a term t can be computed in a finite number of rewriting steps smaller or equal to a *fixed bound* $bd(t)$ depending on t. Why so? Because: (1) we can choose $bd(t)$ to be the maximum of the smallest rewriting depths needed to compute $(t\theta_i)!_{\vec{\mathcal{E}}} = u_i$ for $1 \leqslant i \leqslant n$, and (2) since for each variant (w, β) of t there is a $(u_i, \theta_i) \in [\![t]\!]_{\vec{\mathcal{E}}}$ such that $(u_i, \theta_i) \sqsupseteq (w, \beta) \Leftrightarrow \exists \gamma (\beta =_B \theta_i\gamma \wedge u_i\gamma =_B w)$, by the fact that the rewrite relation is B-coherent, we can obtain a rewrite sequence from $t\beta$ to $(t\beta)!_{\vec{\mathcal{E}}} =_B w$ of length l with $l \leqslant bd(t)$ just by instantiating by γ the sequence of length l from $t\theta_i$ to $(t\theta_i)!_{\vec{\mathcal{E}}} =_B u_i$.

In fact we have the following equivalence (see [5], and for a more precise statement, [2])

Theorem 3. *(FVP iff Boundedness). Give a convergent $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ associated to an equational theory $\mathcal{E}$ such that B has a finitary unification algorithm, $\vec{\mathcal{E}}$ is FVP iff $\vec{\mathcal{E}}$ has the boundedness property.*

Furthermore, since for *any* convergent $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ such that B has a finitary unification algorithm a complete (but not necessarily finite) set of $\vec{\mathcal{E}}$-variants of a term t can be effectively generated by folding variant narrowing [14]. Furthermore, as proved in [2] (and mechanized in Maude), there is a simple semi-decision procedure to check whether such a convergent $\vec{\mathcal{E}}$ is FVP: it is so if and only if for each $f \in \Sigma$ the term $f(x_1,...,x_n)$, with variables $x_1,...,x_n$ having most general possible sorts, has a finite number of most general $\vec{\mathcal{E}}$-variants.

Another key point about $\vec{\mathcal{E}}$ being FVP is that, as proved in [14], if B has a finitary unification algorithm, then $E \cup B$ has also a finitary unification algorithm called *variant unification*. If $\vec{\mathcal{E}}$ is FVP and it furthermore has a constructor subspecification $\vec{\mathcal{E}}_\Omega \subseteq \vec{\mathcal{E}}$, say with $\vec{\mathcal{E}}_\Omega = (\Omega, B_\Omega, \vec{E}_\Omega)$, then several more notions appear: (1) A *constructor variant* of t is a variant (u,θ) of t such that u is an Ω-term. The set of constructor variants of t is denoted $[\![t]\!]^{\Omega}_{\vec{\mathcal{E}}}$. (2) A *constructor unifier* of a system of Σ-equations $u_1 = v_1 \wedge ... \wedge u_n = v_n$ is a $E \cup B$-unifier α such that for $1 \leqslant i \leqslant n$, $u_i\alpha!_{\vec{\mathcal{E}}}$ is an Ω-term. As explained in [22, 24], under very mild conditions on the constructor subspecification $\vec{\mathcal{E}}_\Omega \subseteq \vec{\mathcal{E}}$, if $\vec{\mathcal{E}}$ is FVP and B and B_Ω have finitary unification algorithms, there is an effectively computable finite subset $\{(u_1,\theta_1),...,(u_n,\theta_n)\}$ of most general constructor variants in $[\![t]\!]^{\Omega}_{\vec{\mathcal{E}}}$. Furthermore, under the same assumptions, for any system of Σ-equations $\phi \equiv u_1 = v_1 \wedge ... \wedge u_n = v_n$ there is an algorithm computing a finite set $Unif^{\Omega}_{E \cup B}(\phi)$ of most general constructor unifiers. In particular, since any ground substitution $\rho : Y \to T_\Sigma$ is $E \cup B$-equivalent to the ground constructor substitution $\rho!_{\vec{\mathcal{E}}}$, any ground unifier of ϕ is $E \cup B$-equivalent to a ground constructor unifier that is an instance up to $E \cup B$-equality of a constructor unifier in $Unif^{\Omega}_{E \cup B}(\phi)$.

Note that all the results and algorithms known so far about variants depend crucially on the assumption that the convergent theory $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ is such that B has a finitary unification algorithm. The notion of variant, however, is more general and has been defined without the finitary unification assumption. But what happens when we weaken this assumption and assume instead that B just *has* a (not necessarily finitary) unification algorithm? For example, B may contain an associativity axiom for some non-commutative symbol f, and then the B-unification algorithm will be infinitary. What happens to variants in such an infinitary wonderland?

3 Variants in the Infinitary Unification Wonderland

To explore variants in the infinitary unification wonderland, let us begin with an example.

Example 1. Consider a 2-sorted equational theory **St** of strings, whose signature Σ has sorts *St* and *Pred* and operators: (i) constants *a*,*b* of sort *St*, (ii) a constant *tt* of sort *Pred*, (iii) a string concatenation operator $\_\_ : St\ St \to St$, and (iv) an equality predicate $\_ \equiv \_ : St\ St \to Pred$; and with just two equations: the associativity axiom for string concatenation $(xy)z = x(yz)$, which we denote by A, and the equality predicate definition $x \equiv x = tt$, denoted E. By orienting the equality definition as a singleton rule set $\vec{E}$, we get an obviously convergent rewrite theory $\vec{\mathbf{St}} = (\Sigma, A, \vec{E})$. Furthermore, $\vec{\mathbf{St}}$ enjoys the following two nice properties: (1) **Boundedness**, since all terms of sort *St* are in $\vec{\mathbf{St}}$-canonical form, and any term of sort *Pred not* in $\vec{\mathbf{St}}$-canonical form must be of the form $u \equiv v$ with $u =_A v$, and can therefore be put in the canonical form *tt* in *one* rewrite step. (2) **Finite Variants for all Function Symbols**. Each $f \in \Sigma$ is such that $f(x_1,...,x_n)$ is either a constant (n=0) in canonical form, or the string expression $x_1\ x_2$ which is in canonical form and has the single most general variant $(x_1\ x_2, id)$, with *id* the identity substitution, or is the equality expression $x_1 \equiv x_2$, and when we request its variants form its

Maude specification we get two of them, namely, $(x_1 \equiv x_2, id)$, and $(tt, \{x_1 \mapsto x_3, x_2 \mapsto x_3\})$. *Ergo*, by Theorem 3, $\vec{\mathbf{St}}$ must be FVP! Right? No, this is wrong. Consider the term $a\,x \equiv x\,a$. When we request its variants from Maude (where it is written as `a x ~ x a`) we get:

```
Variant #1
rewrites: 0 in 0ms cpu (3ms real) (0 rewrites/second)
Pred: (a #1:St) ~ #1:St a
x --> #1:St
Warning: Unification modulo the theory of operator __
has encountered an instance for which it may not be complete.

Variant #2
rewrites: 1 in 1ms cpu (8ms real) (660 rewrites/second)
Pred: tt
x --> a

No more variants.
rewrites: 1 in 1ms cpu (8ms real) (647 rewrites/second)
Warning: Some variants may have been missed due to incomplete
unification algorithm(s).
```

That is, up to renaming of variable we get two variants: $(a\,x \equiv x\,a, id)$, and $(tt, \{x \mapsto a\})$, *plus* a serious warning: since, to cope in a practical manner with the infinitary nature of associative unification, the Maude unification algorithm for associativity is incomplete, for a unification problem having an infinite number of solutions, only a finite number of them are returned. Here, when trying to narrow the term $a\,x \equiv x\,a$ with the rule $x \equiv x \rightarrow tt$, solving the A-unification problem $(a\,x \equiv x\,a) = (y \equiv y)$ reduces to solving the A-unification problem $a\,x = x\,a$. But this problem does *not* have a finite set of solutions. It has instead the infinite set of solutions $\{x = a, x = a\,a, x = a\,a\,a, ..., x = a \overset{n}{\ldots} a, ...\}$. Therefore, the term $a\,x \equiv x\,a$ has the following *infinite* set of variants: $(a\,x \equiv x\,a, id), (tt, \{x \mapsto a\}), ..., (tt, \{x \mapsto a \overset{n}{\ldots} a,\}), ...$, none of which are comparable with each other in the $\sqsupseteq$ relation. *Ergo*, Theorem 3 fails in general for a convergent $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ where the B-unification algorithm is not finitary. In particular, $\vec{\mathbf{St}}$ has the boundedness property but is *not* FVP.

Given this, somewhat perplexing situation, we can ask two questions: (a) What do we *know* at this point? (b) How can (1) and (2) hold for $\vec{\mathbf{St}}$ and, yet, $\vec{\mathbf{St}}$ is not FVP? Question (a) can be given two answers. First of all, we know that if $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ is FVP, then, as explained in Section 2.2, the boundedness property follows from the FVP property, and is therefore enjoyed by $\vec{\mathcal{E}}$. Second, we also know:

Theorem 4. *(Boundedness Sufficient Condition). If a convergent theory $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$, (1) has a B-unification algorithm, and (2) (Σ-boundedness) for each $f \in \Sigma$ the term $f(x_1, ..., x_n)$ with most general possible variables $x_1, ..., x_n$ has a finite number of most general $\vec{\mathcal{E}}$-variants, then $\vec{\mathcal{E}}$ has the boundedness property.*

The proof is indeed the same structural induction proof as in Theorem 7 of [2], with Theorem 7 suitably rephrased by means of Theorem 6 in [2].

Question (b) can be generalized to question (b'): How can a convergent theory $\vec{\mathcal{E}}$ have the boundedness property, yet not be FVP? Question (b') can perhaps be best answered in narrowing terms: since for each term t we have a bound $bd(t)$ on the maximum of the minimal-length

terminating rewrite sequences starting at all its instances $t\gamma$ for any $\vec{\mathcal{E}}$-canonical γ, by the Lifting Lemma for $\vec{\mathcal{E}}$-narrowing (see Section 2.2), we know that we can compute a complete set of most general variants for t by bounding the $\vec{\mathcal{E}}$-narrowing tree of t at depth $bd(t)$, and then obtaining as most general variants the pairs (u,θ) such that $t \overset{\theta}{\leadsto}{}^{n}_{\vec{\mathcal{E}}} u$, $n \leqslant bd(t)$, $u = u!_{\vec{\mathcal{E}}}$, and $\theta = \theta!_{\vec{\mathcal{E}}}$. The problem, though, is that, as shown in Example 1, the bounded narrowing tree for t may have *infinite branching* at some of its nodes, so that the finiteness of such a set of most general variants (and therefore the FVP property) cannot be ensured in general. So, what can we do?

3.1 Finitely Branching Theories

To ensure that a convergent $\vec{\mathcal{E}}$ having the boundedness property is FVP, "all" we need to do is to make sure that the $\vec{\mathcal{E}}$-narrowing tree of any term t never has infinite branching. This can be essentially ensured in two slightly different ways:

Definition 3. *(FB/CFB Theories). A convergent theory $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ having a B-unification algorithm is called* finitely branching *(FB) (resp.* contextually finitely branching *(CFB) with respect to a theory $\vec{\mathcal{E}}'$ whose axioms B' are collapse-free (i.e., such that if $(u = v) \in B'$, then neither u nor v can be variables), iff for each $(l \to r) \in \vec{E}$ and any equality $l = w$ such that w is not a variable and $vars(l) \cap vars(w) = \emptyset$, the set $Unif_B(l = w)$ is finite (resp. $\vec{\mathcal{E}}$ is FB and, furthermore, any equality $l = w$ such that w is a term which is not a variable in the disjoint union $\vec{\mathcal{E}} \uplus \vec{\mathcal{E}}'$, (where all sorts and function symbols in $\vec{\mathcal{E}}$ and $\vec{\mathcal{E}}'$ have been made different, except for the possible identification of two sorts s in $\vec{\mathcal{E}}$ and s' in $\vec{\mathcal{E}}'$) and such that $vars(l) \cap vars(w) = \emptyset$, the set $Unif_{B \uplus B'}(l = w)$ is finite).*

which gives us:

Theorem 5. *(FVP whenever FB + Boundedness). An FB convergent theory $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ having a B-unification algorithm is FVP iff it has the boundedness property.*

Proof. The proof of the $(\Rightarrow)$ part follows from any FVP theory having the boundedness property. The proof of the $(\Leftarrow)$ part follows under the boundedness assumption from the already-described narrowing-based method to extract a complete set of most general variants of a term t from its $\vec{\mathcal{E}}$-narrowing tree bounded at depth $bd(t)$, which is a *finite* tree by the FB assumption. □

What algorithmic consequences can we derive from Theorem 5? The main one is a *business as usual* consequence: under the FB assumption, the *finite variant narrowing algorithm* specified and proved correct in [14] for the case where the axioms B of $\vec{\mathcal{E}}$ have a *finitary* B-unification algorithm applies and is correct *exactly as before* when the B-unification algorithm is infinitary, because in all the folding variant narrowing steps the B-unification problems for each narrowing step *are finitary* thanks to the FB property. A second practical consequence, indeed an obvious corollary of Theorem 5, is that, thanks to Theorem 4, to check that a convergent $\vec{\mathcal{E}}$ is FVP we just need three conditions: (i) it has a B-unification algorithm; (ii) satisfies the Σ-boundedness property; and (iii) is FB. Conditions (i) and (ii) are easily checkable. How about condition (iii)?

3.2 Checking FB and CFB

How can we *check* that a convergent theory is FB or CFB? And how *restrictive* are conditions ensuring FB/CFB in practice? The answers will of course depend on the axioms B in question. For many examples, and certainly for any example specifiable in Maude, the source of infinity in B-unification for a set B of associativity and/or commutativity and/or identity axioms will always come from some associative but not commutative symbol in the signature Σ. Let us call any such symbol an A-symbol, and shorten associative unification to A-unification. A useful general observation is that in (order-sorted) A-unification, there are classes of pure A-terms l, i.e., involving at most a single A-symbol, such that any pure A-unification problem $l = w$ where $vars(l) \cap vars(w) = \emptyset$, has always a finite set of A-solutions. For example, one such class is the class of *linear* A-terms (see, e.g., [12]). This class, and other similar such classes, can be exploited to ensure conditions such as FB.[2] For my present purposes, it is not necessary to develop here the technical details of how criteria such as linearity of A-terms can be used for checking FB. It will be enough for me to illustrate in some detail how properties such as linearity of rules involving in some way A-symbols can be used to check FB in a concrete example, namely, a data type of strings:

Example 2. (Strings Data Type). Consider the following algebraic data type of strings, whose rewrite theory $\vec{\mathbf{St}} = (\Sigma, A, \vec{E})$ has signature Σ with sorts X (where we understand X as a *parameter sort*, i.e., in Maude this would be the sort `Elt` of the `TRIV` parameter theory), *NeSt* and *St*, subsort inclusions $X < NeSt < St$, a constant ε of sort *St*, a subsort-overloaded string concatenation operator $\_\_ : St\,St \to St$, $\_\_ : NeSt\,NeSt \to NeSt$, two functions $first, last : NeSt \to X$, and two other functions $rest, prior : NeSt \to St$. The only axiom in A is the associativity axiom $(u\,v)\,w = u\,(v\,w)$, were u,v,w are variables of sort *St*, p,q,r variables of sort *NeSt*, and x is a variable of sort X. The rules $\vec{E}$ are: (i) the identity rules $u\,\varepsilon \to u$ and $\varepsilon\,u \to u$, (ii) the *first* and *rest* rules $first(x) \to x$, $first(x\,q) \to x$, $rest(x) \to \varepsilon$, $rest(x\,q) \to q$, and (iii) the *last* and *prior* rules $last(x) \to x$, $last(q\,x) \to x$, $prior(x) \to \varepsilon$, $prior(q\,x) \to q$. The theory $\vec{\mathbf{St}}$ is clearly terminating, since the associativity axiom is term-size-preserving, and all the rules are term-size-decreasing. It is also sort-decreasing and strictly A-coherent. The only critical pairs modulo A are those between the rules $u\,\varepsilon \to u$ and $\varepsilon\,u \to u$, which are clearly joinable, so $\vec{\mathbf{St}}$ is also confluent and therefore convergent. It satisfies also the Σ-boundedness property: for each non-constant symbol $f \in \Sigma$, the term $f(x_1,...,x_n)$ has exactly 3 most general variants.

We can naturally give to $\mathbf{St}$ the meaning of a *parameterized data type*, where X is the parameter sort, so that we can denote[3] $\mathbf{St}$ as $\mathbf{St}[X]$. $\mathbf{St}[X]$ then defines a *theory transformation* of the form $(\mathbf{M}, s) \mapsto \mathbf{St}[s]$. That is, given a convergent theory $\vec{\mathbf{M}}$ and a sort s in $\vec{\mathbf{M}}$, we obtain the theory $\vec{\mathbf{St}}[s]$ by first forming the *disjoint union* of $\vec{\mathbf{St}}[X]$ and $\vec{\mathbf{M}}$, and then identifying the sorts X and s. For example, for $\mathbf{M}$ a specification of the natural numbers with a sort *Nat*, then $\vec{\mathbf{St}}[Nat]$ would give us a data type of strings of natural numbers, and for $\mathbf{M}$ a specification of the booleans with a sort *Bool*, $\vec{\mathbf{St}}[Bool]$ would be a data type of bit strings. This suggest proving the FB property not just for $\vec{\mathbf{St}}[X]$, but for any instance $\vec{\mathbf{St}}[s]$; that is proving that $\vec{\mathbf{St}}[X]$ is CFB for all the context theories $\mathbf{M}$.

[2] Note that this class does not impose very strong restrictions on the rewrite rules of a convergent theory $\vec{\mathcal{E}}$ that we want to check FB: all we would need is something like an "A-linearity" condition on rules $l \to r$ in $\vec{E}$. Call a sort s an *A-sort* (resp. *A-reachable sort*) if it is the sort of an associative but non-commutative symbol (resp. if there is a term t of sort s having a variable whose sort is an A-sort). Call $l \to r$ *A-linear* iff any variable of l having an A-reachable sort appears only once in l.

[3] In Maude it would be denoted $\mathbf{St}\{X :: TRIV\}$. Its instantiation to sort s of module $\mathbf{M}$ would be denoted $\mathbf{St}\{s\}$. Here I follow the simple notation introduced in [22].

Theorem 6. *(CFB for $\vec{\mathbf{St}}[s]$). Assuming the axioms $B_{\mathbf{M}}$ of $\mathbf{M}$ are any combination of associativity and/or commutativity axioms,*[4] *the transformation $(\mathbf{M},s) \mapsto \mathbf{St}[s]$ preserves the CFB property. That is, if $\vec{\mathbf{M}}$ is CFB with respect to $\vec{\mathbf{St}}[X]$, then $\vec{\mathbf{St}}[X]$ is also CFB with respect to $\vec{\mathbf{M}}$, and therefore $\vec{\mathbf{St}}[s]$ is FB for any pair $(\mathbf{M},s)$.*

Let me also illustrate the breakdown of the FB property due to the presence of non-left-linear rewrite rules. Assume that we add to our $\vec{\mathbf{St}}$ module above the following sort, operator and equations: (1) a new sort *Pred* of predicates, (2) a constant *tt* of sort *Pred* and a string membership predicate $\_\in\_ : X\,St \to Pred$, and the definition of that predicate (in the positive case) by the rewrite rules: $x \in x \to tt$, $x \in x\,u \to tt$, $x \in u\,x \to tt$ and $x \in u\,x\,v \to tt$. The extended module thus obtained, let us denote it $\vec{\mathbf{St}}^{\in}$, is also terminating due to the same term-size decreasing nature of all rules, strictly A-coherent as before, sort decreasing, and also confluent, since all the new critical pairs associated to the string membership rules are joinable to the constant *tt*. Furthermore, since the term $x \in u$ has exactly 5 variants, the module is Σ-bounded and therefore, by Theorem 4 satisfies the Boundedness property. However, $\vec{\mathbf{St}}^{\in}$ is not FVP. Here is a counterexample in Maude, when we ask for the variants of the term $first(q\,q') \in first(q'\,q)$:

```
Variant #1
rewrites: 0 in 0ms cpu (0ms real) (0 rewrites/second)
Pred: first(#1:NeSt #2:NeSt) in first(#2:NeSt #1:NeSt)
q --> #1:NeSt
q' --> #2:NeSt
Warning: Unification modulo the theory of operator __ has encountered an
    instance for which it may not be complete.

Variant #2
rewrites: 5 in 3ms cpu (17ms real) (1516 rewrites/second)
Pred: tt
q --> %1:NeSt
q' --> %1:NeSt

Variant #3
rewrites: 5 in 3ms cpu (17ms real) (1492 rewrites/second)
Pred: %1:X in first(%3:NeSt %1:X %2:NeSt)
q --> %1:X %2:NeSt
q' --> %3:NeSt

Variant #4
rewrites: 5 in 3ms cpu (17ms real) (1470 rewrites/second)
Pred: %1:X in first(%2:NeSt %1:X)
q --> %1:X
q' --> %2:NeSt

Variant #5
rewrites: 5 in 3ms cpu (17ms real) (1458 rewrites/second)
```

[4] I am purposefully avoiding identity axioms for two reasons: (1) because, thanks to the theory transformation $\vec{\mathcal{E}} \mapsto \vec{\mathcal{E}}_U$ in [10] mapping a convergent $\vec{\mathcal{E}}$ with identity axioms U into a semantically equivalent convergent $\vec{\mathcal{E}}_U$ where such axioms have been transformed into rewrite rules, this involves no real loss of generality; and (2) because we need "collapse-free" axioms in the CFB property.

```
Pred: first(%3:NeSt %1:X %2:NeSt) in %1:X
q --> %3:NeSt
q' --> %1:X %2:NeSt

Variant #6
rewrites: 5 in 3ms cpu (17ms real) (1446 rewrites/second)
Pred: first(%2:NeSt %1:X) in %1:X
q --> %2:NeSt
q' --> %1:X

Variant #7
rewrites: 13 in 6ms cpu (23ms real) (1985 rewrites/second)
Pred: #3:X in #1:X
q --> #3:X #4:NeSt
q' --> #1:X #2:NeSt

Variant #8
rewrites: 13 in 6ms cpu (24ms real) (1973 rewrites/second)
Pred: #2:X in #1:X
q --> #2:X #3:NeSt
q' --> #1:X

Variant #9
rewrites: 13 in 6ms cpu (24ms real) (1965 rewrites/second)
Pred: #3:X in #1:X
q --> #3:X
q' --> #1:X #2:NeSt

Variant #10
rewrites: 13 in 6ms cpu (24ms real) (1957 rewrites/second)
Pred: #2:X in #1:X
q --> #2:X
q' --> #1:X

Variant #11
rewrites: 17 in 7ms cpu (25ms real) (2282 rewrites/second)
Pred: tt
q --> %1:X %2:NeSt
q' --> %1:X %3:NeSt

Variant #12
rewrites: 17 in 7ms cpu (25ms real) (2266 rewrites/second)
Pred: tt
q --> %1:X %2:NeSt
q' --> %1:X

Variant #13
rewrites: 17 in 7ms cpu (25ms real) (2251 rewrites/second)
Pred: tt
q --> %1:X
q' --> %1:X %2:NeSt
```

```
No more variants.
rewrites: 17 in 7ms cpu (25ms real) (2232 rewrites/second)
Warning: Some variants may have been missed due to incomplete unification
    algorithm(s).
```

The breakdown of the FB property is manifested in this example by the fact that, when trying to narrow the term $first(q\ q') \in first(q'\ q)$ with the rule $x \in x \rightarrow tt$, the unification problem $(x \in x) = (first(q\ q') \in first(q'\ q))$ is equivalent to the system of equations $x = first(q\,q') \wedge x = first(q'\,q)$, which is equivalent to the equation $first(q\,q') = first(q'\,q)$, which, in turn, is equivalent to the A-equation $q\,q' = q'\,q$, which has an infinite number of solutions.

3.3 Variant Unification

Suppose that $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ is a convergent FB theory having a B-unification algorithm and that we have checked that it has the boundedness property, for example by checking that it has the Σ-boundedness property. Then, by Theorem 5 $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ is FVP. So we should be able to obtain an $E \cup B$-unification algorithm by folding variant narrowing [14], right? Yes, but with a twist. Suppose that the B-unification algorithm is infinitary. Then, $E \cup B$-unification will *a fortiori* be infinitary. For example, even though the convergent $\vec{\mathbf{St}}$ module enjoys the Σ-boundedness property and is FB and therefore FVP, since the terms $q\,q'$ and $q'\,q$ cannot be narrowed at all by the rules in $\vec{\mathbf{St}}$, folding variant narrowing reduces the $E \cup A$-unification problem to the A-unification problem: $q\,q' = q'\,q$, which has an infinite number of solutions. That is, folding variant narrowing *does* indeed provide an $E \cup B$-unification algorithm for any FVP theory, even when B-unification is infinitary; but of course $E \cup B$-unification is also infinitary. The key observation, however, from an algorithmic point of view, is that the *folding variant narrowing* part of the variant $E \cup B$-unification algorithm is *finite* for any unification problem; but in the last step, when each unification problem obtained by exploring the finite $\vec{\mathcal{E}}$-narrowing tree for the problem is transformed into a B-unification problem we arrive at the unavoidable infinitary nature of the algorithm. Let me further clarify this point by reducing variant $E \cup A$-unification for a single equation in our $\vec{\mathbf{St}}$ module to folding variant narrowing of single terms in the transformed module $\vec{\mathbf{St}}^{\equiv}$ obtained by adding: (1) a new sort *Pred* of predicates, (2) a constant *tt* of sort *Pred* an equality predicate $\_\equiv\_ : St\ St \rightarrow Pred$, and the equality rewrite rule $u \equiv u \rightarrow tt$. The module $\vec{\mathbf{St}}^{\equiv}$ thus obtained is still convergent. Furthermore, since the term $u \equiv v$ has exactly two variants, it enjoys the boundedness property. Furthermore, any $E \cup A$-unification problem $t = t'$ in $\vec{\mathbf{St}}$ can be reduced to a folding variant narrowing problem in $\vec{\mathbf{St}}^{\equiv}$, namely, the variant $E \cup A$-unifiers of $t = t'$ are exactly the substitutions $\theta = \theta_1 ... \theta_{n+1}$ associated to variant narrowing sequences in $\vec{\mathbf{St}}^{\equiv}$ of the term $t \equiv t'$ of the form:

$$t \equiv t' \overset{\theta_1}{\leadsto}_{\vec{\mathbf{St}}} t_1 \equiv t'_1 ... t_{n-1} \equiv t'_{n-1} \overset{\theta_n}{\leadsto}_{\vec{\mathbf{St}}} t_n \equiv t'_n t_n \equiv t'_n \overset{\theta_{n+1}}{\leadsto}_{\vec{\mathbf{St}}^{\equiv}} tt.$$

where, the first n steps are actually $\vec{\mathbf{St}}$-narrowing steps in the FB subtheory $\vec{\mathbf{St}}$, and therefore build a *finitely branching* tree. Only the *last* narrowing step, corresponding to narrowing with the rule $u \equiv u \rightarrow tt$, and *equivalent* to solving the A-unification problem $t_n = t'_n$ with A-unifier θ_{n+1} is a $\vec{\mathbf{St}}^{\equiv}$-narrowing step where the narrowing tree can become infinitely branching due to the infinitary nature of A-unification.

3.4 I am Feeling Lucky!

But what can we do when our convergent theory $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ having an infinitary B-unification algorithm has the boundedness property, but either is not FB (like in our $\vec{\mathbf{St}}^{\in}$ example) or we just

do not *know* for sure whether it is FB? Shall we just give up? Not at all! We should just *wing it*! Maybe *we are lucky*, and the lack of FB does not come back and bite us *for the particular problem we need to solve*. Furthermore, if our problem is a *variant unification* problem, there isn't such a drastic difference between the FB case and the case only enjoying the boundedness property, since the transformed theory $\vec{\mathcal{E}}^{\equiv}$ in which we recast a variant $E \cup B$-unification problem into computing $\vec{\mathcal{E}}^{\equiv}$-variants is *not* an FB theory, but only a theory satisfying the boundedness property. Sure, in the folding variant $\vec{\mathcal{E}}^{\equiv}$-narrowing sequences used to compute $E \cup B$-unifiers of a problem $t = t'$ by narrowing $t \equiv t'$, if we are FB, *only* the last step can be infinitely branching. But what practical difference does it make if, by dropping FB, now *all* narrowing steps are *potentially* infinitely branching? Maybe we are just lucky, and actually they are not so for our given problem $t = t'$. For example, our $\vec{\mathbf{St}}^{\in}$ module is a perfectly fine module: there is nothing wrong with it and if we want to reason formally about the string membership predicate, loss of FB is the price we have to pay. So what? Why should this stop us from reasoning about string memberships?

In the *most common* case of infinitary B-unification that we are likely to encounter in practice, at least using Maude, B is infinitary because of the presence of some associative but not commutative symbol. But the way this infinitary nature of B-unification is handled by Maude should give us strong encouragement and inspiration for not giving up in computing variants and variant unifiers in the non FB case. The reasons for encouragement are the following: (1) Although *in principle* A- and a fortiory B-unification, when $A \subseteq B$ (of course, without making A-operators AC in B!), are infinitary, *in practice* the infinitary case is never encountered in many substantial problems. For example, in analyzing some cryptographic protocols involving strings by folding variant narrowing in the Maude-NPA tool [13], many thousands of A-unification problems need to be solved, yet for all protocols for which this has been done, the infinitary case has not showed up: the A and B-unification algorithms (for $A \subseteq B$) provided by Maude cover a very large class of unification problems for which unification is finitary; and this is greatly helped by the fact that, in narrowing applications, such unification problems always involve terms with *disjoint* sets of variables. (2) Using Maude's *incomplete* A and B-unification algorithms, we can *know* when the algorithm has found a complete set of unifiers and when it has not. This means that, if Maude does *not* print a warning as the ones shown in earlier examples in this paper, we *know* that we are complete, which can be very important for verification problems. For example, if Maude-NPA does not print any such warnings and terminates without having found an specified security attack, we can the know for sure that such an attack is not possible *modulo* the algebraic properties specified for the protocol.

We can put all these brave ideas together in a somewhat more systematic fashion as follows. The incomplete (when $A \subseteq B$) B-unification algorithm that Maude provides can be described as an *incomplete* but *finitary* algorithm of the form $IUnif_B(\phi)$ with respect to the *complete* but *infinitary* algorithm $Unif_B(\phi)$. $IUnif_B(\phi)$ works as follows: for each B-unification system of equations ϕ *always responds* (up to avoiding the Turing tar pits of high computational complexity of algorithms such as AC-unification) with a *pair* (Θ, f), where Θ is a finite set of B-unifiers of ϕ, and where f is a Boolean flag indicating completeness, so that if $f = true$, then we *know* that $\Theta = Unif_B(\phi)$. Instead, if $f = false$, we only know that $\Theta \subseteq Unif_B(\phi)$. Since, at least in Maude, all we actually have available is the pragmatically useful $IUnif_B(\phi)$, we are *always* in the "I am feeling lucky" mode anyway. So why not live with incompleteness (actually not so different in practice form the claimed completeness of finitary unification algorithms when we fall into a Turing tar pit) and just *pretend* that we have a finitary B-unification algorithm, since this is just what $IUnif_B(\phi)$ gives us, while keeping us honest and informed about whether completeness has been achieved?

4 Conclusions and Related Work

So, what do we know, and what can be concluded, from the somewhat long-winded exhortation in Section 3.4 and from all the results in this paper? At least the following:

1. In the infinitary B-unification case, the equivalence $FVP \Leftrightarrow Boundedness$ drops to just an implication $FVP \Rightarrow Boundedness$.
2. The equivalence $FVP \Leftrightarrow Boundedness$ is regained if $\vec{\mathcal{E}}$ is FB, and then:
   - effective computation of a finite set of variants by folding variant narrowing works exactly as in the finitary B-unification case, but
   - variant $E \cup B$-unification also works as before, except that now, it is necessarily infinitary, since B-unification is so.
3. The practical difference between $\vec{\mathcal{E}}$ being FB and satisfying only the boundedness property is not as drastic as one might fear: we should just wing it and hope for the best (i.e., for no incompleteness warnings from Maude). We can still compute: (i) a possibly knowingly incomplete set of variants of a term; and (ii) a possibly knowingly incomplete set of variant $E \cup B$-unifiers.
4. At the cost of having only a generating algorithm (resp. a semi-decision procedure) we can wing it even further and drop the boundedness property, i.e., assume just that $\vec{\mathcal{E}}$ is convergent, then we get: (i) the generation of a possibly infinite and possibly knowingly incomplete set of variants of a term; and (ii) a semi-algorithm for computing a possibly infinite and possibly knowingly incomplete set of variant $E \cup B$-unifiers.

Regarding related work, besides all the previous literature on variants and variant computation mentioned in the references (and in the references in those references), perhaps the most important additional paper behind this work is the, sadly as yet unpublished, SRI manuscript on his brilliant order-sorted A- and B-unification algorithms (with $A \subseteq B$) by Steven Eker [12], for which only a summary has appeared in published form in [9].

Last but not least, without the efficient implementation of the $IUnif_B(\phi)$ algorithm and of variants and variant unification modulo B, also when $A \subseteq B$, in Maude 3 [8], all the results in this paper would be still theoretically relevant, but somewhat of a pipe dream. As it is, thanks to Maude 3 they are also of direct and immediate practical relevance for many applications.

Of course, further work on criteria for checking the FB property, as well as further experimentation with the ideas proposed in this paper, are exciting future prospects.

Acknowledgements. My warmest thanks to Santiago Escobar and Steven Eker for many discussions that have helped me arrive at the ideas presented here. I cordially thank the referees for their very helpful suggestions to improve the paper. This work has been partially supported by NRL under contract N00173-17-1-G002.

References

1. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.E.: ELAN from a rewriting logic point of view. Theoretical Computer Science **285**, 155–185 (2002)
2. Cholewa, A., Meseguer, J., Escobar, S.: Variants of variants and the finite variant property. Tech. rep., CS Dept. University of Illinois at Urbana-Champaign (February 2014), available at `http://hdl.handle.net/2142/47117`
3. Ciobaca., S.: Verification of Composition of Security Protocols with Applications to Electronic Voting. Ph.D. thesis, ENS Cachan (2011)
4. 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)

5. Comon-Lundth, H., Delaune, S.: The finite variant property: how to get rid of some algebraic properties, in Proc *RTA'05*, Springer LNCS 3467, 294–307, 2005
6. Dershowitz, N., Jouannaud, J.P.: Rewrite systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Vol. B, pp. 243–320. North-Holland (1990)
7. van Deursen, A., Heering, J., Klint, P.: Language Prototyping: An Algebraic Specification Approach. World Scientific (1996)
8. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.L.: Programming and symbolic computation in Maude. J. Log. Algebr. Meth. Program. **110** (2020). https://doi.org/10.1016/j.jlamp.2019.100497, https://doi.org/10.1016/j.jlamp.2019.100497
9. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: Associative unification and symbolic reasoning modulo associativity in Maude. In: Rusu, V. (ed.) Proc. Rewriting Logic and Its Applications - 12th International Workshop, WRLA 2018. Lecture Notes in Computer Science, vol. 11152, pp. 98–114. Springer (2018)
10. Durán, F., Lucas, S., Meseguer, J.: Termination modulo combinations of equational theories. In: Frontiers of Combining Systems, 7th International Symposium, FroCoS 2009, Trento, Italy, September 16-18, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5749, pp. 246–262. Springer (2009)
11. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1. Springer (1985)
12. Eker, S.: A pragmatic approach to implementing associative unification, unpublished manuscript, SRI International, circa 2015
13. Escobar, S., Meadows, C., Meseguer, J.: Maude-NPA: cryptographic protocol analysis modulo equational properties. In: Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures, LNCS, vol. 5705, pp. 1–50. Springer (2009)
14. Escobar, S., Sasse, R., Meseguer, J.: Folding variant narrowing and optimal variant termination. J. Algebraic and Logic Programming **81**, 898–928 (2012)
15. Futatsugi, K., Diaconescu, R.: CafeOBJ Report. World Scientific (1998)
16. 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)
17. Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.P.: Introducing OBJ. In: Software Engineering with OBJ: Algebraic Specification in Action, pp. 3–167. Kluwer (2000)
18. Jouannaud, J.P., Kirchner, C., Kirchner, H.: Incremental construction of unification algorithms in equational theories. In: Proc. ICALP'83. pp. 361–373. Springer LNCS 154 (1983)
19. Meier, S., Schmidt, B., Cremers, C., Basin, D.A.: The TAMARIN prover for the symbolic analysis of security protocols. In: Proc. CAV 2013. Lecture Notes in Computer Science, vol. 8044, pp. 696–701. Springer (2013)
20. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)
21. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Proc. WADT'97. pp. 18–61. Springer LNCS 1376 (1998)
22. Meseguer, J.: Variant-based satisfiability in initial algebras. Sci. Comput. Program. **154**, 3–41 (2018)
23. Meseguer, J.: Generalized rewrite theories, coherence completion, and symbolic methods. J. Log. Algebr. Meth. Program. **110** (2020)
24. Skeirik, S., Meseguer, J.: Metalevel algorithms for variant satisfiability. J. Log. Algebr. Meth. Program. **96**, 81–110 (2018)
25. Skeirik, S., Stefanescu, A., Meseguer, J.: A constructor-based reachability logic for rewrite theories. In: Proc. Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017. Lecture Notes in Computer Science, vol. 10855, pp. 201–217. Springer (2017)

Variant Satisfiability of Parameterized Strings

José Meseguer

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

Abstract. Two variant-based satisfiability procedures for QF formulas in the initial models of the instances of increasingly more expressive parameterized data types of strings are proposed. The first has four selector functions decomposing a list concatenation into its parts. The second adds a list membership predicate. The meaning of "parametric" here is much more general than is the case for decision procedures for strings in current SMT solvers, wich are parametric on a finite alphabet. The parameterized data types presented here are parametric on a (typically infinite) algebraic data type of string elements.

1 Introduction

Variant satisfiability [16, 17] is a flexible method to develop decision procedures for satisfiability of QF formulas in initial models of *user-definable* algebraic data types. The power and generality of variant satisfiability can be greatly increased by specifying and proving decidable the satisfiability of QF formulas in *parameterized* data types of the form $\mathbf{Q}[X]$, which map another data type $\mathbf{M}$, also with decidable initial satisfiability and a chosen sort s in $\mathbf{M}$, to the instance $\mathbf{Q}[\mathbf{M},X \mapsto s]$ replacing the formal parameter X by the actual parameter s in $\mathbf{M}$. In fact, a useful collection of such parameterized variant satisfiability procedures was presented in [16]. The procedures presented in this work have the form $\mathbf{St}[X]$ for a parameterized data type of strings with four selector functions decomposing a list concatenation into its parts, which is then extended to a parameterized data type $\mathbf{St}^{\in}[X]$ that adds a list membership predicate.

The procedures can decide a given satisfiability problem by giving a "satisfiable" answer and a witness, or an "unsatisfiable" answer whenever the associative unification problems that need to be solved have a complete set of solutions. If the set of solutions is not complete, a witness proving the satisfiability of the formula can still be provided, but if no such witness is found, the procedure must answer "don't know." However, a full yes-no answer can be given in practice for many problems. This is because the associative unification algorithm in Maude 3 [7, 5] used in the experiments presented in this paper always terminates with a finite set of solutions to a unification problem, but is optimized to succeed in solving many associative unification problems that do have complete finite sets of solutions, and always warns about incompleteness when the set of solutions it returns could be incomplete.

To the best of my knowledge, this is the first variant satisfiability procedure that involves associativity axioms for non-commutative operators: all former variant satifiabilty procedures have involved any combination of associativity and/or commutativity and/or identity axioms, *except* associativity without commutativity. The deep reason for this case not having been treated before is that, as summarized in Section 2.4 and more fully explained in the companion paper [12], all the variant-based results and algorithms have up to now been developed under the assumption that the data type's equational theory has the form $E \cup B$, with the equations E convergent as rewrite rules modulo axioms B, and B having a *finitary* unification algorithm.

Since the associativity axiom needed for strings was outside the class of such axioms B, no theoretical framework existed for it within the known variant-based techniques.

A point worth emphasizing is that the meaning of "parametric" here is much more general than is the case for decision procedures for strings in current SMT solvers, e.g., [18, 1, 19, 11], wich are parametric on a *finite alphabet*. The parameterized modules presented here are parametric on any algebraic data type as its (typically infinite) set of string elements. Those SMT-based procedures for strings and the ones presented here complement each other in several ways: (i) the traditional string procedures assume a finite alphabet of string elements, whereas here an infinite set of elements in a user-definable data type is assumed; for example: strings whose elements can be binary trees holding natural numbers as their leaf elements; (ii) in variant satisfiability, the decision procedures for the data type of elements and for the parameterized module, in this case strings, are seamlessly combined: no Nelson-Oppen-like combination procedures are needed at all.

2 Preliminaries

2.1 Background on Order-Sorted First-Order Logic

We assume familiarity with the notions of an order-sorted signature Σ on a poset of sorts $(S,\leqslant)$, an order-sorted Σ-algebra A, and the term Σ-algebras T_Σ and $T_\Sigma(X)$ for X an S-sorted set of variables. We also assume familiarity with the notions of: (i) order-sorted substitution θ, its domain $dom(\theta)$ and range $ran(\theta)$, and its application $t\theta$ to a term t; (ii) a *preregular* order-sorted signature Σ, where each term t has a least sort, denoted $ls(t)$; (iii) the set $\widehat{S} = S/(\geqslant \cup \leqslant)^+$ of *connected components* of $(S,\leqslant)$; and (iv) for A a Σ-algebra, the set A_s of it elements of sort $s \in S$, and the set $A_{[s]} = \bigcup_{s' \in [s]} A_{s'}$ for $[s] \in \widehat{S}$. All these notions are explained in detail in [15, 9]. The material below is adapted from [16].

The first-order language of *equational Σ-formulas* is defined in the usual way: its atoms are *Σ-equations* $t = t'$, where $t, t' \in T_\Sigma(X)_{[s]}$ for some $[s] \in \widehat{S}$ and each X_s is assumed countably infinite. The set $Form(\Sigma)$ of *equational Σ-formulas* is then inductively built from atoms by: conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and universal $(\forall x_1 : s_1,...,x_n : s_n)$ and existential $(\exists x_1 : s_1,...,x_n : s_n)$ quantification with distinct sorted variables $x_1 : s_1,...,x_n : s_n$, with $s_1,...,s_n \in S$ (by convention, for $\varnothing$ the empty set of variables and φ a formula, we define $(\forall \varnothing)\, \varphi \equiv (\exists \varnothing)\, \varphi \equiv \varphi$). A literal $\neg(t = t')$ is denoted $t \neq t'$. Given a Σ-algebra A, a formula $\varphi \in Form(\Sigma)$, and an assignment $\alpha \in [Y \to A]$, where $Y \supseteq fvars(\varphi)$, with $fvars(\varphi)$ the free variables of φ, the *satisfaction relation* $A,\alpha \models \varphi$ is defined inductively as usual: for atoms, $A,\alpha \models t = t'$ iff $t\alpha = t'\alpha$; for Boolean connectives it is the corresponding Boolean combination of the satisfaction relations for subformulas; and for quantifiers: $A,\alpha \models (\forall x_1 : s_1,...,x_n : s_n)\, \varphi$ (resp. $A,\alpha \models (\exists x_1 : s_1,...,x_n : s_n)\, \varphi$) holds iff for all $(a_1,...,a_n) \in A_{s_1} \times ... \times A_{s_n}$ (resp. for some $(a_1,...,a_n) \in A_{s_1} \times ... \times A_{s_n}$) we have $A,\alpha[x_1 : s_1 := a_1,...,x_n : s_n := a_n] \models \varphi$, where if $\alpha \in [Y \to A]$, then $\alpha[x_1 : s_1 := a_1,...,x_n : s_n := a_n] \in [(Y \cup \{x_1 : s_1,...,x_n : s_n\}) \to A]$ and is such that for $y : s \in (Y \setminus \{x_1 : s_1,...,x_n : s_n\})$, $\alpha[x_1 : s_1 := a_1,...,x_n : s_n := a_n](y : s) = \alpha(y : s)$, and $\alpha[x_1 : s_1 := a_1,...,x_n : s_n := a_n](x_i : s_i) = a_i$, $1 \leqslant i \leqslant n$. We say that φ is *valid* in A (resp. is *satisfiable* in A) iff $A,\underline{\varnothing} \models (\forall Y)\, \varphi$ (resp. $A,\underline{\varnothing} \models (\exists Y)\, \varphi$), where $Y = fvars(\varphi)$ and $\underline{\varnothing} \in [\underline{\varnothing} \to A]$ denotes the empty S-sorted assignment of values in A to the empty S-sorted family $\underline{\varnothing}$ of variables. The notation $A \models \varphi$ abbreviates validity of φ in A. More generally, a set of formulas $\Gamma \subseteq Form(\Sigma)$ is called *valid* in A, denoted $A \models \Gamma$, iff $A \models \varphi$ for each $\varphi \in \Gamma$. For a subsignature $\Omega \subseteq \Sigma$ and $A \in \mathbf{OSAlg}_\Sigma$, the *reduct* $A|_\Omega \in \mathbf{OSAlg}_\Omega$ agrees with A in the interpretation of all sorts and operations in Ω and discards everything in $\Sigma \setminus \Omega$. If $\varphi \in Form(\Omega)$ we have the equivalence $A \models \varphi \Leftrightarrow A|_\Omega \models \varphi$.

An OS *equational theory* is a pair $T = (\Sigma, E)$, with E a set of Σ-equations. $\mathbf{OSAlg}_{(\Sigma,E)}$ denotes the full subcategory of $\mathbf{OSAlg}_{\Sigma}$ with objects those $A \in \mathbf{OSAlg}_{\Sigma}$ such that $A \models E$, called the (Σ,E)-*algebras*. $\mathbf{OSAlg}_{(\Sigma,E)}$ has an *initial algebra* $T_{\Sigma/E}$ [15]. Given $T = (\Sigma,E)$ and $\varphi \in Form(\Sigma)$, we call φ *T-valid*, written $E \models \varphi$, iff $A \models \varphi$ for all $A \in \mathbf{OSAlg}_{(\Sigma,E)}$. We call φ *T-satisfiable* iff there exists $A \in \mathbf{OSAlg}_{(\Sigma,E)}$ with φ satisfiable in A. Note that φ is *T-valid* iff $\neg\varphi$ is *T-unsatisfiable*. The inference system in [15] is *sound and complete* for OS equational deduction, i.e., for any OS equational theory (Σ,E), and Σ-equation $u = v$ we have an equivalence $E \vdash u = v \Leftrightarrow E \models u = v$. Deducibility $E \vdash u = v$ is abbreviated as $u =_E v$, called *E-equality*. An *E-unifier* of a system of Σ-equations, i.e., of a conjunction $\phi = u_1 = v_1 \wedge ... \wedge u_n = v_n$ of Σ-equations, is a substitution σ such that $u_i\sigma =_E v_i\sigma$, $1 \leqslant i \leqslant n$. An *E-unification algorithm* for (Σ,E) is an algorithm generating a *complete set* of E-unifiers $Unif_E(\phi)$ for any system of Σ equations ϕ, where "complete" means that for any E-unifier σ of ϕ there is a $\tau \in Unif_E(\phi)$ and a substitution ρ such that $\sigma =_E (\tau\rho)|_{dom(\sigma)\cup dom(\tau)}$, where $=_E$ here means that for any variable x we have $x\sigma =_E x(\tau\rho)|_{dom(\sigma)\cup dom(\tau)}$. The algorithm is *finitary* if it always terminates with a *finite set* $Unif_E(\phi)$ for any ϕ.

Given a set of equational axioms B used for deduction modulo B, a preregular OS signature Σ is called *B-preregular*[1] iff for each $u = v \in B$ and substitutions ρ, $ls(u\rho) = ls(v\rho)$. The axioms B are called *collapse-free*) iff for each $(u = v) \in B$ neither u not v are variables.

At first sight, the above definition of order-sorted first-order logic seems too restrictive, since we might like to consider more general order-sorted first-order theories (OS-FO theories) having signatures of the form (Σ,Π), where Σ is an OS-signature of function symbols as before, and Π is a signature of typed predicate symbols of the form $p : s_1...s_n$, whose arguments have sorts $s_1...s_n$. However, as explained in [16], there is no real loss of generality in assuming that all atomic formulas are equations: atomic predicates with symbols in Π can be specified as equations by turning predicates into new function symbols of an added sort *Pred* having a constant *tt*, so that each predicate $p : s_1...s_n$ is now viewed as a function symbol $p : s_1...s_n \rightarrow Pred$. Then, any formula φ in the FO signature (Σ,Π) can be transformed into an equational formula $\widetilde{\varphi}$ in the order-sorted signature $\Sigma \cup \Pi$, where *Pred* and *tt* have been added, but where each $p : s_1...s_n$ in the original Π is now represented as a function symbol $p : s_1...s_n \rightarrow Pred$. The $\varphi \mapsto \widetilde{\varphi}$ tranformation is very simple: each atom $p(t_1,...,t_n)$ is replaced by the equational atom $p(t_1,...,t_n) = tt$. In this way, if Γ is a set of first-order formulas in the OS-FO signature (Σ,Π), then $\widetilde{\Gamma}$ is a set of equational first-order formulas in the functional signature $\Sigma \cup \Pi$. Since an OS-FO *theory* is just a pair $((\Sigma,\Pi),\Gamma)$ with Γ a set of first-order (Σ,Π)-formulas, we then get a *semantically equivalent* [16] theory $(\Sigma \cup \Pi,\widetilde{\Gamma})$. By abuse of language we call $((\Sigma,\Pi),\Gamma)$ *equational* iff $(\Sigma \cup \Pi,\widetilde{\Gamma})$ is an equational theory. The semantic equivalence between these two formulations involves also the fact that equational theories in this sense both have initial models, which can be recovered from each other. In particular, we can recover from the initial algebra $T_{\Sigma\cup\Pi/\widetilde{\Gamma}}$ the initial model of $((\Sigma,\Pi),\Gamma)$ [16].

2.2 Convergent Theories, Constructors and Narrowing

Given an order-sorted equational theory $\mathcal{E} = (\Sigma, E \cup B)$, where B is a collection of associativity and/or commutativity and/or identity axioms and Σ is B-preregular, we can associate to it a corresponding *rewrite theory* [14] $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ by orienting the equations E as left-to right rewrite

[1] If $B = B_0 \uplus U$, with B_0 associativity and/or commutativity axioms, and U identity axioms, the B-preregularity notion can be *broadened* by requiring only that: (i) Σ is B_0-preregular in the standard sense that $ls(u\rho) = ls(v\rho)$ for all $u = v \in B_0$ and substitutions ρ; and (ii) the axioms U oriented as rules $\vec{U}$ are *sort-decreasing* in the sense explained in Section 2.2.

rules. That is, each $(u = v) \in E$ is transformed into a rewrite rule $u \to v$. The main purpose of the rewrite theory $\vec{\mathcal{E}}$ is to reduce the complex bidirectional reasoning with equations to the much simpler unidirectional reasoning with rules under suitable assumptions. We assume familiarity with the notion of subterm $t|_p$ of t at a term position p and of term replacement $t[w]_p$ of $t|_p$ by w at position p (see, e.g., [4]). The rewrite relation $t \to_{\vec{E},B} t'$ (which can be abbreviated to $t \to_{\vec{\mathcal{E}}} t'$) holds iff there is a subterm $t|_p$ of t, a rule $(u \to v) \in \vec{E}$ and a substitution θ such that $u\theta =_B t|_p$, and $t' = t[v\theta]_p$. We denote by $\to^*_{\vec{E},B}$ the reflexive-transitive closure of $\to_{\vec{E},B}$. The requirements on $\vec{\mathcal{E}}$ allowing us to reduce equational reasoning to rewriting are the following: (i) $vars(v) \subseteq vars(u)$; (ii) *sort-decreasingness*: for each substitution θ we must have $ls(u\theta) \geqslant ls(v\theta)$; (iii) *strict B-coherence*: if $t_1 \to_{\vec{E},B} t'_1$ and $t_1 =_B t_2$ then there exists $t_2 \to_{\vec{E},B} t'_2$ with $t'_1 =_B t'_2$; (iv) *confluence*: for each term t if $t \to^*_{\vec{E},B} v_1$ and $t \to^*_{\vec{E},B} v_2$, then there exist rewrite sequences $v_1 \to^*_{\vec{E},B} w_1$ and $v_2 \to^*_{\vec{E},B} w_2$ such that $w_1 =_B w_2$; (v) *termination*: the relation $\to_{\vec{E},B}$ is well-founded. If $\vec{\mathcal{E}}$ satisfies conditions (i)–(v) then it is called *convergent*. The key point is that then, given a term t, all terminating rewrite sequences $t \to^*_{\vec{E},B} w$ end in a term w, denoted $t!_{\vec{\mathcal{E}}}$, that is unique up to B-equality, and its called t's *canonical form*. Three major results then follow for the ground convergent case: (1) (Church-Rosser Theorem) for any terms t,t' we have $t =_{E \cup B} t'$ iff $t!_{\vec{\mathcal{E}}} =_B t'!_{\vec{\mathcal{E}}}$, (2) the canonical forms of ground terms are the elements of the *canonical term algebra* $C_{\Sigma/E,B}$, where for each $f : s_1 \ldots s_n \to s$ in Σ and canonical terms $t_1 \ldots t_n$ with $ls(t_i) \leqslant s_i$ the operation $f_{C_{\Sigma/E,B}}$ is defined by the identity: $f_{C_{\Sigma/E,B}}(t_1 \ldots t_n) = f(t_1 \ldots t_n)!_{\vec{\mathcal{E}}}$, and (3) we have a Σ-isomorphism $T_{\mathcal{E}} \cong C_{\Sigma/E,B}$.

Given a convergent rewrite theory $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ and a subsignature Ω on the same poset of sorts, a *constructor subspecification* is a convergent rewrite subtheory $\vec{\mathcal{E}}_\Omega = (\Omega, B_\Omega, \vec{E}_\Omega)$ of $\vec{\mathcal{E}}$ (i.e., we have an inclusion of convergent theories $(\Omega, B_\Omega, \vec{E}_\Omega) \subseteq (\Sigma, B, \vec{E})$) such that: (i) for each ground term t, $t!_{\vec{\mathcal{E}}} \in T_\Omega$, and (ii) $T_{\mathcal{E}}|_\Omega \cong T_{\mathcal{E}_\Omega}$. Furthermore, if $E_\Omega = \emptyset$, Ω is then called a signature of *free constructors modulo axioms* B_Ω. Furthermore, if $\vec{\mathcal{E}}_\Omega \subseteq \vec{\mathcal{E}}$ is a constructor subspecification we say that $\vec{\mathcal{E}}$ *sufficiently complete* w.r.t. Ω.

Whenever we have an inclusion of convergent theories $\vec{\mathcal{E}} \subseteq \vec{\mathcal{E}}'$, with respective signatures Σ and Σ', such that $T_{\mathcal{E}'}|_\Sigma \cong T_{\mathcal{E}}$, we say that $\vec{\mathcal{E}}'$ *protects* $\vec{\mathcal{E}}$. Therefore, condition (ii) above just states that $\vec{\mathcal{E}}$ protects $\vec{\mathcal{E}}_\Omega$.

Narrowing. Given a convergent $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ such that B has a (not necessarily finitary) unification algorithm, by replacing B-matching by B-unification, we can generalize the rewrite relation $t \to_{\vec{\mathcal{E}}} t'$ to the *narrowing relation* $t \rightsquigarrow_{\vec{\mathcal{E}}} t'$, often decorated as $t \overset{\theta}{\rightsquigarrow}_{\vec{\mathcal{E}}} t'$, which holds between Σ-terms t and t' iff there is a *non-variable* position p in t, a rewrite rule $(l \to r) \in \vec{E}$ (renamed if necessary so as not to share variables with t), and a B-unifier $\theta \in Unif_B(l = t|_p)$, with $ran(\theta)$ all fresh new variables "standardized apart," i.e., never generated before in the same narrowing process, such that $t' = (t[r]_p)\theta$. Likewise, in $t \overset{\theta}{\rightsquigarrow}{}^*_{\vec{\mathcal{E}}} t'$, the relation $\rightsquigarrow^*_{\vec{\mathcal{E}}}$ denotes the reflexive-transitive closure of $\rightsquigarrow_{\vec{\mathcal{E}}}$, and θ denotes the composition $\theta = \theta_1 \ldots \theta_n$ of the substitutions appearing in the n steps of the narrowing sequence, or the identity substitution when the sequence has length 0. What narrowing a term t with $\rightsquigarrow_{\vec{\mathcal{E}}}$, means is to *symbolically execute* t in the theory $\vec{\mathcal{E}}$. That is, even though t may be in $\vec{\mathcal{E}}$-canonical form and may not be executable by rewriting with $\to_{\vec{\mathcal{E}}}$, by instantiating it with $\rightsquigarrow_{\vec{\mathcal{E}}}$ in all possible "most general" ways, it *becomes* executable. Specifically, if $t \overset{\theta}{\rightsquigarrow}{}^*_{\vec{\mathcal{E}}} t'$ holds, then $t\theta \to^*_{\vec{\mathcal{E}}} t'$ also holds and, conversely, (the so-called "Lifting

Lemma" [10]), if γ is an $\vec{\mathcal{E}}$-canonical substitution and $t\gamma \to^*_{\vec{\mathcal{E}}} u$ holds, then there is a narrowing sequence $t \overset{\theta}{\leadsto}{}^*_{\vec{\mathcal{E}}} t'$ of same length as $t\gamma \to^*_{\vec{\mathcal{E}}} u$, and with same term positions and rules at each step, and a substitution δ such that $t'\delta =_B u$.

2.3 Variant Satisfiability in a Nutshell

Given a convergent rewrite theory $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ associated to an equational theory $\mathcal{E}$ a *variant* of a term t is a pair (u,θ) such that θ is a substitution in canonical form, i.e., $\theta(x) = \theta(x)!_{\vec{\mathcal{E}}}$ for each x, and $u =_B (t\theta)!_{\vec{\mathcal{E}}}$. $\vec{\mathcal{E}}$ has the *finite variant property* (FVP) iff for any term t there is a finite set $[\![t]\!]_{\vec{\mathcal{E}}} = \{(u_1,\theta_1),...,(u_n,\theta_n)\}$ of variants of t that are *most general possible* among all such variants, where the "most general" relation $\sqsupseteq$ between variants is defined by the equivalence: $(v,\alpha) \sqsupseteq (w,\beta) \Leftrightarrow \exists \gamma (\beta =_B \alpha\gamma \wedge v\gamma =_B w)$. Furthermore, if B has a finitary unification algorithm, a finite set of most general variants in $[\![t]\!]_{\vec{\mathcal{E}}}$ can be effectively computed for any t by *folding variant narrowing* [8].

There are two ways to understand variants and FVP. One is in terms of the notion of a *pattern*, i.e., a term u with variables describing *something*. The variants $\{(u_1,\theta_1),...,(u_n,\theta_n)\}$ of a term t in an FVP theory $\vec{\mathcal{E}}$ are clearly patterns $u_1,...,u_n$ (plus the added technical monkeys $\theta_1,...,\theta_n$). But what do they describe? Obviously, up to B-equality, the infinite set of *all* patterns of the form $(t\theta)!_{\vec{\mathcal{E}}}$ for a given term t. But there is a second, equivalent way of understanding the FVP notion. Implicit in the idea that for *any* variant (w,β) of t there is a most general one $(u_1,\theta_1) \in [\![t]\!]_{\vec{\mathcal{E}}}$ lies the property, called the *boundedness property* [3]: that *all variants* of a term t can be computed in a finite number of rewriting steps smaller or equal to a *fixed bound* $bd(t)$ depending on t. Why so? Because: (1) we can choose $bd(t)$ to be the maximum of the smallest rewriting depths needed to compute $(t\theta_i)!_{\vec{\mathcal{E}}} = u_i$ for $1 \leqslant i \leqslant n$, and (2) since for each variant (w,β) of t there is a $(u_i,\theta_i) \in [\![t]\!]_{\vec{\mathcal{E}}}$ such that $(u_i,\theta_i) \sqsupseteq (w,\beta) \Leftrightarrow \exists \gamma (\beta =_B \theta_i\gamma \wedge u_i\gamma =_B w)$, by the fact that the rewrite relation is B-coherent, we can obtain a rewrite sequence from $t\beta$ to $(t\beta)!_{\vec{\mathcal{E}}} =_B w$ of length l with $l \leqslant bd(t)$ just by instantiating by γ the sequence of length l from $t\theta_i$ to $(t\theta_i)!_{\vec{\mathcal{E}}} =_B u_i$.

In fact we have the following equivalence (see [3], and for a more precise statement, [2]).

Theorem 1. *(FVP iff Boundedness). Given a convergent $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ associated to an equational theory $\mathcal{E}$ such that B has a finitary unification algorithm, $\vec{\mathcal{E}}$ is FVP iff $\vec{\mathcal{E}}$ has the boundedness property.*

Furthermore, since for *any* convergent $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ such that B has a finitary unification a complete (but not necessarily finite) set of $\vec{\mathcal{E}}$-variants of a term t can be effectively generated by folding variant narrowing [8], and in fact this has been mechanized in Maude, as proved in [2], there is a simple semi-decision procedure to check whether such an $\vec{\mathcal{E}}$ is FVP: it is so if and only if for each $f \in \Sigma$ the term $f(x_1,...,x_n)$, with variables $x_1,...,x_n$ of most general possible sorts, has a finite number of most general $\vec{\mathcal{E}}$-variants. Call this property the *Σ-boundedness* property.

Another key point about $\vec{\mathcal{E}}$ being FVP is that, as proved in [8], if B has a finitary unification algorithm, then $E \cup B$ has also a finitary unification algorithm called *variant unification*. If $\vec{\mathcal{E}}$ is FVP and it has a constructor subspecification $\vec{\mathcal{E}}_\Omega \subseteq \vec{\mathcal{E}}$, say with $\vec{\mathcal{E}}_\Omega = (\Omega, B_\Omega, \vec{E}_\Omega)$, then several more notions appear: (1) A *constructor variant* of t is a variant (u,θ) of t such that u is an Ω-term. The set of constructor variants of t is denoted $[\![t]\!]^{\Omega}_{\vec{\mathcal{E}}}$. (2) A *constructor unifier* of a system of Σ-equations $u_1 = v_1 \wedge ... \wedge u_n = v_n$ is a $E \cup B$-unifier α such that for $1 \leqslant i \leqslant n$, $u_i\alpha!_{\vec{\mathcal{E}}}$ is an

Ω-term. As explained in [16, 17], under very mild conditions on the constructor subspecification $\vec{\mathcal{E}}_\Omega \subseteq \vec{\mathcal{E}}$, if $\vec{\mathcal{E}}$ is FVP and B and B_Ω have finitary unification algorithms, there is an effectively computable finite subset $\{(u_1,\theta_1),...,(u_n,\theta_n)\}$ of most general constructor variants in $[\![t]\!]^\Omega_{\vec{\mathcal{E}}}$. Also, under the same assumptions, for any system of Σ-equations $\phi \equiv u_1 = v_1 \wedge ... \wedge u_n = v_n$ there is an algorithm computing a finite set $Unif^\Omega_{E\cup B}(\phi)$ of most general constructor unifiers. In particular, since any ground substitution $\rho : Y \to T_\Sigma$ is $E \cup B$-equivalent to the ground constructor substitution $\rho!_{\vec{\mathcal{E}}}$, any ground unifier of ϕ is $E \cup B$-equivalent to a ground constructor unifier that is an instance up to $E \cup B$-equality of a constructor unifier in $Unif^\Omega_{E\cup B}(\phi)$.

But there is more. If $\vec{\mathcal{E}}$ is FVP, has a constructor subspecification $\vec{\mathcal{E}}_\Omega \subseteq \vec{\mathcal{E}}$, with both B and B_Ω having finitary unification algorithms, *and* satisfiability of QF Ω-formulas in $T_{\mathcal{E}_\Omega}$ is decidable, then satisfiability of QF Σ-formulas in $T_{\mathcal{E}}$ is also decidable by a *variant satisfiability* algorithm [16, 17].

The plot then thickens, since to prove decidable satisfiability in $T_{\mathcal{E}}$, we just need criteria ensuring decidable satisfiability in the initial algebra of constructors $T_{\mathcal{E}_\Omega}$. The simplest, yet commonly occurring, case is when $\vec{\mathcal{E}}_\Omega$ has the form: $\vec{\mathcal{E}}_\Omega = (\Omega, B_\Omega, \emptyset)$, i.e., the constructors are free modulo B_Ω, and B_Ω is any combination of associativity and/or commutativity and/or identity axioms, except associativity without commutativity [16]. This result is a special instance of a more general criterion, called *OS-compactness*. Here is the definition, slightly extended to include the notion of *weakly OS-compact* theory that we shall need to deal with associative but not commutative axioms as those of strings:

Definition 1. *(OS-Compactness). An equational OS-FO theory* $((\Sigma,\Pi),\Gamma)$ *is called* OS-compact *(resp.* weakly OS-compact*) iff: (i) for each sort s in Σ we can effectively determine whether s is finite or infinite in $T_{\Sigma\cup\Pi/\widetilde{\Gamma},s}$, and, if finite, we can effectively compute a representative ground term $rep([u]) \in [u]$ for each $[u] \in T_{\Sigma\cup\Pi/\widetilde{\Gamma},s}$; (ii) $=_{\widetilde{\Gamma}}$ is decidable; (iii) $\widetilde{\Gamma}$ has a finitary unification algorithm (resp. a unification algorithm); and (iv) if $\bigwedge D$ is a finite conjunction of negated (Σ,Π)-atoms whose variables have all infinite sorts and for each $u \neq v$ in $\bigwedge \widetilde{D}$ we have $u \neq_{\widetilde{\Gamma}} v$, then $\bigwedge D$ is satisfiable in $T_{\Sigma\cup\Pi/\widetilde{\Gamma}}$.*

*We call an OS equational theory (Σ,E) OS-*compact *(resp.* weakly OS-compact *iff the OS-FO theory $((\Sigma,\emptyset),E)$ is OS-*compact *(resp.* weakly OS-compact*).*

The proof that $\vec{\mathcal{E}}_\Omega = (\Omega, B_\Omega, \emptyset)$ is OS-compact when B_Ω is any combination of associativity and/or commutativity and/or identity axioms, except associativity without commutativity relies on the crucial fact that then any equation $u = v$ involving a single variable x has a *finite* set of solutions modulo B_Ω. That is why, up to now, no variant satisfiability results have been proved when some of the constructor axioms in B_Ω are associative but not commutative: the OS-compactness proof does not extend to that case. For example, if $\_\_$ is an associative operator and a a constant to which it can be applied, the equation $a\,x = x\,a$ does *not* have a finite set of solutions modulo associativity, but the infinite set of solutions $\{x = a, x = a\,a, x = a\,a\,a, ..., x = a \overset{n}{...} a, ...\}$. In fact, to deal with the variant-related properties of convergent theories having axioms B for which B-unification is infinitary new concepts and results are needed. Fortunately, those needed in this paper have recently been developed in the companion paper [12], which I summarize in what follows.

2.4 Variants and FVP when B-unification is Infinitary

The first main difference between the finitary and infinitary B-unification cases is that in the infinitary case Theorem 1 no longer holds: all we have then is that FVP implies the boundedness

property [12]. The reason for this discrepancy is that a convergent theory may enjoy the boundedness property but may fail to be FVP because the narrowing tree of a term may be infinitely branching for some nodes due to the infinitary nature of B-unification, so in general there isn't anymore a finite set of nodes in the narrowing tree of t up to bound $bd(t)$ from which we can gather a *finite* set of most general variants. To recover the equivalence between the boundedness property and FVP in the infinitary case, we need the new notion of an FB theory:

Definition 2. *[12] (FB/CFB Theories). A convergent theory $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ having a B-unification algorithm is called* finitely branching *(FB) (resp.* contextually finitely branching *(CFB) with respect to a theory $\vec{\mathcal{E}}'$ whose axioms B' are collapse-free) iff for each $(l \to r) \in \vec{E}$ and any equality $l = w$ such that w is not a variable and $vars(l) \cap vars(w) = \emptyset$, the set $Unif_B(l = w)$ is finite (resp. and any equality $l = w$ such that w is a term which is not a variable in the disjoint union $\vec{\mathcal{E}} \uplus \vec{\mathcal{E}}'$, (where all sorts and function symbols in $\vec{\mathcal{E}}$ and $\vec{\mathcal{E}}'$ have been made different, except for the possible identification of two sorts s in $\vec{\mathcal{E}}$ and s' in $\vec{\mathcal{E}}'$) and such that $vars(l) \cap vars(w) = \emptyset$, the set $Unif_{B \uplus B'}(l = w)$ is finite).*

The nice result about the FB property is that if: (i) $\vec{\mathcal{E}}$ is FB and (i) enjoys the easy to check Σ-boundedness property, then $\vec{\mathcal{E}}$ is FVP [12]. Furthermore, then the finite set of variants for any term t can be computed by folding variant narrowing *in exactly the same way* as in the finitary case. One important difference, however, is that the variant $E \cup B$-unification algorithm for an FVP theory is no longer finitary. This is to be expected, since variant $E \cup B$-unification relies on B-unification, which is itself infinitary.

This infinitary nature of $E \cup B$-unification poses some substantial challenges, but they are not unsurmountable. Specifically, assuming that B is a combination of associativity and/or commutativity and/or identity axioms which include some associative axioms for non-commutative operators, these substantial challenges have already been addressed by the *incomplete*, but complete in a large number of practical cases, B-unification algorithm designed by Steven Eker in [7] and summarized in [6], which is supported by Maude 3 [5]. Let me summarize this incomplete algorithm. Let us call it $IUnif_B$. Given a system of equations ϕ, $IUnif_B$ provides a generally *incomplete* but *finitary* algorithm computing $E \cup B$-unifiers $IUnif_B(\phi)$ instead of the *complete* but generally *infinitary* algorithm $Unif_B(\phi)$. $IUnif_B(\phi)$ works as follows: for each B-unification system of equations ϕ *always responds* with a *pair* (Θ, f), where Θ is a finite set of B-unifiers of ϕ, and where f is a Boolean flag indicating completeness, so that if $f = true$, then we *know* that $\Theta = Unif_B(\phi)$. Instead, if $f = false$, we only know that $\Theta \subseteq Unif_B(\phi)$. Since, at least in Maude, all we actually have available is the pragmatically useful $IUnif_B(\phi)$, we are *always* potentially incomplete anyway, except for computing the variants of a term t when $\vec{\mathcal{E}}$ is FB and enjoys the Σ-boundedness property. This can have a liberating effect. After all, we may as well just relax the FVP requirement and keep only the easy to check requirement that our convergent $\vec{\mathcal{E}}$ has the Σ-boundedness property. *If* for the computations of the $\vec{\mathcal{E}}$-variants of a term t, or the variant $E \cup B$ unifiers of a system of equations ϕ Maude does not give any warnings about the underlying $IUnif_B$ algorithm having returned an incomplete solution set, *then* our results are *complete*, and all our variant-related algorithms, and all our theoretical claims are *exactly the same* as in the finitary B-unifications case. In practice, this ideal scenario can happen considerably more often than one might expect.

3 Variant Satisfiability for Two Parameterized String Data Types

I begin with a parameterized data type $\mathbf{St}[X]$ of strings with an associative string concatenation operator _ _ that enjoys the Σ-boundedness property and ensures the FB property (and there-

fore the FVP) for any instance $\mathbf{St}[s]$ under reasonable assumptions. I then extend $\mathbf{St}[X]$ to a parameterized data type $\mathbf{St}^{\in}[X]$ by adding a string membership predicate. $\mathbf{St}^{\in}[X]$ enjoys the Σ-boundedness property but not the FB property. $\mathbf{St}^{\in}[X]$ is not FVP but, as just mentioned in Section 2.4, this is no real obstacle in practice when we rely on the underlying $IUnif_B$ algorithm: if some incompleteness problem is found, we will know about it and, as explained below, even this may not hinder us from achieving our desired verification goals. I then show how for any instances of $\mathbf{St}[X]$, resp. $\mathbf{St}^{\in}[X]$, satisfiability problems in the initial algebra of such an instance can be either decided, or may instead receive the "unknown" answer, depending on the possible warnings of the underlying $IUnif_B$ algorithm. Let me describe $\mathbf{St}[X]$.

Example 1. (Strings Parameterized Data Type $\mathbf{St}[X]$). Consider the following algebraic data type of strings, where X is its *parameter sort*. Its rewrite theory $\vec{\mathbf{St}}[X] = (\Sigma, A, \vec{E})$ has signature Σ with sorts X,[2] *NeSt* and *St*, subsort inclusions $X < NeSt < St$, a constant ε of sort *St*, a subsort-overloaded string concatenation operator $\_\_ : St\ St \to St$, $\_\_ : NeSt\ NeSt \to NeSt$, two functions $first, last : NeSt \to X$, and two other functions $rest, prior : NeSt \to St$. The only axiom in A is the associativity axiom $(u\,v)\,w = u\,(v\,w)$, were u,v,w are variables of sort *St*, p,q,r variables of sort *NeSt*, and x is a variable of sort X. The rules $\vec{E}$ are: (i) the identity rules $u\,\varepsilon \to u$ and $\varepsilon\,u \to u$, (ii) the *first* and *rest* rules $first(x) \to x$, $first(x\,q) \to x$, $rest(x) \to \varepsilon$, $rest(x\,q) \to q$, and (iii) the *last* and *prior* rules $last(x) \to x$, $last(q\,x) \to x$, $prior(x) \to \varepsilon$, $prior(q\,x) \to q$. $\vec{\mathbf{St}}$ has a constructor subtheory[3] $\vec{\mathbf{St}}_{\Omega} = (\Omega, A_{\Omega}, \emptyset)$ with same sorts and subsorts and where Ω has the constant ε and the concatenation operator for non-empty strings $\_\_ : NeSt\ NeSt \to NeSt$, and A_{Ω} has just the associativity *axiom* $(p\,q)\,r = p\,(q\,r)$. The theory $\vec{\mathbf{St}}$ is clearly terminating, since the associativity axiom is term-size-preserving, and all the rules are term-size-decreasing. It is also sort-decreasing and strictly A-coherent. The only critical pairs modulo A are those between the rules $u\,\varepsilon \to u$ and $\varepsilon\,u \to u$, which are clearly joinable, so $\vec{\mathbf{St}}$ is also confluent and therefore convergent. It satisfies also the Σ-boundedness property: for each non-constant symbol $f \in \Sigma$, the term $f(x_1,...,x_n)$ has exactly 3 most general variants.

The parameterized data type $\mathbf{St}[X]$ then defines a *theory transformation* of the form $(\mathbf{M}, s) \mapsto \mathbf{St}[\mathbf{M}, X \mapsto s]$. That is, given a convergent theory $\vec{\mathbf{M}}$ and a sort s in $\vec{\mathbf{M}}$, we obtain the theory $\mathbf{St}[\mathbf{M}, X \mapsto s]$ by first forming the *disjoint union* of $\vec{\mathbf{St}}[X]$ and $\vec{\mathbf{M}}$, and then identifying the sorts X and s. For example, for $\mathbf{M}$ a specification $\mathbf{N}$ of the natural numbers with a sort *Nat*, then $\vec{\mathbf{St}}[\mathbf{N}, X \mapsto Nat]$ gives us a data type of strings of natural numbers, and for $\mathbf{M}$ a specification **Bool** of the booleans with a sort *Bool*, $\vec{\mathbf{St}}[\mathbf{Bool}, X \mapsto Bool]$ gives us a data type of bit strings. A natural question to ask is: when are the instances of $\mathbf{St}[X]$ FVP? This question has been answered and proved in [12] as follows:

Theorem 2. *[12] (CFB for $\vec{\mathbf{St}}[\mathbf{M}, X \mapsto s]$). Assuming the axioms $B_{\mathbf{M}}$ of $\mathbf{M}$ are any combination of associativity and/or commutativity axioms, the transformation $(\mathbf{M}, s) \mapsto \mathbf{St}[\mathbf{M}, X \mapsto s]$ preserves the CFB property. That is, if $\vec{\mathbf{M}}$ is FB and CFB with respect to $\vec{\mathbf{St}}[X]$, then $\vec{\mathbf{St}}[X]$ is also CFB with respect to $\vec{\mathbf{M}}$, and therefore $\vec{\mathbf{St}}[\mathbf{M}, X \mapsto s]$ is FB (and hence FVP) for any pair $(\mathbf{M}, s)$.*

Let me now describe the $\vec{\mathbf{St}}^{\in}[X]$ extension:

[2] in Maude, X would be the sort `Elt` of the `TRIV` parameter theory.

[3] For more details about sufficient completeness of parameterized OS theories and methods for checking it see [13].

Example 2. (Strings Parameterized Data Type $\vec{\mathbf{St}}^{\in}[X]$). The extension $\vec{\mathbf{St}}^{\in}[X]$ has the form: $\vec{\mathbf{St}}^{\in}[X] = (\Sigma \cup \Pi, A, \vec{E} \cup \vec{\Gamma})$ and is obtained from an OS-FO theory $\mathbf{St}^{\in}[X] = ((\Sigma, \Pi), A, \vec{E} \cup \Gamma)$ where Π has just a string membership predicate $\_\in\_ : X\,St$, which is expressed functionally in $\Sigma \cup \Pi$ as $\_\in\_ : X\,St \to Pred$, where the new sort *Pred* with constant *tt* has been added to Π. The string membership axioms Γ are: $x \in x$, $x \in x\,u$, $x \in u\,x$, $x \in u\,x\,v$, and are expressed in $\mathbf{St}[X]$ as the following string membership rules $\vec{\Gamma}$: $x \in x \to tt$, $x \in x\,u \to tt$, $x \in u\,x \to tt$, and $x \in u\,x\,v \to tt$. $\mathbf{St}^{\in}[X]$ has a constructor subtheory $\mathbf{St}^{\in}_{\Omega \cup \Pi}[X] = (\Omega \cup \Pi, A_{\Omega}, \vec{\Gamma})$. That is, $\mathbf{St}^{\in}_{\Omega \cup \Pi}[X]$ just adds to $\vec{\mathbf{St}}_{\Omega}$ the rules $\vec{\Gamma}$ and Π with its new sort *Pred*. $\vec{\mathbf{St}}^{\in}$ is likewise convergent and satisfies the Σ-boundedness property: for the new operator $\_\in\_$, the most general term $x \in u$ has 5 variants. However, as explained in [12], the new rules in $\vec{\Gamma}$ are *not* FB, so we cannot expect $\vec{\mathbf{St}}^{\in}$ to produce FVP instances; but it can produce instances satisfying the Σ-boundedness property, which for variant satisfiability purposes may be just enough as we shall see.

Let now $\vec{\mathbf{M}} = (\Sigma_{\mathbf{M}}, B_{\mathbf{M}}, \vec{E}_{\mathbf{M}})$ have only associative and/or commutative axioms $B_{\mathbf{M}}$, be convergent, enjoy the Σ-boundedness property, and have a constructor subtheory $\vec{\mathbf{M}}_{\Omega} = (\Sigma_{\mathbf{M}_{\Omega}}, B_{\mathbf{M}_{\Omega}}, \vec{E}_{\mathbf{M}_{\Omega}})$ enjoying the same properties. Let now s be a sort in $\vec{\mathbf{M}}$, and let us consider the relationship between $\vec{\mathbf{St}}^{\in}[\mathbf{M}, X \mapsto s]$ and $\mathbf{St}^{\in}_{\Omega \cup \Pi}[\mathbf{M}_{\Omega}, X \mapsto s]$. Since in $\mathbf{St}^{\in}_{\Omega \cup \Pi}[X]$ the only terms of sort X are variables $x, y, \ldots$ of sort X, and $\mathbf{St}^{\in}_{\Omega \cup \Pi}[X]$ is made up of constructors, we have the following protecting relationships between the corresponding initial algebras (again, see [13] for more details about sufficient completeness of parameterized OS theories): (1) (parameter protection) $T_{\mathbf{St}^{\in}[\mathbf{M}, X \mapsto s]}\big|_{\Sigma_{\mathbf{M}}} \cong T_{\mathbf{M}}$, (2) (parameter constructor protection) $T_{\mathbf{St}^{\in}_{\Omega \cup \Pi}[\mathbf{M}_{\Omega}, X \mapsto s]}\big|_{\Sigma_{\mathbf{M}_{\Omega}}} \cong T_{\mathbf{M}_{\Omega}}$, and (3) (instance protection): $T_{\mathbf{St}^{\in}[\mathbf{M}, X \mapsto s]}\big|_{\Sigma_{\mathbf{M}} \cup \Omega \cup \Pi} \cong T_{\mathbf{St}^{\in}_{\Omega \cup \Pi}[\mathbf{M}_{\Omega}, X \mapsto s]}$. In particular, by (3), $\mathbf{St}^{\in}_{\Omega \cup \Pi}[\mathbf{M}_{\Omega}, X \mapsto s]$ is the constructor subtheory of $\mathbf{St}^{\in}[\mathbf{M}, X \mapsto s]$. (4) $\vec{\mathbf{St}}^{\in}[\mathbf{M}, X \mapsto s]$ has the Σ-boundedness property because both $\vec{\mathbf{St}}^{\in}[X]$ and $\vec{\mathbf{M}}$ do and, due to the disjointness of the signatures and the axiom sets, the computations of variants for operators with maximal arguments in either of the disjoint signatures can be computed by folding variant narrowing exactly as before: no operator from the other signature can ever show up in such narrowing trees. In particular, Σ-boundedness is also enjoyed by $\mathbf{St}^{\in}_{\Omega \cup \Pi}[\mathbf{M}_{\Omega}, X \mapsto s]$.

3.1 Variant Satisfiability of $\mathbf{St}[X]$

A fortiori, all the above properties (1)–(4) are also satisfied by $\vec{\mathbf{St}}[\mathbf{M}, X \mapsto s]$, $\vec{\mathbf{St}}_{\Omega}[\mathbf{M}_{\Omega}, X \mapsto s]$, and all the associated initial algebras and reducts. Suppose, further, that $\vec{\mathbf{M}}$ is FB and CFB with respect to $\vec{\mathbf{St}}[X]$. Then, by Theorem 2 and by $\vec{\mathbf{M}}$ FB, $\vec{\mathbf{St}}[\mathbf{M}, X \mapsto s]$, $\vec{\mathbf{St}}_{\Omega}[\mathbf{M}_{\Omega}, X \mapsto s]$, $\vec{\mathbf{M}}$, and $\vec{\mathbf{M}}_{\Omega}$ are all FVP. Suppose that $\vec{\mathbf{M}}_{\Omega}$ is also weakly OS-compact and that $T_{\mathbf{M}, s}$ is an infinite set, and therefore so is $T_{\mathbf{M}_{\Omega}, s}$. Note that then: (5) by weak OS-compactness of $\vec{\mathbf{M}}_{\Omega}$ and (2), we can effectively determine whether any sort in $\mathbf{St}_{\Omega}[\mathbf{M}_{\Omega}, X \mapsto s]$ is finite or infinite and can compute representative ground terms for finite sorts; (6) since $\mathbf{St}_{\Omega}[\mathbf{M}_{\Omega}, X \mapsto s]$ is FVP, it has a unification algorithm. Therefore, $\mathbf{St}_{\Omega}[\mathbf{M}_{\Omega}, X \mapsto s]$ satisfies conditions (i)–(iii) in Definition 1 of weak OS-compactness. The key pending question then is: does it also satisfy condition (iv) so that $\mathbf{St}_{\Omega}[\mathbf{M}_{\Omega}, X \mapsto s]$ is actually weakly OS-compact? The answer to this question is provided by the following theorem, whose proof is similar to that for parameterized lists in Theorem 9 of [16].

Theorem 3. *(Preservation of Weak OS-Compactness). Under the above assumptions on* $\vec{\mathbf{M}}$ *and* s, $\mathbf{St}_{\Omega}[\mathbf{M}_{\Omega}, X \mapsto s]$ *is weakly OS-compact.*

Let me now explain how Theorem 3 provides a "knowingly incomplete" satisfiability decision procedure for satisfiability of QF $\mathbf{St}[\mathbf{M},X\mapsto s]$-formulas in the initial algebra $T_{\mathbf{St}[\mathbf{M},X\mapsto s]}$, that is, for satisfiability of QF-formulas in CFB instances of the parameterized string data type when $\mathbf{M}_\Omega$ is weakly OS-compact and s is an infinite sort. This "knowingly incomplete" decision procedure is the natural extension to the infinitary unification case of the variant satisfiability procedure in [16]. By putting QF formulas in disjunctive normal form it is enough to consider conjunctions of literals of the form $\bigwedge G \wedge \bigwedge D$, with the G equalities and the D disequalities. Since $\vec{\mathbf{St}}[\mathbf{M},X\mapsto s]$ is FVP, we can compute the variant $\vec{\mathbf{St}}[\mathbf{M},X\mapsto s]$-unifiers of $\bigwedge G$ by folding variant narrowing modulo axioms $A\cup B_{\mathbf{M}}$. The $A\cup B_{\mathbf{M}}$-unification calls made by the folding variant narrowing algorithm will be computed by $IUnif_{A\cup B_{\mathbf{M}}}$. Since for some calls the number of $Unif_{A\cup B_{\mathbf{M}}}$-unifiers may be infinite, two things can happen for the input $\bigwedge G$. **Case** (a): the variant-unification algorithm returns with a finite number Θ of $\vec{\mathbf{St}}[\mathbf{M},X\mapsto s]$-unifiers and no warning from $IUnif_{A\cup B_{\mathbf{M}}}$. Then we know that such a finite set is complete and can reduce the satisfiability of $\bigwedge G \wedge \bigwedge D$, to that of $\bigvee_{\theta\in\Theta}(\bigwedge D)\theta$. As an optimization, we may first compute the constructor variant unifiers of $\bigwedge G$ from Θ as explained in [17], but this is not essential. Then for each disjunct $(\bigwedge D)\theta$ we can effectively compute its constructor variants, which are conjunctions in $\mathbf{St}_\Omega[\mathbf{M}_\Omega,X\mapsto s]$, and for each such constructor variant and all variables of finite sort in it (which sorts must be in $\mathbf{M}_\Omega$) the canonical forms of all the substitutions of those variables by all the choices of effectively computable ground representatives of those sorts, which we can do thanks to the weak OS-compactness of $\mathbf{M}_\Omega$. Then our original conjunction is satisfiable thanks to Theorem 3 if and only if at least one of those normalized conjunctions of constructor disequalities, whose variables are all infinite, is axiom-consistent.

The "knowingly incomplete" case of this satisfiability decision procedure appears in **Case** (b): we obtain an incompleteness warning from the underlying $IUnif_{A\cup B_{\mathbf{M}}}$ and therefore an incomplete set Θ of variant unifiers of $\bigwedge G$. We then proceed as in the complete case, and can also prove satisfiability of $\bigwedge G \wedge \bigwedge D$ if we can find in the end some axiom-consistent normalized conjunction of constructor disequalities (whose variables are all infinite). However, if we cannot find any such conjunction of constructor disequalities our "knowingly incomplete" procedure must reply: "don't know." Let us see some examples.

First of all, by the syntactic disjointness assumption between $\mathbf{M}$ and $\mathbf{St}[X]$, it is possible to prove *generic theorems* in $\mathbf{St}[X]$, i.e., theorems only involving the syntax of $\mathbf{St}[X]$ which will apply to all instances $\mathbf{St}[\mathbf{M},X\mapsto s]$ under the above assumptions on $\mathbf{M}$ and s. This is because, by the disjointness assumption, up to sort renaming for X, constructor variants and constructor variant unifiers for pure $\mathbf{St}[X]$ formulas coincide in $\mathbf{St}[X]$ and in $\mathbf{St}[\mathbf{M},X\mapsto s]$. Here are two simple such generic theorems, where q has sort *NeSt*:

1. $q = first(q)\, rest(q)$
2. $q = prior(q)\, last(q)$

Let us see how (1) is proved; the proof of (2) is entirely similar. We just need to show that $q \neq first(q)\, rest(q)$ is unsatisfiable. We can do so by computing its constructor variants, which are:

1. $x \neq x$,and
2. $x\,q' \neq x\,q'$

which are both obviously unsatisfiable.

A somewhat more interesting generic theorem is the following equivalence, relating the four functions *first*, *rest*, *prior* and *last*, where x,y have sort X and q,q' sort *NeSt*:

$$first(q) = x \wedge last(q) = y \wedge prior(q) = q' \Leftrightarrow q = x\, rest(q')\, y \wedge prior(q) = q'$$

where the condition $prior(q) = q'$ just states that the string q has length $\geqslant 2$. Note that the simpler equivalence $first(q) = x \wedge last(q) = y \Leftrightarrow q = x\, rest(q')\, y$ is *invalid* for strings, because it fails for strings of length 1. For example, for $q = x$ it yields the contradiction $x = x\, x$.

Proving the above equivalence requires proving for the $(\Rightarrow)$ part that:

1. $first(q) = x \wedge last(q) = y \wedge prior(q) = q' \wedge q \neq x\, rest(q')\, y$, and
2. $first(q) = x \wedge last(q) = y \wedge prior(q) = q' \wedge prior(q) \neq q'$

are both unsatisfiable. But (2) is always false, so unsatisfiable, and we only need to deal with (1).

Likewise, for proving the $(\Leftarrow)$ side we need to show that

1. $q = x\, rest(q')\, y \wedge prior(q) = q' \wedge first(q) \neq x$
2. $q = x\, rest(q')\, y \wedge prior(q) = q' \wedge last(q) \neq y$
3. $q = x\, rest(q')\, y \wedge prior(q) = q' \wedge piror(q) \neq q'$

are all unsatisfiable. But (3) is always false, thus unsatisfiable, so we only need to deal with (1) and (2).

For proving the $(\Rightarrow)$ implication we only need to deal with (1). But selecting the constructor variants unifiers of $first(q) = x \wedge last(q) = y \wedge prior(q) = q'$ among all other we get:

```
Unifier #2

q --> %2:X %3:NeSt %1:X
x --> %2:X
y --> %1:X
q' --> %2:X %3:NeSt

Unifier #3

q --> %2:X %1:X
x --> %2:X
y --> %1:X
q' --> %2:X
```

We can then instantiate $rest(q')$ by these two unifiers to get:

```
Maude> reduce rest(%2:X %3:NeList) .
result NeSt: %3:NeList

Maude> reduce rest(%2:X) .
result St: nil
```

So that the constructor variants of the instances of $q \neq x\, rest(q')\, y$ by unifiers `#2` and `#3` are, respectively:

```
%2:X %3:NeList %1:X =/=  %2:X %3:NeList %1:X

%2:X %1:X =/= %2:X %1:X
```

which are both unsatisfiable.

For proving the $(\Leftarrow)$ part, the constructor unifiers of $q = x\, rest(q')\, y \wedge prior(q) = q'$ are:

```
Unifier #1
rewrites: 15 in 5ms cpu (6ms real) (2676 rewrites/second)
q --> %1:X %2:X
```

```
x --> %1:X
q' --> %1:X
y --> %2:X

Unifier #2
rewrites: 15 in 5ms cpu (6ms real) (2632 rewrites/second)
q --> %1:X %2:NeList %3:X
x --> %1:X
q' --> %1:X %2:NeList
y --> %3:X
```

It is enough to show that the normalized substitution instances by unifiers #1 and #2 of the disequalities are unsatisfiable constructor variants of: (i) $first(q) \neq x$ and (ii) $last(q) \neq y$. For (i) this follows from

```
Maude> red first(#1:X #2:X) .
result X: #1:X
```

yielding an unsatisfiable constructor disequality summarized as: $x_1 \neq x_1$, and

```
Maude> red first(#1:X #2:NeList #3:X) .
result X: #1:X
```

yielding an unsatisfiable constructor disequality summarized as: $x_1 \neq x_1$.
For (ii) this follows from:

```
Maude> red last(#1:X #2:X) .
result X: #2:X
```

yielding an unsatisfiable constructor disequality summarized as: $x_2 \neq x_2$, and

```
Maude> red last(#1:X #2:NeSt #3:X) .
result X: #3:X
```

yielding an unsatisfiable constructor disequality summarized as: $x_3 \neq x_3$.

3.2 Variant Satisfiability of $\mathbf{St}^{\in}[X]$

In the case of the parameterized module $\mathbf{St}^{\in}[X]$, we cannot rely anymore on the preservation of weak OS-Compactness provided by Theorem 3. But we still have properties (1)–(4). This means that it is sound to still follow the same approach as for $\mathbf{St}[X]$, but we may have somewhat weaker chances of success because, besides having to cope a fortiori with the fact that variant unification using $IUnif_{A \cup B_{\mathbf{M}}}$ as our underlying mechanism will in general be incomplete, we need to face the additional potential source of incompleteness associated with not being able in general to compute complete finite sets of variants/constructor variants. However, if we do not get any warnings from $IUnif_{A \cup B_{\mathbf{M}}}$ when performing either task, we can still be able to decide the initial satisfiability of the given QF formula. This is the case for two reasons: (A). Since $\mathbf{St}^{\in}[X]$ only adds the new sort *Pred* to $\mathbf{St}[X]$ and the predicates do not affect at all the terms in the original sort, it is easy to show that $\mathbf{St}^{\in}[X]$ *protects* $\mathbf{St}[X]$. Therefore, if we instantiate $\mathbf{St}^{\in}[X]$ to $\mathbf{St}^{\in}[\mathbf{M},X \mapsto s]$ with $\vec{\mathbf{M}}$ and s satisfying the assumptions in Theorem 3, $\mathbf{St}[\mathbf{M},X \mapsto s]$ will still be FVP, and Theorem 3 will still apply to the reduct $\mathbf{St}_{\Omega}[\mathbf{M}_{\Omega},X \mapsto s]$ of $\mathbf{St}^{\in}_{\Omega \cup Pi}[\mathbf{M}_{\Omega},X \mapsto s]$. (B). But (A) means that,

since the "data" as opposed to the "predicate" part has not been affected, given a conjunction $\bigwedge G \wedge \bigwedge D$ for which we have been able to compute some variant unifiers of $\bigwedge G$ we can still compute a complete set of constructor variants for the "data disequalies" in $\bigwedge D$. However, we may or may not be able to compute a complete set of constructor variants for the "predicate disequalities". But for any canonical and axiom-consistent constructor variant, say, $\bigwedge D'$, of $(\bigwedge D)\theta$ with variables only of infinite sorts associated to any unifier θ of $\bigwedge G$ that we may have obtained, we can soundly *conclude* that the original conjunction $\bigwedge G \wedge \bigwedge D$ is satisfiable in the initial algebra. The reason why this is so is because, as in the proof of Theorem 3: (i) we can first *reduce* the initial satisfiability of $\bigwedge D'$ to that of the substitution $\bigwedge D'\{\overline{Y} \mapsto \overline{y}\}$ explained in the proof of Theorem 3, which will also be in canonical form and axiom-consistent; (ii) we can then transform the "data disequalities" in $\bigwedge D'\{\overline{Y} \mapsto \overline{y}\}$ into corresponding canonical and axiom-consistent $\mathbf{M}_{\Omega}$-disequalities as in the proof of Theorem 3; and (iii) we can do so also for the "predicate disequalities", which must be of one of the following two forms: (a) $t \in \varepsilon \neq tt$, which is always valid and can be disregarded; or (b) $t \in t_1 \ldots t_n \neq tt$, with $n \geqslant 1$, which is semantically equivalent to the conjunction of $\mathbf{M}_{\Omega}$-disequalities $t \neq t_1 \wedge \ldots \wedge t \neq t_n$. In this way, we can reduce $\bigwedge D'\{\overline{Y} \mapsto \overline{y}\}$ to an equivalent conjunction of canonical and axiom-consistent disequalities, showing that our original $\bigwedge D'$ was satisfiable in the initial algebra and therefore the decision method is sound. Of course, as in the $\mathbf{St}[X]$ case, if at any point in the entire process of computing unifiers θ for $\bigwedge G$, and then constructor variants with only infinite sort variables for $(\bigwedge D)\theta$, we get an incompleteness warning from the underlying $IUnif_{A \cup B_{\mathbf{M}}}$ algorithm and we have not been able to find a canonical and axiom-consistent constructor disequality variant, we must answer "don't know."

Let us see an example illustrating two facts: (i) that by performing instantiations $\mathbf{St}^{\in}[\mathbf{M}, X \mapsto s]$ with $\vec{\mathbf{M}}$ and s satisfying the assumptions in Theorem 3, we obtain for free a kind of implicit "Nelson-Oppen" combination of the variant satisfiability available for $\mathbf{M}$ and that for $\mathbf{St}^{\in}[X]$, and (ii) that the instantiation process can be nested, either with some other variant-satisfiable parameterized module or with $\mathbf{St}^{\in}[X]$ itself. We can consider, for example, a nested instantiation of the form $\mathbf{St}^{\in}[\mathbf{St}^{\in}[\mathbf{N}, X \mapsto Nat]X \mapsto NeSt']$ describing lists of lists of natural numbers. To ensure syntax disjointness, in the inner instantiation all sorts and operators must be renamed to, say, a primed form, except that we can rename _ _ to _;_. Also, to ensure that no identity axioms are used, $\mathbf{N}$ is assumed to be a specification of Presburger arithmetic where $+$ is AC, which can be specified in a way similar to the one used in [16], except that we use a subsort relation $NzNat < Nat$, specify $+$ in both sorts, but declare it as a constructor in $NzNat$, and declare 0 of sort Nat, 1 of sort $NzNat$, declare the rewrite rule $n + 0 \rightarrow n$ as well as those defining, say $>$ as a Boolean predicate.

Here is a somewhat amusing, domain-specific theorem that is valid in the initial algebra of $\mathbf{St}^{\in}[\mathbf{St}^{\in}[\mathbf{N}, X \mapsto Nat]X \mapsto NeSt']$:

$$(first'(first(q)) > 0 = true \wedge first'(first(q)) = first(q)) \Rightarrow (0 \in' first(q) \neq tt)$$

This property can be shown valid for the data type of strings of strings of natural numbers because Maude cannot find any variant unifiers for its negation, that is, for the conjunction

$$first'(first(q)) > 0 = true \wedge first'(first(q)) = first(q) \wedge 0 \in' first(q) \neq tt$$

Intuitively, the theorem says that, for q a non-empty string of non-empty strings of natural numbers, if the first element of its first element is greater than 0 and $first'(first(q)) = first(q)$, then 0 cannot appear in the string of natural numbers $first(q)$.

4 Related Work and Conclusions

Three types of most closely related work are: (1) Decision procedures for strings in current SMT solvers, e.g., [18, 1, 19, 11]. As pointed out in the Introduction, they are parametric on a finite alphabet, whereas $\mathbf{St}[X]$ and $\mathbf{St}^{\in}[X]$ are parametric on an infinite user-definable algebraic data type of string elements. They cannot be compared directly with each other: they cover complementary cases and different applications. An advantage of $\mathbf{St}[X]$ and $\mathbf{St}^{\in}[X]$ is that they are directly suited to reason about the initial models of order-sorted user-defined data types *and* their composition by both importations an instantiations of parametric data types. This can be very useful when reasoning about the correctness of declarative rule-based programs and in many other applications. (2) Previous work on variant satisfiability, e.g., [16, 17]. In comparison with that work, what this work adds is to bring data types with associative but not commutative axioms, including parameterized ones, within the fold of variant satisfiability. (3) The recent work by Steven Eker in defining and building his very efficient and practical and order-sorted A- and B-unification algorithm $A \subseteq B$ [7, 5], here denoted $IUnif_B$, has been crucial for this work.

Much work remains ahead. Just the definition of $\mathbf{St}[X]$ and $\mathbf{St}^{\in}[X]$, their properties, theoretical framework needed, and some simple experiments is what has been possible to do for the moment. Two items for near-term work include the optimization and integration of $\mathbf{St}[X]$ and $\mathbf{St}^{\in}[X]$ within the current variant satisfiablity prototype tool [17], and substantial experimentation and exploration of synergies with the traditional SMT-based string procedures.

Acknowledgements. I cordially thank the referees for their very helpful suggestions to improve the paper. This work has been partially supported by NRL under contract N00173-17-1-G002.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 462–469. Springer (2015)
2. Cholewa, A., Meseguer, J., Escobar, S.: Variants of variants and the finite variant property. Tech. rep., CS Dept. University of Illinois at Urbana-Champaign (February 2014), available at `http://hdl.handle.net/2142/47117`
3. Comon-Lundth, H., Delaune, S.: The finite variant property: how to get rid of some algebraic properties, in Proc *RTA'05*, Springer LNCS 3467, 294–307, 2005
4. Dershowitz, N., Jouannaud, J.P.: Rewrite systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Vol. B, pp. 243–320. North-Holland (1990)
5. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.L.: Programming and symbolic computation in Maude. J. Log. Algebr. Meth. Program. **110** (2020). https://doi.org/10.1016/j.jlamp.2019.100497, https://doi.org/10.1016/j.jlamp.2019.100497
6. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: Associative unification and symbolic reasoning modulo associativity in Maude. In: Rusu, V. (ed.) Proc. Rewriting Logic and Its Applications - 12th International Workshop, WRLA 2018. Lecture Notes in Computer Science, vol. 11152, pp. 98–114. Springer (2018)
7. Eker, S.: A pragmatic approach to implementing associative unification, unpublished manuscript, SRI International, circa 2015
8. Escobar, S., Sasse, R., Meseguer, J.: Folding variant narrowing and optimal variant termination. J. Algebraic and Logic Programming **81**, 898–928 (2012)
9. 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)

10. Jouannaud, J.P., Kirchner, C., Kirchner, H.: Incremental construction of unification algorithms in equational theories. In: Proc. ICALP'83. pp. 361–373. Springer LNCS 154 (1983)
11. Liang, T., Tsiskaridze, N., Reynolds, A., Tinelli, C., Barrett, C.W.: A decision procedure for regular membership and length constraints over unbounded strings. In: Proc. FroCoS 2015. Lecture Notes in Computer Science, vol. 9322, pp. 135–150. Springer (2015)
12. Meseguer, J.: Variants in the infinitary unification wonderland, submitted to WRLA 2020
13. Meseguer, J.: Order-sorted parameterization and induction. In: Semantics and Algebraic Specification. Lecture Notes in Computer Science, vol. 5700, pp. 43–80. Springer (2009)
14. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)
15. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Proc. WADT'97. pp. 18–61. Springer LNCS 1376 (1998)
16. Meseguer, J.: Variant-based satisfiability in initial algebras. Sci. Comput. Program. **154**, 3–41 (2018)
17. Skeirik, S., Meseguer, J.: Metalevel algorithms for variant satisfiability. J. Log. Algebr. Meth. Program. **96**, 81–110 (2018)
18. Trinh, M., Chu, D., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1232–1243 (2014)
19. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Berzish, M., Dolby, J., Zhang, X.: Z3str2: an efficient solver for strings, regular expressions, and length constraints. Formal Methods in System Design **50**(2-3), 249–288 (2017)

Inductive Reasoning with Equality Predicates, Contextual Rewriting and Variant-Based Simplification

José Meseguer and Stephen Skeirik

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

Abstract. We present an inductive inference system for proving validity of formulas in the initial algebra $T_{\mathcal{E}}$ of an order-sorted equational theory $\mathcal{E}$ with 17 inference rules, where only 6 of them require user interaction, while the remaining 11 can be automated as *simplification rules* and can be combined together as a limited, yet practical, automated inductive theorem prover. The 11 simplification rules are based on powerful equational reasoning techniques, including: equationally defined equality predicates, constructor variant unification, variant satisfiability, order-sorted congruence closure, contextual rewriting and recursive path orderings. For $\mathcal{E} = (\Sigma, E \uplus B)$, these techniques work modulo B, with B a combination of associativity and/or commutativity and/or identity axioms.

1 Introduction

In inductive theorem proving for equational specifications there is a tension between automated approaches and explicit induction ones. For two examples of automated inductive provers we can mention, among various others, Spike [1] and the superposition-based "inductionless induction" prover in [5]; and for explicit induction provers we can mention, again among various others, RRL [14] and the Maude ITP [4, 13]. The advantage of automated provers is that they do not need interaction, although they often require proving auxiliary lemmas. Explicit induction is less automated, but provides substantial flexibility. This work presents an approach that combines automated and explicit-induction theorem proving in the context of proving validity in the initial algebra $T_{\mathcal{E}}$ of an order-sorted equational theory $\mathcal{E}$ for both arbitrary quantifier-free (QF) formulas (expressed as conjunctions of clauses, some of which can be combined together as "superclauses" in the sense of Section 3.1) and existential closures of such clauses/superclauses.

The combination is achieved by an inference system having 17 inference rules, where 11 of them are *goal simplification rules* that can be fully automated, whereas the remaining 6 require explicit user commands. In fact, we have combined 9 of those simplification rules into an automated inductive simplification strategy that we call *ISS*. An even more powerful *ISS*$^+$ combining all the 11 simplification rules could likewise be developed. Because the simplification rules are very powerful, *ISS* can be used on its own as an automatic *oracle* to answer inductive validity questions, that is, as a limited, yet quite practical, automated inductive theorem prover. How practical? As we explain in Section 4.3, practical enough to prove *all* the thousands of inductive validity verification conditions (VCs) that were generated in the deductive verification proof in constructor-based reachability logic of the security properties of the IBOS Browser described in [27]. It was the remarkable effectiveness of (a simplified version of) *ISS* as a backend oracle in the IBOS proof that gave us the stimulus for this work.

So, what is the secret of such effectiveness? There isn't a secret as such, but a novel *combination* of powerful *automatable* equational reasoning techniques that, to the best of our knowledge,

have never before been combined together for inductive theorem proving purposes. They include: (1) equationally defined equality predicates [12]; (2) constructor variant unification [20, 28]; (3) variant satisfiability [20, 28]; (4) order-sorted congruence closure [19]; (5) contextual rewriting [31]; and (6) recursive path orderings [23, 11]. All these techniques work modulo axioms B.

2 Preliminaries

2.1 Background on Order-Sorted First-Order Logic

We assume familiarity with the notions of an order-sorted signature Σ on a poset of sorts $(S,\leqslant)$, an order-sorted Σ-algebra A, and the term Σ-algebras T_Σ and $T_\Sigma(X)$ for X an S-sorted set of variables. We also assume familiarity with the notions of: (i) Σ-homomorphism $h : A \to B$ between Σ-algebras A and B, so that Σ-algebras and Σ-homomorphisms form a category $\mathbf{OSAlg}_\Sigma$; (ii) order-sorted (i.e., sort-preserving) substitution θ, its domain $dom(\theta)$ and range $ran(\theta)$, and its application $t\theta$ to a term t; (iii) *preregular* order-sorted signature Σ, i.e., a signature such that each term t has a least sort, denoted $ls(t)$; (iv) the set $\widehat{S} = S/(\geqslant \cup \leqslant)^+$ of *connected components* of $(S,\leqslant)$; and (v) for A a Σ-algebra, the set A_s of it elements of sort $s \in S$, and the set $A_{[s]} = \bigcup_{s' \in [s]} A_{s'}$ for $[s] \in \widehat{S}$. We furthermore assume that all signatures Σ have *non-empty sorts*, i.e., $T_{\Sigma,s} \neq \emptyset$ for each $s \in S$. All these notions are explained in detail in [18, 10]. The material below is adapted from [20].

The first-order language of *equational Σ-formulas* is defined in the usual way: its atoms[1] are *Σ-equations* $t = t'$, where $t, t' \in T_\Sigma(X)_{[s]}$ for some $[s] \in \widehat{S}$ and each X_s is assumed countably infinite. The set $Form(\Sigma)$ of *equational Σ-formulas* is then inductively built from atoms by: conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and universal ($\forall x_1 : s_1,...,x_n : s_n$) and existential ($\exists x_1 : s_1,...,x_n : s_n$) quantification with distinct sorted variables $x_1 : s_1,...,x_n : s_n$, with $s_1,\ldots,s_n \in S$ (by convention, for $\emptyset$ the empty set of variables and φ a formula, we define $(\forall \emptyset)\, \varphi \equiv (\exists \emptyset)\, \varphi \equiv \varphi$). A literal $\neg(t = t')$ is denoted $t \neq t'$. Given a Σ-algebra A, a formula $\varphi \in Form(\Sigma)$, and an assignment $\alpha \in [Y \to A]$, where $Y \supseteq fvars(\varphi)$, with $fvars(\varphi)$ the free variables of φ, the *satisfaction relation* $A,\alpha \models \varphi$ is defined inductively as usual: for atoms, $A,\alpha \models t = t'$ iff $t\alpha = t'\alpha$; for Boolean connectives it is the corresponding Boolean combination of the satisfaction relations for subformulas; and for quantifiers: $A,\alpha \models (\forall x_1 : s_1,...,x_n : s_n)\, \varphi$ (resp. $A,\alpha \models (\exists x_1 : s_1,\ldots,x_n : s_n)\, \varphi$) holds iff for all $(a_1,\ldots,a_n) \in A_{s_1} \times \ldots \times A_{s_n}$ (resp. for some $(a_1,\ldots,a_n) \in A_{s_1} \times \ldots \times A_{s_n}$) we have $A,\alpha[x_1 : s_1 := a_1,\ldots,x_n : s_n := a_n] \models \varphi$, where if $\alpha \in [Y \to A]$, then $\alpha[x_1 : s_1 := a_1,...,x_n : s_n := a_n] \in [(Y \cup \{x_1 : s_1,...,x_n : s_n\}) \to A]$ and is such that for $y : s \in (Y \setminus \{x_1 : s_1,...,x_n : s_n\})$, $\alpha[x_1 : s_1 := a_1,...,x_n : s_n := a_n](y : s) = \alpha(y : s)$, and $\alpha[x_1 : s_1 := a_1,...,x_n : s_n := a_n](x_i : s_i) = a_i$, $1 \leqslant i \leqslant n$. We say that φ is *valid* in A (resp. is *satisfiable* in A) iff $A,\underline{\emptyset} \models (\forall Y)\, \varphi$ (resp. $A,\underline{\emptyset} \models (\exists Y)\, \varphi$), where $Y = fvars(\varphi)$ and $\underline{\emptyset} \in [\underline{\emptyset} \to A]$ denotes the empty S-sorted assignment of values in A to the empty S-sorted family $\underline{\emptyset}$ of variables. The notation $A \models \varphi$ abbreviates validity of φ in A. More generally, a set of formulas $\Gamma \subseteq Form(\Sigma)$ is called *valid* in A, denoted $A \models \Gamma$, iff $A \models \varphi$ for each $\varphi \in \Gamma$. For a subsignature $\Omega \subseteq \Sigma$ and $A \in \mathbf{OSAlg}_\Sigma$, the *reduct* $A|_\Omega \in \mathbf{OSAlg}_\Omega$ agrees with A in the interpretation of all sorts and operations in Ω and discards everything in $\Sigma \setminus \Omega$. If $\varphi \in Form(\Omega)$ we have the equivalence $A \models \varphi \Leftrightarrow A|_\Omega \models \varphi$.

An OS *equational theory* is a pair $T = (\Sigma, E)$, with E a set of (possibly conditional) Σ-equations. $\mathbf{OSAlg}_{(\Sigma,E)}$ denotes the full subcategory of $\mathbf{OSAlg}_\Sigma$ with objects those $A \in \mathbf{OSAlg}_\Sigma$

[1] As explained in [20], there is no real loss of generality in assuming that all atomic formulas are equations: predicates can be specified by equational formulas using additional function symbols of a fresh new sort *Pred* with a constant *tt*, so that a predicate $p(t_1,...,t_n)$ becomes $p(t_1,...,t_n) = tt$.

such that $A \models E$, called the (Σ,E)-*algebras*. $\mathbf{OSAlg}_{(\Sigma,E)}$ has an *initial algebra* $T_{\Sigma/E}$ [18]. Given $T = (\Sigma,E)$ and $\varphi \in Form(\Sigma)$, we call φ *T-valid*, written $E \models \varphi$, iff $A \models \varphi$ for all $A \in \mathbf{OSAlg}_{(\Sigma,E)}$. We call φ *T-satisfiable* iff there exists $A \in \mathbf{OSAlg}_{(\Sigma,E)}$ with φ satisfiable in A. Note that φ is *T-valid* iff $\neg\varphi$ is *T-unsatisfiable*. The inference system in [18] is *sound and complete* for OS equational deduction, i.e., for any OS equational theory (Σ,E), and Σ-equation $u = v$ we have an equivalence $E \vdash u = v \;\Leftrightarrow\; E \models u = v$. Deducibility $E \vdash u = v$ is abbreviated as $u =_E v$, called *E-equality*. An *E-unifier* of a system of Σ-equations, i.e., of a conjunction $\phi = u_1 = v_1 \wedge ... \wedge u_n = v_n$ of Σ-equations, is a substitution σ such that $u_i\sigma =_E v_i\sigma$, $1 \leqslant i \leqslant n$. An *E-unification algorithm* for (Σ,E) is an algorithm generating a *complete set* of E-unifiers $Unif_E(\phi)$ for any system of Σ equations ϕ, where "complete" means that for any E-unifier σ of ϕ there is a $\tau \in Unif_E(\phi)$ and a substitution ρ such that $\sigma =_E (\tau\rho)|_{dom(\sigma)\cup dom(\tau)}$, where $=_E$ here means that for any variable x we have $x\sigma =_E x(\tau\rho)|_{dom(\sigma)\cup dom(\tau)}$. The algorithm is *finitary* if it always terminates with a *finite set* $Unif_E(\phi)$ for any ϕ.

Given a set of equations B used for deduction modulo B, a preregular OS signature Σ is called *B-preregular*[2] iff for each $u = v \in B$ and substitutions ρ, $ls(u\rho) = ls(v\rho)$.

2.2 Background on Convergent Theories and Constructors

Given an order-sorted equational theory $\mathcal{E} = (\Sigma,E \cup B)$, where B is a collection of associativity and/or commutativity and/or identity axioms and Σ is B-preregular, we can associate to it a corresponding *rewrite theory* [17] $\vec{\mathcal{E}} = (\Sigma,B,\vec{E})$ by orienting the equations E as left-to right rewrite rules. That is, each $(u = v) \in E$ is transformed into a rewrite rule $u \to v$. For simplicity we recall here the case of unconditional equations; for how conditional equations (whose conditions are conjunctions of equalities) are likewise transformed into conditional rewrite rules see, e.g., [15]. The main purpose of the rewrite theory $\vec{\mathcal{E}}$ is to reduce the complex bidirectional reasoning with equations to the much simpler unidirectional reasoning with rules under suitable assumptions. We assume familiarity with the notion of subterm $t|_p$ of t at a term position p and of term replacement $t[w]_p$ of $t|_p$ by w at position p (see, e.g., [6]). The rewrite relation $t \to_{\vec{E},B} t'$ holds iff there is a subterm $t|_p$ of t, a rule $(u \to v) \in \vec{E}$ and a substitution θ such that $u\theta =_B t|_p$, and $t' = t[v\theta]_p$. We denote by $\to^*_{\vec{E},B}$ the reflexive-transitive closure of $\to_{\vec{E},B}$. The requirements on $\vec{\mathcal{E}}$ allowing us to reduce equational reasoning to rewriting are the following: (i) $vars(v) \subseteq vars(u)$; (ii) *sort-decreasingness*: for each substitution θ we must have $ls(u\theta) \geqslant ls(v\theta)$; (iii) *strict B-coherence*: if $t_1 \to_{\vec{E},B} t'_1$ and $t_1 =_B t_2$ then there exists $t_2 \to_{\vec{E},B} t'_2$ with $t'_1 =_B t'_2$; (iv) *confluence* (resp. *ground confluence*) modulo B: for each term t (resp. ground term t) if $t \to^*_{\vec{E},B} v_1$ and $t \to^*_{\vec{E},B} v_2$, then there exist rewrite sequences $v_1 \to^*_{\vec{E},B} w_1$ and $v_2 \to^*_{\vec{E},B} w_2$ such that $w_1 =_B w_2$; (v) *termination*: the relation $\to_{\vec{E},B}$ is well-founded (for $\vec{E}$ conditional, we require *operational termination* [15]). If $\vec{\mathcal{E}}$ satisfies conditions (i)–(v) (resp. the same, but (iv) weakened to ground confluence modulo B), then it is called *convergent* (resp. *ground convergent*). The key point is that then, given a term (resp. ground term) t, all terminating rewrite sequences $t \to^*_{\vec{E},B} w$ end in a term w, denoted $t!_{\vec{\mathcal{E}}}$, that is unique up to B-equality, and its called t's *canonical form*. Three major results then follow for the ground convergent case: (1) for any ground terms t,t' we

[2] If $B = B_0 \uplus U$, with B_0 associativity and/or commutativity axioms, and U identity axioms, the B-preregularity notion can be *broadened* by requiring only that: (i) Σ is B_0-preregular in the standard sense that $ls(u\rho) = ls(v\rho)$ for all $u = v \in B_0$ and substitutions ρ; and (ii) the axioms U oriented as rules $\vec{U}$ are *sort-decreasing* in the sense explained in Section 2.2.

have $t =_{E \cup B} t'$ iff $t!_{\vec{\mathcal{E}}} =_B t'!_{\vec{\mathcal{E}}}$, (2) the B-equivalence classes of canonical forms are the elements of the *canonical term algebra* $C_{\Sigma/E,B}$, where for each $f : s_1 \ldots s_n \to s$ in Σ and B-equivalence classes of canonical terms $[t_1],\ldots,[t_n]$ with $ls(t_i) \leqslant s_i$ the operation $f_{C_{\Sigma/E,B}}$ is defined by the identity: $f_{C_{\Sigma/E,B}}([t_1]\ldots[t_n]) = [f(t_1 \ldots t_n)!_{\vec{\mathcal{E}}}]$, and (3) we have an isomorphism $T_{\mathcal{E}} \cong C_{\Sigma/E,B}$.

A ground convergent rewrite theory $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ is called *sufficiently complete* with respect to a subsignature Ω, whose operators are then called *constructors*, iff for each ground Σ-term t, $t!_{\vec{\mathcal{E}}} \in T_\Omega$. Furthermore, for $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ sufficiently complete w.r.t. Ω, a ground convergent rewrite subtheory $(\Omega, B_\Omega, \vec{E}_\Omega) \subseteq (\Sigma, B, \vec{E})$ is called a *constructor subspecification* iff $T_{\mathcal{E}}|_\Omega \cong T_{\Omega/E_\Omega \cup B_\Omega}$. If $E_\Omega = \emptyset$, then Ω is called a signature of *free constructors modulo axioms* B_Ω.

2.3 Equationally Defined Equality Predicates in a Nutshell

Equationally-defined equality predicates [12] achieve a remarkable feat for QF formulas in initial algebras under reasonable executability conditions: they *reduce* first-order logic satisfaction of QF formulas in an initial algebra $T_{\mathcal{E}}$ to *purely equational reasoning*. This is achieved by a theory transformation[3] $\mathcal{E} \mapsto \mathcal{E}^=$ such that, provided: $\mathcal{E} = (\Sigma, E \cup B)$, with B any combination of associativity and/or commutativity axioms, is ground convergent and operationally terminating modulo B, and is sufficiently complete with respect to a subsignature Ω of constructors such that $T_{\mathcal{E}}|_\Omega \cong T_{\Omega/B_\Omega}$, with $B_\Omega \subseteq B$, then: (i) $\mathcal{E}^=$ is ground convergent operationally terminating and sufficiently complete and protects[4] a new copy of the Booleans, of sort *NewBool*, where true and false are respectively denoted $\top$, $\bot$, conjunction and disjunction are respectively denoted $\wedge$, $\vee$, negation is denoted $\neg$, and a QF Σ-formula φ is a term of sort *NewBool*. Furthermore, for any ground QF Σ-formula φ we have:

$$T_{\mathcal{E}} \models \varphi \Leftrightarrow \varphi!_{\vec{\mathcal{E}}^=} = \top \quad \textit{and} \quad T_{\mathcal{E}} \not\models \varphi \Leftrightarrow \varphi!_{\vec{\mathcal{E}}^=} = \bot.$$

That is, we can decide the validity of ground QF Σ-formulas in $T_{\mathcal{E}}$ by reducing them to canonical form with the ground convergent rules in $\vec{\mathcal{E}}^=$. In particular, and this is the property that we will systematically exploit in Section 3.2, for any QF Σ-formula φ, possibly with variables, we have $T_{\mathcal{E}} \models (\varphi \Leftrightarrow \varphi!_{\vec{\mathcal{E}}^=})$, where $\varphi!_{\vec{\mathcal{E}}^=}$ may be a much simpler formula, sometimes just $\top$ or $\bot$. Since the $\mathcal{E} \mapsto \mathcal{E}^=$ excludes identity axioms from $\mathcal{E}$, one lingering doubt is what to do when $\mathcal{E}$ has also identity axioms U. The answer is that we can use the semantics-preserving theory transformation $\mathcal{E} \mapsto \mathcal{E}_U$ defined in [7], which turns U into rules $\vec{U}$ and preserves ground convergence, to reduce to the case $U = \emptyset$, provided we have $T_{\mathcal{E}_U}|_\Omega \cong T_{\Omega/B_\Omega}$.

2.4 Order-Sorted Congruence Closure in a Nutshell

Let (Σ, B) be an order-sorted theory where the axioms B are only associativity-commutativity (AC) axioms and Σ is B-preregular. Now let Γ be a set of ground Σ-equations. The question is: is $B \cup \Gamma$-equality *decidable*? (when Σ has just a binary AC operator, this is called the "word problem for commutative semigroups"). The answer, provided in [19], is yes! We can perform a ground Knuth-Bendix completion of Γ into an equivalent (modulo B) set of ground rewrite rules

[3] In [12] the equality predicate is denoted $\_ \sim \_$, instead of the standard notation $\_ = \_$. Here we use $\_ = \_$ throughout. This has the pleasant effect that a QF formula φ is both a formula and a Boolean expression, which of course amounts to mechanizing by equational rewriting the Tarskian semantics of QF formulas in first-order-logic for initial algebras.

[4] That is, there is a subtheory inclusion $\mathcal{B} \subseteq \mathcal{E}$, with $\mathcal{B}$ having signature $\Sigma_{\mathcal{B}}$ and only sort *NewBool* such that: (i) $T_{\mathcal{B}}$ the initial algebra of the Booleans, and (ii) $T_{\mathcal{E}^=}|_{\Sigma_{\mathcal{B}}} \cong T_{\mathcal{B}}$.

$cc^{\succ}_B(\Gamma)$ that is convergent modulo B, so that $t =_{B \cup \Gamma} t'$ iff $t!_{\vec{\mathcal{E}}_{cc^{\succ}_B(\Gamma)}} =_{B^{\square}} t'!_{\vec{\mathcal{E}}_{cc^{\succ}_B(\Gamma)}}$, where $\vec{\mathcal{E}}_{cc^{\succ}_B(\Gamma)}$ is the rewrite theory $\vec{\mathcal{E}}_{cc^{\succ}_B(\Gamma)} = (\Sigma^{\square}, B^{\square}, cc^{\succ}_B(\Gamma))$, with $\Sigma^{\square}$ the "kind completion" of Σ, which is automatically computed by Maude by adding a so-called "kind" sort $\top_{[s]}$ above each connected component $[s] \in \widehat{S}$ of $(S, \leqslant)$ and lifting each operation $f : s_1 \cdots s_n \to s$ to its kinded version $f : \top_{[s_1]} \cdots \top_{[s_n]} \to \top_{[s]}$, and where $B^{\square}$ is obtained from B by replacing each variable of sort s in B by a corresponding variable of sort $\top_{[s]}$. The symbol $\succ$ in $cc^{\succ}_B(\Gamma)$ is a total well-founded order on ground terms modulo B that is used to orient the equations into rules. In all our uses we will take $\succ$ to be an AC RPO based on a total order on function symbols [24]. The need to extend Σ to $\Sigma^{\square}$ is due to the fact that some terms in $cc^{\succ}_B(\Gamma)$ may be $\Sigma^{\square}$-terms that fail to be Σ-terms.

Extending the above congruence closure framework from AC axioms B to axioms B that contain any combination of associativity and/or commutativity axioms is quite smooth, but requires a crucial caveat: if some operator $f \in \Sigma$ is only associative, then $cc^{\succ}_B(\Gamma)$ may be an infinite set that cannot be computed in practice. This is due to the undecidability of the "word problem for semigroups." The Maude implementation of $cc^{\succ}_B(\Gamma)$ used in Section 4 supports this more general combination of axioms B, but when some $f \in \Sigma$ is only associative, it has a bound on the number of iterations of the ground completion cycle. This of course means that, if the completion process has not terminated before the bound is reached, the above decidability result does not hold. However, for our inductive simplification purposes it is enough to obtain a set of ground rules $cc^{\succ}_B(\Gamma)$ that is guaranteed to be *terminating* modulo B, and that, thanks to the, perhaps partial, completion, "approximates convergence" much better than the original Γ.

2.5 Contextual Rewriting in a Nutshell

Let (Σ, B) be an order-sorted theory where the axioms B contain any combination of associativity and/or commutativity axioms. What can we do to prove that in (Σ, B) an implication of the form $\Gamma \to u = v$, with variables $vars(\Gamma \to u = v) = X$ and Γ a conjunction of equations, is valid? We can: (i) add to Σ a set of fresh new constants $\overline{X}$ obtained from X by changing each $x \in X$ into a constant $\overline{x} \in \overline{X}$ of same sort as x, (ii) replace the conjunction Γ by the ground conjunction $\overline{\Gamma}$ obtained by replacing each $x \in X$ in Γ by its corresponding $\overline{x} \in \overline{X}$, and obtaining likewise the ground equation $\overline{u} = \overline{v}$. By the Lemma of Constants and the Deduction Theorem we have [18]:

$$(\Sigma, B) \vdash \Gamma \to u = v \;\Leftrightarrow\; (\Sigma(\overline{X}), \cup B \cup \{\overline{\Gamma}\}) \vdash \overline{u} = \overline{v}$$

where $\Sigma(\overline{X})$ is obtained from Σ by adding the fresh new constants $\overline{X}$, and $\{\overline{\Gamma}\}$ denotes the set of ground equations associated to the conjunction $\overline{\Gamma}$. But, disregarding the difference between $\overline{\Gamma}$ and $\{\overline{\Gamma}\}$, and realizing that $cc^{\succ}_B(\overline{\Gamma})$ is equivalent modulo $B^{\square}$ to $\overline{\Gamma}$, if we can prove $\overline{u}!_{\vec{\mathcal{E}}_{cc^{\succ}_B(\overline{\Gamma})}} =_{B^{\square}} \overline{v}!_{\vec{\mathcal{E}}_{cc^{\succ}_B(\overline{\Gamma})}}$, then we have proved $(\Sigma(\overline{X}), \cup B \cup \{\overline{\Gamma}\}) \vdash \overline{u} = \overline{v}$ and therefore $(\Sigma, B) \vdash \Gamma \to u = v$, where $\vec{\mathcal{E}}_{cc^{\succ}_B(\overline{\Gamma})} = (\Sigma(\overline{X})^{\square}, B^{\square}, cc^{\succ}_B(\overline{\Gamma}))$. Furthermore, if $\vec{\mathcal{E}}_{cc^{\succ}_B(\overline{\Gamma})}$ is *convergent* (this may only fail to be the case if some $f \in \Sigma$ is associative but not commutative) this is an *equivalence*: $(\Sigma, B) \vdash \Gamma \to u = v$ iff $\overline{u}!_{\vec{\mathcal{E}}_{cc^{\succ}_B(\overline{\Gamma})}} =_{B^{\square}} \overline{v}!_{\vec{\mathcal{E}}_{cc^{\succ}_B(\overline{\Gamma})}}$, and therefore a decision procedure. Rewriting with $\vec{\mathcal{E}}_{cc^{\succ}_B(\overline{\Gamma})}$ is called *contextual rewriting* [31], since we are using the "context" $\overline{\Gamma}$ suitably transformed into $cc^{\succ}_B(\overline{\Gamma})$. Many increasingly more powerful variations on this method are possible. For example, we may replace (Σ, B) by $\mathcal{E} = (\Sigma, E \cup B)$, with $\vec{\mathcal{E}}$ ground convergent and then rewrite $\overline{u} = \overline{v}$ not only with $cc^{\succ}_B(\overline{\Gamma})$ but also with $\vec{\mathcal{E}}$. Likewise, we may consider not just a ground equation $\overline{u} = \overline{v}$, but a ground QF formula $\overline{\varphi}$ and rewrite $\overline{\varphi}$ not only with $cc^{\succ}_B(\overline{\Gamma})$ but also with $\vec{\mathcal{E}}^{=}_U$.

2.6 Variant Unification and Satisfiability in a Nutshell

Consider an order-sorted equational theory $\mathcal{E}=(\Sigma,E\cup B)$ such that $\vec{\mathcal{E}}$ is ground convergent and suppose we have a constructor subspecification $(\Omega,B_\Omega,\emptyset)\subseteq(\Sigma,B,\vec{E})$, so that $T_{\mathcal{E}}|_\Omega\cong T_{\Omega/B_\Omega}$. Suppose, further, that we have a subtheory $\mathcal{E}_1\subseteq\mathcal{E}$ such that: (i) $\vec{\mathcal{E}}_1$ is convergent and has the finite variant property[5] (FVP) [9], (ii) can be "sandwiched" between $\vec{\mathcal{E}}$ and the constructors as $(\Omega,B_\Omega,\emptyset)\subseteq(\Sigma_1,B_1,\vec{E}_1)\subseteq(\Sigma,B,\vec{E})$, (iii) B_1 can involve any combination of associativity and/or commutativity and/or identity axioms, except associativity without commutativity; and (iv) $T_{\mathcal{E}}|_{\Sigma_1}\cong T_{\mathcal{E}_1}$, which forces $T_{\mathcal{E}_1}|_\Omega\cong T_{\Omega/B_\Omega}$.

Then, if Γ is a conjunction of Σ_1-equations, since $T_{\mathcal{E}}|_\Omega\cong T_{\Omega/B_\Omega}$, a ground $\mathcal{E}$-unifier ρ of Γ is always $\mathcal{E}$-equivalent to its normal form to $\rho!_{\vec{\mathcal{E}}}$, with $\rho!_{\vec{\mathcal{E}}}(x)=\rho(x)!_{\vec{\mathcal{E}}}$, which is a ground Ω-substitution, that is, a *constructor* ground $\mathcal{E}$-unifier of Γ. But since $\Omega\subseteq\Sigma_1$ and $T_{\mathcal{E}}|_{\Sigma_1}\cong T_{\mathcal{E}_1}$, which implies $C_{\Sigma/E,B}|_{\Sigma_1}\cong C_{\Sigma_1/E_1,B_1}$, this makes $\rho!_{\vec{\mathcal{E}}}$ a *constructor* ground $\mathcal{E}_1$-unifier of Γ. But, under the assumptions for B_Ω, by the results in [20, 28] we can compute a complete, finite set $Unif^{\Omega}_{\mathcal{E}_1}(\Gamma)$ of constructor $\mathcal{E}_1$-unifiers of Γ, so that any constructor ground $\mathcal{E}_1$-unifier of Γ, and therefore up to $\mathcal{E}$-equivalence any ground $\mathcal{E}$-unifier of Γ, is an instance of a unifier in $Unif^{\Omega}_{\mathcal{E}_1}(\Gamma)$.

Note, furthermore, that under the assumptions on B_1, (Ω,B_Ω) is an OS-compact theory [20]. Therefore, again by [20], satisfiability (and therefore validity) of any QF Σ_1-formula in $T_{\mathcal{E}_1}$, and by $T_{\mathcal{E}}|_{\Sigma_1}\cong T_{\mathcal{E}_1}$ also in $T_{\mathcal{E}}$, is *decidable*.

3 Superclause-Based Inductive Reasoning

3.1 Superclauses and Inductive Theories

Since predicate symbols can always be transformed into function symbols by adding a fresh new sort *Pred*, we can reduce all of order-sorted first-order logic to just reasoning about equational formulas whose only atoms are equations. Any quantifier-free formula ϕ can therefore be put in conjunctive normal form (CNF) as a conjunction of equational clauses $\phi\equiv\bigwedge_{i\in I}\Gamma_i\rightarrow\Delta_i$, where Γ_i, denoted $u_1=v_1,\cdots,u_n=v_n$ is a conjunction of equations $\bigwedge_{1\leqslant i\leqslant n}u_i=v_i$ and Δ_i, denoted $w_1=w'_1,\cdots,w_m=w'_m$ is a disjunction of equations $\bigvee_{1\leqslant k\leqslant m}w_k=w'_k$. In our inductive inference system, higher efficiency can be gained by applying the inference rules not to a single clause, but to a conjunction of related clauses sharing the same condition Γ. Thus, we will assume that all clauses $\{\Gamma\rightarrow\Delta_l\}_{l\in L}$ with the same condition Γ in the CNF of ϕ have been gathered together into a semantically equivalent formula of the form $\Gamma\rightarrow\bigwedge_{l\in L}\Delta_l$, which we call a *superclause*. We will use the notation Λ to abbreviate $\bigwedge_{l\in L}\Delta_l$. Therefore, Λ denotes a conjunction of disjunctions of equations. Superclauses, of course, generalize clauses, which generalize Horn clauses, which, in turn, generalize equations. Thus, superclauses give us a more general setting for inductive reasoning, because superclauses are more general formulas.

What is an *inductive theory*? In an order-sorted equational logic framework, the simplest possible inductive theories we can consider are order-sorted conditional equational theories $\mathcal{E}=(\Sigma,E\cup B)$, where E is a set of conditional equations (i.e. Horn clauses) of the form $u_1=v_1\wedge\cdots\wedge u_n=v_n\rightarrow w=w'$, and B is a set of equational axioms such as associativity and/or

[5] An $\vec{\mathcal{E}}_1$-*variant* (or $\vec{E}_1,B_1$-*variant*) of a Σ_1-term t is a pair (v,θ), where θ is a substitution in canonical form, i.e., $\theta=\theta!_{\vec{\mathcal{E}}_1}$, and $v=_{B_1}(t\theta)!_{\vec{\mathcal{E}}_1}$. $\vec{\mathcal{E}}_1$ is FVP iff any such t has a finite set of variants $\{(u_1,\alpha_1),...,(u_n,\alpha_n)\}$ which are "most general possible" in the precise sense that for any variant (v,θ) of t there exist i, $1\leqslant i\leqslant n$, and substitution γ such that: (i) $v=_{B_1}u_i\gamma$, and (ii) $\theta=_{B_1}\alpha_i\gamma$.

commutativity and/or identity. Inductive properties are then properties satisfied in the *initial algebra* $T_{\mathcal{E}}$ associated to $\mathcal{E}$. Note that this is exactly the initial semantics of functional modules in the Maude language. So, as a first approximation, a Maude user can think of an inductive theory as an order-sorted functional module. The problem, however, is that as we perform inductive reasoning, the inductive theory of $\mathcal{E}$, which we denote by $[\mathcal{E}]$ to emphasize its initial semantics, needs to be extended by: (i) extra constants, and; (ii) extra formulas such as: (a) induction hypotheses, (b) lemmas, and (c) hypotheses associated to modus ponens reasoning. Thus, we will consider general inductive theories of the form $[\overline{X},\mathcal{E},H]$, where $\overline{X}$ is a fresh set of constants having sorts in Σ and H is a set of $\Sigma(\overline{X})$ clauses[6], corresponding to formulas of types (a)-(c) above. The *models* of an inductive theory $[\overline{X},\mathcal{E},H]$ are exactly the $\Sigma^{\sqsubset}(\overline{X})$-algebras A such that $A|_{\Sigma^{\sqsubset}} \cong T_{\mathcal{E}^{\sqsubset}}$ and $A \models H$, where $\mathcal{E}^{\sqsubset} = (\Sigma^{\sqsubset}, E \cup B^{\sqsubset})$ and $\Sigma^{\sqsubset}$ is the kind completion of Σ defined in Section 2.4. Note that, since $T_{\mathcal{E}^{\sqsubset}}|_{\Sigma} = T_{\mathcal{E}}$ [19], the key relation for reasoning is $A|_{\Sigma} \cong T_{\mathcal{E}}$. Such algebras A have a very simple description: they are pairs $(T_{\mathcal{E}^{\sqsubset}}, a)$ where $a : \overline{X} \to T_{\mathcal{E}}$ is the assignment interpreting constants $\overline{X}$. In Maude, such inductive theories $[\overline{X},\mathcal{E},H]$ can be defined as *functional theories* which *protect* the functional module $[\mathcal{E}]$, which in our expanded notation is identified with the inductive theory $[\emptyset,\mathcal{E},\emptyset]$ with neither extra constants nor extra hypotheses, lemmas, or assumptions.

We will furthermore assume that $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$, with $B = B_0 \uplus U$, is ground convergent, with a total RPO order modulo $B_0 \succ$ making $\vec{\mathcal{E}}_U$ operationally terminating, and that there is a "sandwich" $(\Omega, B_\Omega, \emptyset) \subseteq (\Sigma_1, B_1, \vec{E}_1) \subseteq (\Sigma, B, \vec{E})$ with $B_\Omega \subseteq B_0$ satisfying all the requirements in Section 2.6, including the sufficient completeness of $\vec{\mathcal{E}}$ w.r.t. Ω, the finite variant property of $\mathcal{E}_1$, and the OS-compactness of (Ω, B_Ω).

We finally assume that the clauses H in an inductive theory $[\overline{X},\mathcal{E},H]$ have been decomposed into the disjoint union $H = H_e \cup H_{ne}$ where H_{e_U} are the *executable* hypotheses, which are conditional equations, that are *orientable* using the same RPO order $\succ$ modulo B_0 that makes $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ ground convergent, so $\vec{E}_U \cup \vec{H}_{e_U}$ is ground operationally terminating modulo B_0.

Under the above assumptions, up to isomorphism, and identifying $T_{\mathcal{E}^{\sqsubset}}$ with $C_{\Sigma^{\sqsubset}/E,B}$, a model (A,a) of an inductive theory $[\overline{X},\mathcal{E},H]$ has a very simple description as a pair $(T_{\mathcal{E}^{\sqsubset}}, [\overline{\alpha}])$, where $\alpha : X \to T_\Omega$ is a ground constructor substitution, $[\alpha]$ denotes the composition $X \xrightarrow{\alpha} T_\Omega \xrightarrow{[\_]} T_{\Omega/B_\Omega}$, with $[\_]$ the unique Ω-homomorphism mapping each term t to its B_Ω-equivalence class $[t]$, and where $[\overline{\alpha}] : \overline{X} \to T_{\mathcal{E}^{\sqsubset}}$ maps each $\overline{x} \in \overline{X}$ to $[\alpha](x)$, where $x \in X$ is the variable with same sort associated to $\overline{x} \in \overline{X}$. The fact that for each clause $\Gamma \to \Delta$ in H we must have $(T_{\mathcal{E}^{\sqsubset}}, [\overline{\alpha}]) \models \Gamma \to \Delta$ has also a very simple expression. Let $Y = vars(\Gamma \to \Delta)$. Then, $(T_{\mathcal{E}^{\sqsubset}}, [\overline{\alpha}]) \models \Gamma \to \Delta$ exactly means that for each ground constructor substitution $\beta : Y \to T_\Omega$ we have $T_{\mathcal{E}^{\sqsubset}}, [\alpha] \uplus [\beta] \models (\Gamma \to \Delta)^\circ$, where $(\Gamma \to \Delta)^\circ$ is obtained from $\Gamma \to \Delta$ by replacing each constant $\overline{x} \in \overline{X}$ appearing in it by its corresponding variable $x \in X$.

3.2 Inductive Inference System

The inductive inference system that we present below transforms *inductive goals* of the form: $[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda$ —where $[\overline{X},\mathcal{E},H]$ is an inductive theory and $\Gamma \to \Lambda$ is a $\Sigma^{\sqsubset}(\overline{X})$-superclause— into sets of goals, with the empty set of goals denoted $\top$, suggesting that the goal from which it was generated has been proved (is a *closed* goal). However, in the special case of goals of the form $[\emptyset,\mathcal{E},\emptyset] \Vdash \Gamma \to \Lambda$, called *initial goals*, we furthermore require that $\Gamma \to \Lambda$ is a

[6] Even when, say, an induction hypothesis in H might originally be a superclause $\Gamma \to \bigwedge_{l \in L} \Delta_l$, for executability reasons we will always decompose it into its corresponding set of clauses $\{\Gamma \to \Delta_l\}_{l \in L}$.

Σ-superclause; and we also allow *existential initial goals* of the form $[\varnothing,\mathcal{E},\varnothing] \Vdash \exists(\Gamma \to \Lambda)$, with $\exists(\Gamma \to \Lambda)$ the existential closure of a Σ-superclause. A *proof tree* is a tree of goals where at the root we have the original goal that we want to prove and the children of each node in the tree have been obtained by applying an inference rule in the usual bottom-up proof search fashion. Goals in the leaves are called the *pending goals*. A proof tree is *closed* if it has no pending goals, i.e., all the leaves are marked $\top$. Soundness of the inference system means that if the goal $[\overline{X},\mathcal{E},H] \Vdash \phi$ is the root of a closed proof tree, then ϕ is valid in the inductive theory $[\overline{X},\mathcal{E},H]$, i.e., it is satisfied by all the models $(T_{\mathcal{E}^{\square}},[\overline{\alpha}])$ of $[\overline{X},\mathcal{E},H]$ in the sense explained above.

The inductive inference system presented below consists of two sets of inference rules: (1) *simplification rules*, which are easily amenable to automation, and (2) *standard rules*, which are typically applied under user guidance, although they could also be automated by tactics.

Inductive Simplification Rules

Equality Predicate Simplification (**EPS**).

$$\frac{[\overline{X},\mathcal{E},H] \Vdash (\Gamma \to \Lambda)!_{\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}_{e_U}}}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda}$$

where $B = B_0 \cup U$ is the decomposition of B into unit axioms U and the remaining associative and/or commutative axioms B_0, and $\vec{\mathcal{E}}_{\overline{X}_U} = (\Sigma(\overline{X}),B_0,\vec{E}_U \cup \vec{U})$ is the semantically equivalent rewrite theory obtained from $\vec{\mathcal{E}}_{\overline{X}} = (\Sigma(\overline{X}),B,\vec{E})$ using the $\vec{\mathcal{E}} \mapsto \vec{\mathcal{E}}_U$ transformation specified in [7]. That is, we add the axioms U as rules $\vec{U}$ and transform the equations $\vec{E}$ into $\vec{E}_U$ by mapping each $(l \to r) \in \vec{E}$ to the set of rules $\{l_i \to r\alpha_i \mid 1 \leqslant i \leqslant n\}$, where $\{(l_i,\alpha_i)\}_{1 \leqslant i \leqslant n}$ is the finite set of $\vec{U},B_0$-variants of l. For example, if $\_,\_$ is an ACU multiset union operator of sort *MSet* with identity $\varnothing$ and with subsort *Elt* of elements, a membership rewrite rule $x \in x,S \to true$ modulo ACU with x of sort *Elt* and S of sort *MSet* is mapped to the set of rules $\{x \in x \to true, x \in x,S \to true\}$ modulo AC. Since these theories are semantically equivalent, we have $T_{\mathcal{E}_{\overline{X}}} \cong T_{\mathcal{E}_{\overline{X}_U}}$. $\vec{\mathcal{E}}^{=}_{\overline{X}_U}$ is the theory obtained from $\vec{\mathcal{E}}_{\overline{X}_U}$ by adding to it the equationally defined equality predicates defined in [12].

We also assume that in $\mathcal{E}_\Omega = (\Omega,B_\Omega)$, the axioms B_Ω decompose as $B_\Omega = Q_\Omega \uplus U_\Omega$ with U_Ω the unit axioms and Q_Ω the associative and/or commutative axioms such that $T_{\mathcal{E}_\Omega} \cong T_{\Omega/Q_\Omega}$. This can be arranged with relative ease in many cases by subsort overloading, so that the rules in $\vec{U}_\Omega$ only apply to subsort-overloaded operators that are *not* constructors.

For example, consider sorts $Elt < NeList < List$ and Ω with operators *nil* of sort *List* and $\_;\_ : NeList\ NeList \to NeList$ and B_Ω associativity of $\_;\_$ with identity *nil*, but where $\_;\_ : List\ List \to List$, declared with the same axioms, is in $\Sigma \backslash \Omega$. Then Q_Ω is just the associativity axiom for $(\_;\_)$. Finally, note that the executable hypotheses H_e in the theory $[\overline{X},\mathcal{E},H]$, transformed as rules $\vec{H}_{e_U}$ exactly as for $\vec{E}_U$, are also added to the theory $\vec{\mathcal{E}}^{=}_{\overline{X}_U}$ as extra rewrite rules.

In summary, this inference rule simplifies a superclause $\Gamma \to \Lambda$ with: (i) the rules in $\vec{E}_U$, (ii) the equality predicate rewrite rules, and (iii) the executable hypotheses rewrite rules $\vec{H}_{e_U}$.[7]

[7] Recall that Γ is a conjunction and Λ a conjunction of disjunctions. Therefore, the equality predicate rewrite rules together with $\vec{H}_{e_U}$ may have powerful "cascade effects." For example, if either $\Gamma!_{\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}_{e_U}} = \bot$ or $\Gamma!_{\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}_{e_U}} = \top$, then $(\Gamma \to \Lambda)!_{\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}_{e_U}}$ is a tautology and the goal is proved.

Constructor Variant Unification Left (**CVUL**).

$$\frac{\left\{[\overline{X},\mathcal{E},H] \Vdash (\Gamma' \to \Lambda)\gamma\right\}_{\gamma \in Unif^{\Omega}_{\mathcal{E}_1}(\Gamma)}}{[\overline{X},\mathcal{E},H] \Vdash \Gamma,\Gamma' \to \Lambda}$$

where Γ is a conjunction of $\mathcal{E}_1$-equalities (therefore not containing any constants in $\overline{X}$), Γ' contains no extra such $\mathcal{E}_1$-equalities, and $Unif^{\Omega}_{\mathcal{E}_1}(\Gamma)$ denotes the set of constructor $\mathcal{E}_1$-unifiers of Γ [20, 28].

Constructor Variant Unification Failure Left (**CVUFL**).

$$\frac{\top}{[\overline{X},\mathcal{E},H] \Vdash \Gamma,\Gamma' \to \Lambda}$$

where Γ is a conjunction of $\mathcal{E}_{1\overline{X}}$-equalities, Γ' contains no extra such $\mathcal{E}_{1\overline{X}}$-equalities, Γ° is the conjunction of $\mathcal{E}_1$-equalities obtained by replacing the constants $\overline{x} \in \overline{X}$ by corresponding variables $x \in X$, and $Unif^{\Omega}_{\mathcal{E}_1}(\Gamma^{\circ}) = \varnothing$.

Constructor Variant Unification Failure Right (**CVUFR**).

$$\frac{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda \wedge \Delta}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda \wedge (u = v,\Delta)}$$

where $u = v$ is a $\mathcal{E}_{1\overline{X}}$-equality and $Unif^{\Omega}_{\mathcal{E}_1}((u = v)^{\circ}) = \varnothing$.

Substitution Left (**SUBL**).

$$\frac{[\overline{X},\mathcal{E},H] \Vdash (\Gamma \to \Lambda)\{x \mapsto u\}}{[\overline{X},\mathcal{E},H] \Vdash x = u,\Gamma \to \Lambda}$$

where: (i) x is a variable of sort s, $ls(u) \leqslant s$, and $x \notin vars(u)$; and (ii) u is not a Σ_1-term, or if so, then Γ contains no other Σ_1-equations. Note that $\_ = \_$ is assumed commutative, so both $x = u$ and $u = x$ are covered.

Substitution Right (**SUBR**).

$$\frac{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to x = u \qquad [\overline{X},\mathcal{E},H] \Vdash (\Gamma \to \Lambda)\{x \mapsto u\}}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda \wedge x = u}$$

where x is a variable of sort s, $ls(u) \leqslant s$, and $x \notin vars(u)$.

Clause Subsumption (**CS**).

$$\frac{[\overline{X},\mathcal{E},H \cup \{\Gamma \to \Delta\}] \Vdash \Gamma\theta,\Gamma' \to \Lambda}{[\overline{X},\mathcal{E},H \cup \{\Gamma \to \Delta\}] \Vdash \Gamma\theta,\Gamma' \to \Lambda \wedge (\Delta\theta,\Delta')}$$

Note that in the application of the **CS** inference rule to a *concrete* superclause we implicitly use ACU-matching in the following two ways: (i) on the *left* we identify the meta-notation $\_,\_$ with $\wedge$ as a single ACU symbol with identity $\top$; and (ii) on the *right* we identify the meta-notation $\_,\_$ with $\vee$ as a single ACU symbol with identity $\bot$. Therefore, desugaring $\_,\_$ on the left as $\wedge$ and on the right as $\vee$, what this inference rule is really doing is matching the given concrete superclause goal against the *pattern* $\Gamma \wedge \Gamma' \rightarrow (\Delta \vee \Delta') \wedge \Lambda$, where $\Gamma \rightarrow \Delta$ is a *concrete clause* in the given theory's hypotheses, whereas Γ', Δ', and Λ are *meta-variables*. But if this ACU match succeeds, the *concrete superclause* must have the form: $\Gamma\theta \wedge \Gamma'_0 \rightarrow \Lambda_0 \wedge (\Delta\theta \vee \Delta'_0)$, where now Γ'_0, Δ'_0, and Λ_0 are all *concrete*. In particular, since $\_ \wedge \_$ is ACU with identity $\top$, we could have $\Lambda_0 \equiv \top$, so that our concrete superclause was actually a *clause*. But then, the bottom-up application of the **CS** rule to this concrete clause will result in the tautology goal $\Gamma\theta \wedge \Gamma'_0 \rightarrow \top$, so that in this case the **CS** rule *proves* the given clause goal.

Equation Rewriting (Left and Right) (**ERL** and **ERR**).

$$\textbf{(ERL)}\ \frac{[\overline{X},\mathcal{E},H] \Vdash (u' = v')\theta, \Gamma \rightarrow \Lambda \qquad [\varnothing,\mathcal{E},\varnothing] \Vdash u = v \Leftrightarrow u' = v'}{[\overline{X},\mathcal{E},H] \Vdash (u = v)\theta, \Gamma \rightarrow \Lambda}$$

$$\textbf{(ERR)}\ \frac{[\overline{X},\mathcal{E},H] \Vdash \Gamma \rightarrow \Lambda \wedge ((u' = v')\theta, \Delta) \qquad [\varnothing,\mathcal{E},\varnothing] \Vdash u = v \Leftrightarrow u' = v'}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \rightarrow \Lambda \wedge ((u = v)\theta, \Delta)}$$

where $vars(u' = v') \subseteq vars(u = v)$. When using the equational equivalence as a rewrite rule $(u = v) \rightarrow (u' = v')$ two additional requirements are imposed:

1. The equivalence $(u = v) \Leftrightarrow (u' = v')$ should be verified ahead of time as a separate proof obligation, so that it can be used automatically to simplify many goals without requiring reproving $(u = v) \Leftrightarrow (u' = v')$ each time.
2. The rewrite theory $\vec{\mathcal{E}}^{=}_{U}$ axiomatizing the equality predicates should remain terminating when the rule $(u = v) \rightarrow (u' = v')$ is added to it.

In fact, what requirements (1)–(2) provide is a general method to fully *automate* rules **(ERL)** and **(ERR)** so that they are *subsumed*[8] by a more powerful version of the **(EPS)** simplification rule in which the equality predicate theory $\vec{\mathcal{E}}^{=}_{U}$ has been extended with rules of the form $(u = v) \rightarrow (u' = v')$ proved as lemmas.[9] For an example of a useful rewrite rule of this kind, namely, the clearly terminating rewrite rule $x * z' = y * z' \rightarrow x = y$ for the equality predicate of natural numbers, where x,y range over naturals and z' over non-zero naturals, as well as its proof allowing it to be added to the equality predicate theory $\vec{\mathcal{N}}^{=}_{U}$ of the natural numbers, see Sections 3.3 and 4.3.

Inductive Congruence Closure (**ICC**).

[8] The net effect is not only that **(EPS)** both subsumes **(ERL)** and **(ERR)** and becomes more powerful: by adding such extra rules to $\vec{\mathcal{E}}^{=}_{U}$, the **ICC** simplification rule discussed next, which also performs simplification with equality predicates, also becomes more powerful.

[9] More generally, the equality predicate theory $\vec{\mathcal{E}}^{=}_{U}$ can be extended by adding to it *conditional rewrite rules* that orient inductive theorems of $\mathcal{E}$ or $\mathcal{E}^{=}_{U}$, are executable, and keep $\vec{\mathcal{E}}^{=}_{U}$ operationally terminating. For example, if c and c' are *different* constructors whose sorts belong to the same connected component having a top sort, say, s, then the conditional rewrite rule $x = c(x_1,...,x_n) \wedge x = c'(y_1,...,y_m) \rightarrow \bot$, where x has sort s orients an inductively valid lemma, clearly terminates, and can thus be added to $\vec{\mathcal{E}}^{=}_{U}$. In particular, if p is a Boolean-valued predicate and $u_i =_{B_0} v_i$, $p(u_1,...,u_n) = true \wedge p(v_1,...,v_n) = false$ rewrites to $\bot$.

$$\frac{[\overline{X},\mathcal{E},H] \Vdash \top}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda} \text{ if } \overline{\Gamma'} = \bot \text{ or } \top \in \overline{\Lambda}!_{\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}_{e_U} \cup \vec{\overline{\Gamma'}}}$$

$$\frac{[\overline{X},\mathcal{E},H] \Vdash \Gamma' \to \Lambda'}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda} \text{ if } \overline{\Gamma'} \neq \bot \text{ and } \top \notin \overline{\Lambda}!_{\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}_{e_U} \cup \vec{\overline{\Gamma'}}} \text{ and } \overline{\Lambda}' \in \overline{\Lambda}!_{\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}_{e_U} \cup \vec{\overline{\Gamma'}}}$$

where the notation in the side conditions of the two versions of the **ICC** rule is explained below, and, as explained in Section 2.5, the goal $[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda$ is semantically equivalent to the goal $[\overline{X} \uplus \overline{Y},\mathcal{E},H \cup \{\overline{\Gamma}\}] \Vdash \overline{\Lambda}$, where $Y = vars(\Gamma \to \Lambda)$. Assuming that $\mathcal{E}$ has axioms $B = B_0 \uplus U$ and recalling the definition of the congruence closure $cc^{\succ}_{B_0}(\overline{\Gamma})$ in Section 2.4, let us then define

$$\overline{\Gamma'} = (\bigwedge_{(l \to r) \in cc^{\succ}_{B_0}(\overline{\Gamma})} l = r)!_{\vec{\mathcal{E}}_{\overline{X}_U} \cup \vec{H}_{e_U}}.$$

Assuming $\overline{\Gamma'} \neq \bot$, and disregarding the difference between sets and conjunctions of equations, let us also define

$$\vec{\overline{\Gamma'}} = \text{orient}^{\succ}(\overline{\Gamma'})$$

where for any ground equation $u = v$, we define $\text{orient}^{\succ}(u = v) = \underline{\text{if}}\ u \succ v\ \underline{\text{then}}\ u \to v\ \underline{\text{else}}\ v \to u\ \underline{\text{fi}}$, with $\succ$ an RPO order modulo B_0 total on B_0-equivalence classes of ground terms. Intuitively, what we want is to reduce if possible $\overline{\Lambda}$ to $\top$ using the combined power of $\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}_{e_U} \cup \vec{\overline{\Gamma'}}$. However, since these combined rules, although operationally terminating by construction, need not be confluent, we will increase our chances of reaching the desired $\top$ result if we explore the entire *set* of *all canonical forms* of $\overline{\Lambda}$ under those rules. By abuse of notation, we denote this set — which can be computed in Maude by means of the `search =>!` command— as $\overline{\Lambda}!_{\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}_{e_U} \cup \vec{\overline{\Gamma'}}}$.

The purpose of the rule transformation $cc^{\succ}_{B_0}(\overline{\Gamma}) \mapsto \vec{\overline{\Gamma'}}$ is to further increase the chances of success in simplifying $\overline{\Lambda}$, as compared to just using $cc^{\succ}_{B_0}(\overline{\Gamma})$. In the context of the other rules, $\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}_{e_U}$, these chances can be further increased in two ways: (i) the lefthand side l of a rule $(l \to r) \in cc^{\succ}_{B_0}(\overline{\Gamma})$ may be reducible by $\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}_{e_U}$, thus preventing its application, whereas this can never happen for rules in $\vec{\overline{\Gamma'}}$; (ii) suppose, as an example, a rule in $cc^{\succ}_{B_0}(\overline{\Gamma})$ of the form $s(u) \to s(v)$, where s is the successor constructor for natural numbers, and assume for simplicity that u,v are irreducible by $\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}_{e_U}$; then, thanks to its reduction by $\vec{\mathcal{E}}^{=}_{\overline{X}_U}$, in $\vec{\overline{\Gamma'}}$ this rule will become the rule $u \to v$, which is much more widely applicable than the original rule $s(u) \to s(v)$.

In summary, the first version of the **ICC** rule can fully prove a goal $\Gamma \to \Lambda$ if either: (i) $\overline{\Gamma}$ can be proved unsatisfiable by simplifying to $\bot$ the conjunction associated to its congruence closure using $\vec{\mathcal{E}}_{\overline{X}_U} \cup \vec{H}_{e_U}$, or (ii) we can simplify $\overline{\Lambda}$ to $\top$ using the combined power of $\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}_{e_U} \cup \vec{\overline{\Gamma'}}$. However, if a full proof by either (i) or (ii) cannot be obtained, we can still use the second version of the **ICC** rule to derive a hopefully simpler goal $\Gamma' \to \Lambda'$ (where the choice of $\overline{\Lambda}' \in \overline{\Lambda}!_{\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}_{e_U} \cup \vec{\overline{\Gamma'}}}$ is arbitrary but could be optimized according to some criteria) as partial progress in the proof effort short of actually proving the goal.

Variant Satisfiability (**VARSAT**).

$$\frac{\top}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda}$$

if $\Gamma \to \Lambda$ is an $\mathcal{E}_1$-formula and $\neg(\Gamma^\circ \to \Lambda^\circ)$ is unsatisfiable in $T_{\mathcal{E}_1}$, where $\Gamma^\circ \to \Lambda^\circ$ is obtained from $\Gamma \to \Lambda$ by replacing constants in $\overline{X}$ by corresponding variables in X.

Standard Inductive Rules

Cover Set Induction (**CSI**).

$$\frac{\{[\overline{X} \uplus \overline{Y}_i,\mathcal{E},H \uplus \{(\Gamma \to \Delta_j)\{z \to \overline{y}\}\}_{\overline{y} \in \overline{Y}_{i \leqslant s}}^{j \in J}] \Vdash (\Gamma \to \bigwedge_{j \in J} \Delta_j)\{z \mapsto \overline{u}_i\}\}_{1 \leqslant i \leqslant n}}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \bigwedge_{j \in J} \Delta_j}$$

where $z \in vars(\Gamma \to \bigwedge_{j \in J} \Delta_j)$ has sort s, $\{u_1 \cdots u_n\}$ is a *cover set*[10] for s with fresh variables $Y_i = vars(u_i)$ for $1 \leqslant i \leqslant n$, $Y_{i \leqslant s} = \{y \in Y_i \mid sort(y) \leqslant s\}$, $\overline{Y_i}$ are fresh constants of the same sorts for each Y_i, and $\overline{u}_i$ is the instantiation of u_i by such fresh constants.

Existential ($\exists$).

$$\frac{[\varnothing,\mathcal{E},\varnothing] \Vdash (\Gamma \to \Lambda)\theta}{[\varnothing,\mathcal{E},\varnothing] \Vdash (\exists X)(\Gamma \to \Lambda)}$$

where $vars(\Gamma \to \Lambda) = X$ and θ is a substitution. Note that the ($\exists$) rule only applies when the inductive theory is $[\varnothing,\mathcal{E},\varnothing]$, that is, at the beginning of the inductive reasoning process, and that θ must be provided by the user as a witness.

Lemma Enrichment (**LE**).

$$\frac{[\varnothing,\mathcal{E},\varnothing] \Vdash \Gamma' \to \bigwedge_{j \in J} \Delta'_j \qquad [\overline{X},\mathcal{E},H \uplus \{\Gamma' \to \Delta'_j\}_{j \in J}] \Vdash \Gamma \to \Lambda}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda}$$

Split (**SP**).

$$\frac{\{[\overline{X},\mathcal{E},H] \Vdash (u_i = v_i)\theta, \Gamma \to \Lambda\}_{1 \leqslant i \leqslant n} \qquad [\varnothing,\mathcal{E},\varnothing] \Vdash u_1 = v_1 \vee \cdots \vee u_n = v_n}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda}$$

where $vars((u_1 = v_1 \vee \cdots \vee u_n = v_n)\theta) \subseteq vars(\Gamma \to \Lambda)$.

Case (**CAS**).

$$\frac{\left\{[\overline{X},\mathcal{E},H] \Vdash (\Gamma \to \Lambda)\{z \mapsto u_i\}\right\}_{1 \leqslant i \leqslant n}}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda}$$

[10] A *cover set* for s is a finte set of Ω-terms such that $ls(u_i) \leqslant s$, $1 \leqslant i \leqslant n$, and generating all constructor ground terms of sort s modulo B_Ω, i.e., $T_{\Omega/B_\Omega,s} = \bigcup_{1 \leqslant i \leqslant n} \{[u_i\rho] \mid \rho \in [Y_i \to T_\Omega]\}$.

where $z \in vars(\Gamma \rightarrow \Lambda)$ has sort s and $\{u_1,\cdots,u_n\}$ is a cover set for sort s with the u_i for $1 \leqslant i \leqslant n$ having fresh variables.

Variable Abstraction (**VA**).

$$\frac{[\overline{X},\mathcal{E},H] \Vdash u = w, x_1 = v_1,\cdots,x_n = v_n,\Gamma \rightarrow \Lambda}{[\overline{X},\mathcal{E},H] \Vdash u = v,\Gamma \rightarrow \Lambda}$$

where u and w are Σ_1-terms but v is not and $x_1,\ldots,x_n$ are fresh variables whose sorts are respectively the least sorts of the $v_1,\ldots,v_n$, which are subterms of v such that their top symbols are not in Σ_1, and $v =_B w\{x_1 \mapsto v_1,\ldots,x_n \mapsto v_n\}$.

The main property about the above inference system is the following Soudnness Theorem:

Theorem 1 (Soundness Theorem). *If a closed proof tree can be built from a goal of the form* $[\overline{X},\mathcal{E},H] \Vdash \Gamma \rightarrow \Lambda$, *then* $[\overline{X},\mathcal{E},H] \models \Gamma \rightarrow \Lambda$.

Although the Soundness Theorem is stated in full generality, in practice, of course, its main application will be to initial goals of the form $[\emptyset,\mathcal{E},\emptyset] \Vdash \Gamma \rightarrow \Lambda$.

3.3 Inductive Inference System Example

Suppose we wish to prove the cancellation law for natural number multiplication

$$x * z' = y * z' \Rightarrow x = y$$

where z' is a non-zero natural number while x and y are natural numbers. We specify natural number addition and multiplication as associative-commutative operators in theory $\mathcal{N}$ having a subsort relation $NzNat < Nat$ of non-zero numbers as subset of all naturals. Of course, since the proof of the reverse implication $x = y \Rightarrow x * z' = y * z'$ follows trivially by simplification with the **(ICC)** rule, what we are really proving is the equivalence $x * z' = y * z' \Leftrightarrow x = y$. Therefore, as discussed when introducing the **(ERL)** and **(ERR)** rules, once the above cancellation rule has been proved, the rewrite rule $x * z' = y * z' \rightarrow x = y$ can be added to the equality predicate theory $\vec{\mathcal{N}}^{=}$ ($\mathcal{N}$ has no U axioms, i.e., $\vec{\mathcal{N}}^{=}_{U} = \vec{\mathcal{N}}^{=}$) to obtain a more powerful version of the **(EPS)** simplification rule. We begin with the goal:

$$G : [\emptyset,\mathcal{N},\emptyset] \Vdash x * z' = y * z' \rightarrow x = y$$

After applying the rule **CSI** to the variable x with the cover set $\{0,1+x_1\}$ and simplifying by **EPS** we obtain:

$$G_1 : [\emptyset,\mathcal{N},\emptyset] \Vdash 0 = y * z' \rightarrow 0 = y$$
$$G_2 : [\{\overline{x}_1\},\mathcal{N},\overline{x}_1 * z' = y * z' \rightarrow \overline{x}_1 = y] \Vdash z' + (\overline{x}_1 * z') = y * z' \rightarrow \overline{x}_1 + 1 = y$$

We first prove G_1 by: (a) applying the **CAS** rule to variable y with the cover set $\{0,y'\}$, where y' has the non-zero natural sort $NzNat$; and (b) applying the **EPS** rule to obtain:

$$G_{1.1} : [\emptyset,\mathcal{N},\emptyset] \Vdash 0 = 0 \rightarrow 0 = 0$$
$$G_{1.2} : [\emptyset,\mathcal{N},\emptyset] \Vdash 0 = y' * z' \rightarrow 0 = y'$$

To solve $G_{1.1}$, apply **EPS** to obtain $\top$. To solve $G_{1.2}$, apply **VA** to the term $y' * z'$ which has least sort *NzNat* to obtain:

$$G_{1.2.1} : [\emptyset, \mathcal{N}, \emptyset] \Vdash 0 = z'', z'' = y' * z' \rightarrow 0 = y'$$

where z'' also has sort *NzNat*. Finally apply **CVUL** to obtain $\top$, since the equation $0 = z''$ has no unifiers. This finishes the proof of G_1. We now prove G_2 by: (a) applying the **CAS** rule to variable y with cover set $\{0, y_1 + 1\}$; and (b) applying the **EPS** rule to obtain:

$$G_{2.1} : [\{\overline{x}_1\}, \mathcal{N}, \overline{x}_1 * z' = y * z' \rightarrow \overline{x}_1 = y] \Vdash z' + (\overline{x}_1 * z') = 0 \rightarrow \overline{x}_1 + 1 = 0$$

$$G_{2.2} : [\{\overline{x}_1\}, \mathcal{N}, \overline{x}_1 * z' = y * z' \rightarrow \overline{x}_1 = y] \Vdash \begin{array}{l} z' + (\overline{x}_1 * z') = (y_1 * z') + z' \\ \quad \rightarrow \overline{x}_1 + 1 = y_1 + 1 \end{array}$$

To solve $G_{2.1}$, apply **VA** to the term $z' + (\overline{x}_1 * z')$ which has least sort *NzNat* to obtain:

$$G_{2.1.1} : [\{\overline{x}_1\}, \mathcal{N}, \overline{x}_1 * z' = y * z' \rightarrow \overline{x}_1 = y] \Vdash \begin{array}{l} z' + (\overline{x}_1 * z') = z'', z'' = 0 \\ \quad \rightarrow \overline{x}_1 + 1 = 0 \end{array}$$

where z'' also has sort *NzNat*. As in $G_{1.2.1}$, apply **CVUL** to obtain $\top$. Finally, to solve $G_{2.2}$, we apply **ERL** and **ERR** with the equivalence $z_1 + z_2 = z_1 + z_3 \Leftrightarrow z_2 = z_3$ (which can be proved by variant satisfiability) to obtain:

$$G_{2.2.1} : [\{\overline{x}_1\}, \mathcal{N}, \overline{x}_1 * z' = y * z' \rightarrow \overline{x}_1 = y] \Vdash \overline{x}_1 * z' = y_1 * z' \rightarrow \overline{x}_1 = y_1$$

But note that a proof of $G_{2.2.1}$ immediately follows by **CS**. In summary, we completed the proof after 14 applications of our inference rules.

4 Inference System Mechanization and Examples

Given the extensive nature of the inductive inference system we introduced (with 17 rules), a natural question to ask is: do any effective strategies exist? In fact, a well-chosen answer to this question can be the key differentiator between a tedious proof assistant requiring the user to apply one rule at a time, and an efficient proof assistant automating large parts of a proof. Automating away the rote and tedious parts of mechanical proof allows the user to reason at a higher level of abstraction. As mentioned before, there is a core subset of proof rules, the 11 so-called simplification rules, that can be automated, leaving only the 6 standard rules to be applied by the user. In this section we present: (a) an inductive simplification strategy *ISS* that applies 9 of the 11 simplification rules in combination until a fixpoint is reached (**VARSAT** and **CVUFL** are not included; they will be added in an ISS^+ extension); (b) an overview of how our inductive simplification strategy has been implemented in Maude; (c) a simple example illustrating how the inductive simplification strategy operates in practice; and (d) our encouraging experience using a simplified version of this inductive simplification strategy to automatically prove all VCs generated by the proof of security of the IBOS browser.

4.1 Inductive Simplification Strategy

The strategy we present takes as input a set of goals Φ and outputs a set of goals Φ'. Let $\mathcal{G}$ denote the set of all goals over some ground convergent $\vec{\mathcal{E}}$. Then each individual inductive simplification inference rule R is a function $R \in [\mathcal{G} \to \mathcal{P}(\mathcal{G}) \uplus \{\bot\}]$. That is, each R is a function that takes a single goal as input and either fails to evaluate because the side-condition does not provably hold or else outputs: (i) a non-empty set of goals; (ii) an empty set of goals (equivalently $\top$), that is, the rule closes a branch of the proof tree; or (iii) a counterexample (equivalently $\bot$), that is, the proof immediately terminates in failure. For simplicity, we treat rule side-condition failure (or the rule not matching) as identity (i.e., the original goal is returned unchanged).

Then, for each inference rule R, we can consider the set-lifting of that rule R_{set}, i.e., $R_{\text{set}} \in [\mathcal{P}(\mathcal{G} \uplus \{\bot\}) \to \mathcal{P}(\mathcal{G} \uplus \{\bot\})]$ which is defined by $R_{\text{set}}(\Phi) = \bigcup_{\phi \in \Phi} R(\phi)$. For simplicity of notation, let semicolon $(;)$ denote in-order function composition. For an order-continuous function $f \in [A \to A]$, let $f!$ denote the recursive fixpoint construction of f, i.e.,

$$f!(a) = \begin{cases} a & \text{if } a = f(a) \\ f!(f(a)) & \text{otherwise} \end{cases}$$

Then we define our inductive simplification strategy (*ISS*) by means of a variant simplification strategy (*VSS*). We have:

$$VSS = (\mathbf{EPS}_{\text{set}} \,;\, \mathbf{ERL}_{\text{set}}! \,;\, \mathbf{ERR}_{\text{set}}! \,;\, \mathbf{SUBL}_{\text{set}}! \,;\, \mathbf{SUBR}_{\text{set}}! \,;\, \mathbf{CVUL}_{\text{set}} \,;\, \mathbf{CVUR}_{\text{set}})!$$
$$ISS = (\mathbf{CS}_{\text{set}} \,;\, VSS \,;\, \mathbf{ICC}_{\text{set}} \,;\, \mathbf{CS}_{\text{set}})!$$

The inner strategy simplifies a set of goals by equality predicate simplification, substitution elimination, and variant unification to the limit. The outer strategy takes a set of variant-simplified goals and applies the inductive congruence closure rule. The reason for this stratification is simple: the inductive congruence closure rule (**ICC**) is computationally expensive because the congruence closure algorithm may require many iterations before convergence. By simplifying the input goals as much as possible before applying congruence closure, we increase the speed of convergence.

4.2 Inductive Simplification Strategy Mechanization

The strategy presented above has been mechanized in Maude. Recall that our inference system assumes a ground convergent theory $\vec{\mathcal{E}}$ with with $B = B_0 \uplus U$ and such that $\vec{\mathcal{E}}_U$ is operationally terminating via a recursive path ordering (RPO) $(\succ)$ modulo B_0 that is total on ground terms and that has constructor and finite variant subtheories respectively $(\Omega, B_\Omega, \varnothing) \subseteq (\Sigma_1, B_1, \vec{E}_1) \subseteq (\Sigma, B, \vec{E})$. Since rewriting logic is reflective, for any rewriting logic derivation $\mathcal{R} \vdash t \to t'$, there is a universal rewrite theory that can internalize the theory $\mathcal{R}$ as well as terms t, t' and the derivation $\mathcal{U} \vdash \overline{\mathcal{R} \vdash t \to t'}$. In Maude, key functionality of this universal theory is defined by the prelude module META-LEVEL [3].

Thus, our inductive simplification strategy is defined by a Maude rewrite theory $ISS_{\mathcal{R}}$ that protects META-LEVEL and takes as input: (i) an inductive theory $[\overline{X}, \mathcal{E}, H]$, specified as a functional theory in Maude (or a functional module if $\overline{X} = \varnothing$ and $H = \varnothing$), where each symbol $f \in \Sigma$ is annotated with a natural number that denotes its order in the RPO (the numbers for the fresh constants in $\overline{X}$ are added automatically), and where the functional submodule $\mathcal{E}$ has specified subtheories $(\Omega, B_\Omega, \varnothing)$ and $(\Sigma_1, B_1, \vec{E}_1)$; (ii) a quantifier-free $\Sigma(\overline{X})$-superclause ϕ to be proved.

The equality predicate simplification [12], *B*-recursive path ordering [23, 11], variant unification [20, 28], and order-sorted congruence closure modulo *B* [19] algorithms have all been implemented in Maude. This work combines all of those existing algorithms together into a powerful inductive simplification strategy.

4.3 Inductive Simplification Strategy Example and IBOS VC Proofs

To conclude this section, we first recall the multiplication cancellation law proof from Section 3.3. To recap, we wanted to show:

$$x*z' = y*z' \Rightarrow x = y$$

in the theory $\mathcal{N}$ of naturals mentioned in Section 3.3, where z' is a non-zero natural number and x and y are natural numbers. For our semi-automated proof, we first prove the equivalence $z_1+z_2 = z_1+z_3 \Leftrightarrow z_2+z_3$ by variant satisfiability and make this equivalence available to the tool as a simplification lemma. The proof proceeds as before *except* that we apply our strategy *ISS* at the beginning of the proof *and* after each inference step. Thus, begin with goal:

$$G : [\emptyset, \mathcal{N}, \emptyset] \Vdash x*z' = y*z' \rightarrow x = y$$

Then apply **CSI** on variable x with cover set $\{0, x_1+1\}$ to obtain:

$$G_1 : [\emptyset, \mathcal{N}, \emptyset] \Vdash 0 = y*z' \rightarrow 0 = y$$
$$G_2 : [\{\overline{x}_1\}, \mathcal{N}, \overline{x}_1*z' = y*z' \rightarrow \overline{x}_1 = y] \Vdash z' + (\overline{x}_1*z') = y*z' \rightarrow \overline{x}_1 + 1 = y$$

To prove G_1, we apply the **CAS** rule to variable y with the cover set $\{0, y'\}$ where y' has sort *NzNat*. Since the $\{y \mapsto 0\}$ case is blown away by *ISS* automatically, we obtain the following goal:

$$G_{1.1} : [\emptyset, \mathcal{N}, \emptyset] \Vdash 0 = y'*z' \rightarrow 0 = y'$$

To solve $G_{1.1}$, apply **VA** to the term $y'*z'$ which has least sort *NzNat*; the goal is then immediately closed by *ISS*. This finishes the proof of G_1. We now prove G_2 by applying the **CAS** rule to variable y with cover set $\{0, y_1+1\}$. Since the $y \mapsto y_1+1$ case is immediately closed by *ISS*, we obtain the following goal:

$$G_{2.1} : [\{\overline{x}_1\}, \mathcal{N}, \overline{x}_1*z' = y*z' \rightarrow \overline{x}_1 = y] \Vdash z' + (\overline{x}_1*z') = 0 \rightarrow \overline{x}_1 + 1 = 0$$

To solve $G_{2.1}$, apply **VA** to the term $z' + (\overline{x}_1*z')$ which has least sort *NzNat*; the goal is then immediately closed by *ISS*. In summary, using the *ISS* strategy, we need only 5 inference rule applications versus 14 applications in the original proof. The above proof was carried out using our prototype *ISS* tool, with the not yet implemented standard inference rules applied by hand.

Inductive Simplification Strategy: Automatic Proof of IBOS VCs. The Illinois Browser Operating System (IBOS) [30, 29] is an advanced web browser and operating system built on top of the L4Ka::Pistachio secure microkernel that was developed to push the limits of secure web browser design. In [26, 25] a Maude specification of the IBOS system was developed for which the same-origin policy (SOP) and address bar correctness (ABC) properties were verified using a hand-written state abstraction proof plus bounded model checking. In [22], a first attempt at a fully automated deductive verification of SOP and ABC for IBOS was attempted,

but had to be abandoned because thousands of inductive verification conditions (VCs) were generated. In [27], this deductive verification project was finally completed. Amazingly, using a simplified version of the above inductive simplification strategy as its automatic VC prover backend, the constructor-based reachability logic theorem prover [27] was able to verify all 7 claims corresponding to SOP and ABC for IBOS which in total required approximately 2K lines of Maude code for the system and property specifications and which generated thousands of goals to be solved by the backend VC prover.

5 Related Work and Conclusions

As already mentioned, this work combines automated and explicit-induction theorem proving in a novel way. We of course rely on the extensive literature in this area, which we do not intend to survey. Some of our automatable techniques have been used in some fashion in earlier work, but others have not. For example, congruence closure is used in many provers, but congruence closure modulo is considerably less used, and order-sorted congruence closure modulo is here used for the first time. Contextual rewriting goes back to the Boyer-More prover [2] and has also been used, for example, in RRL [5]; and clause subsumption is used in most automated theorem provers, including inductive ones. Equational simplification is used by everybody, but to the best of our knowledge simplification with equationally-defined equality predicates *modulo* axioms B_0 was only previously used in [21], although in the much easier free case equality predicates have been used to specify "consistency" properties of data types in, e.g., [5]. To the best of our knowledge, neither constructor variant unification nor variant satisfiability have been used in other general-purpose provers, although variant unification is used in various cryptographic protocol verification tools, e.g., [8, 16]. Combining *all* these techniques is new.

In summary, our combination of automated and explicit-induction theorem proving seems to be new and offers the possibility of an inference subsystem that can be automated as a practical oracle for inductive validity of VCs generated by other tools, and that allows a user to focus on applying just 6 inference rules. For the moment, only the *ISS* strategy has been implemented in Maude. Both the extension of *ISS* to ISS^+, and the implementation of an inductive theorem prover supporting the 17 inference rules are unproblematic. They are left for future work.

Acknowledgements. We cordially thank the referees for their very helpful suggestions to improve the paper. Work partially supported by NRL under contract N00173-17-1-G002.

References

1. Bouhoula, A., Rusinowitch, M.: SPIKE: A system for automatic inductive proofs. In: Algebraic Methodology and Software Technology. pp. 576–577 (1995)
2. Boyer, R., Moore, J.: A Computational Logic. Academic Press (1980)
3. 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)
4. Clavel, M., Palomino, M.: The ITP tool's manual, universidad Complutense, Madrid, April 2005, http://maude.sip.ucm.es/itp/
5. Comon, H., Nieuwenhuis, R.: Induction=i-axiomatization+first-order consistency. Inf. Comput. **159**(1-2), 151–186 (2000)
6. Dershowitz, N., Jouannaud, J.P.: Rewrite systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Vol. B, pp. 243–320. North-Holland (1990)

7. Durán, F., Lucas, S., Meseguer, J.: Termination modulo combinations of equational theories. In: Frontiers of Combining Systems, 7th International Symposium, FroCoS 2009, Trento, Italy, September 16-18, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5749, pp. 246–262. Springer (2009)
8. Escobar, S., Meadows, C., Meseguer, J.: Maude-NPA: cryptographic protocol analysis modulo equational properties. In: Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures, LNCS, vol. 5705, pp. 1–50. Springer (2009)
9. Escobar, S., Sasse, R., Meseguer, J.: Folding variant narrowing and optimal variant termination. J. Algebraic and Logic Programming **81**, 898–928 (2012)
10. 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)
11. Gutiérrez, R., Meseguer, J., Skeirik, S.: The Maude termination assistant, in Pre-Proceedings of WRLA 2018
12. Gutiérrez, R., Meseguer, J., Rocha, C.: Order-sorted equality enrichments modulo axioms. Sci. Comput. Program. **99**, 235–261 (2015)
13. Hendrix, J.D.: Decision Procedures for Equationally Based Reasoning. Ph.D. thesis, University of Illinois at Urbana-Champaign (2008), http://hdl.handle.net/2142/10967
14. Kapur, D., Zhang, H.: An overview of rewrite rule laboratory (RRL). In: Proc. RTA-89. Lecture Notes in Computer Science, vol. 355, pp. 559–563. Springer (1989)
15. Lucas, S., Meseguer, J.: Normal forms and normal theories in conditional rewriting. J. Log. Algebr. Meth. Program. **85**(1), 67–97 (2016)
16. Meier, S., Schmidt, B., Cremers, C., Basin, D.A.: The TAMARIN prover for the symbolic analysis of security protocols. In: Proc. CAV 2013. vol. 8044, pp. 696–701. Springer LNCS (2013)
17. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)
18. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Proc. WADT'97. pp. 18–61. Springer LNCS 1376 (1998)
19. Meseguer, J.: Order-sorted rewriting and congruence closure. In: Proc. FOSSACS 2016. Lecture Notes in Computer Science, vol. 9634, pp. 493–509. Springer (2016)
20. Meseguer, J.: Variant-based satisfiability in initial algebras. Sci. Comput. Program. **154**, 3–41 (2018)
21. Rocha, C., Meseguer, J.: Proving safety properties of rewrite theories (2011), in Proc. CALCO 2011, Springer LNCS 6859, 314-328
22. Rocha, C.: Symbolic Reachability Analysis for Rewrite Theories. Ph.D. thesis, University of Illinois at Urbana-Champaign (2012)
23. Rubio, A.: Automated Deduction with Constrained Clauses. Ph.D. thesis, Universitat Politècnica de Catalunya (1994)
24. Rubio, A.: A fully syntactic AC-RPO. Inf. Comput. **178**(2), 515–533 (2002)
25. Sasse, R.: Security models in rewriting logic for cryptographic protocols and browsers. Ph.D. thesis, University of Illinois at Urbana-Champaign (2012), http://hdl.handle.net/2142/34373
26. Sasse, R., King, S.T., Meseguer, J., Tang, S.: IBOS: A correct-by-construction modular browser. In: FACS 2012. Lecture Notes in Computer Science, vol. 7684, pp. 224–241. Springer (2013)
27. Skeirik, S.: Rewriting-based symbolic methods for distributed system verification. Ph.D. thesis, University of Illinois at Urbana-Champaign (2019)
28. Skeirik, S., Meseguer, J.: Metalevel algorithms for variant satisfiability. J. Log. Algebr. Meth. Program. **96**, 81–110 (2018)
29. Tang, S.: Towards Secure Web Browsing. Ph.D. thesis, University of Illinois at Urbana-Champaign (2011), 2011-05-25, http://hdl.handle.net/2142/24307
30. Tang, S., Mai, H., King, S.T.: Trust and protection in the illinois browser operating system. In: 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings. pp. 17–32. USENIX Association (2010)
31. Zhang, H.: Contextual rewriting in automated reasoning. Fundam. Inform. **24**(1/2), 107–123 (1995)

A Remark for the Use of a Path Ordering with an Algebra and a Howard-Style Interpretation of Lambda for Termination Proofs of Typed Rewrite Systems*

Mitsuhiro Okada[1] and Yuta Takahashi[2][0000−0002−5214−7077]

[1] Keio University, 2-15-45 Mita, Minato-ku, Tokyo 108-8345, Japan
mitsu@abelard.flet.keio.ac.jp
[2] IHPST, Université Paris 1 Panthéon-Sorbonne, 13, rue du Four 75006 Paris, France
yuuta.taka84@gmail.com

Abstract. Jouannaud and Rubio's higher-order recursive path ordering HORPO and Blanqui, Jouannaud and Rubio's computability path ordering CPO give natural and good extension of recursive path ordering to treat higher-order rewrite systems. In this paper, we give a case study of yet another direction of such extension of recursive path ordering, avoiding Tait-Girard's computability method plugged in the above mentioned works. This motivation comes from Lévy's question in the RTA open problem 19, which asks for a reasonably straightforward interpretation of simply typed λ-calculus $\lambda_{\rightarrow}$ in a certain well founded ordering. As in the cases of HORPO and CPO, the addition of λ-abstraction and application into path orderings might be considered as one solution, but the following question still remains; Can the termination of $\lambda_{\rightarrow}$ be proved by an interpretation in a first-order well founded ordering in the sense that λ-abstraction/application are not directly built in the ordering? Reconsidering one of Howard's works on proof-theoretic studies, we introduce a path ordering with Howard algebra as a case study towards further studies on Lévy's question.

Keywords: Path orderings · Termination proofs · Term rewrite theory · Simply typed λ-calculus · Primitive recursive functionals of finite type.

1 Introduction

Since Jouannaud and Okada ([10]) showed a solution to a question of the termination problem, namely, the strong normalization problem on typed λ-calculus with higher-order rewrite rules by Tait-Girard's computability method, further improvements have been accumulated [1, 11, 2, 12, 4, 3]. Among them, Jouannaud

* We thank the readers of the earlier version of this paper and the anonymous reviewers for their valuable comments. The first author is supported by JSPS (Japan Society for the Promotion of Science) KAKENHI Grand Numbers 17H02263, 17H02265 and 19KK0006. The second author is a JSPS Overseas Research Fellow.

and Rubio's higher-order recursive path ordering HORPO ([12]) and Blanqui, Jouannaud and Rubio's computability path ordering CPO ([4]) give natural and good extension of recursive path ordering to treat higher-order rewrite systems. On the other hand, Lévy's question in the RTA open problem 19 asks for a termination proof for simply typed λ-calculus $\lambda_{\to}$ using a reasonably straightforward interpretation of $\lambda_{\to}$ in a well founded ordering.[3]

As in the cases of HORPO and CPO, the addition of λ-abstraction and application into path orderings with the use of computability method might be considered as one solution to Lévy's question, but the following question still remains; Can the termination of $\lambda_{\to}$ be proved by an interpretation in a first-order well founded ordering in the sense that λ-abstraction and application are not directly built in the ordering? We are in particular interested in avoiding the computability method and questioning how to embed λ-abstraction/application into a first-order variant of some well-known path ordering (cf. [6, 8, 7]).[4]

The purpose of this paper is to revisit Howard's weak normalization proof for λ-formulation T of primitive recursive functionals of finite type in [9], and introduce a first-order variant of recursive path orderings which enables us to approach Lévy's question in the manner above. First of all, we restrict Howard's proof to $\lambda_{\to}$ since his proof did not show the termination of recursor reduction of T in the sense of strong normalization. Then, Howard's proof turns to a termination proof of $\lambda_{\to}$ using certain two step interpretations. Though this termination proof does not use the computability method, it can be improved for the purpose of finding an interpretation which is simpler and embeds λ-abstraction and application into a variant of some path ordering. Below we simplify Howard's two step interpretations by introducing a path ordering with an algebra in which $\lambda_{\to}$ is embedded with one step (§2). The well foundedness of this path ordering follows from Kruskal's tree theorem and so we do not need the computability method. Next, our path ordering is adapted to termination proofs for some combined systems of $\lambda_{\to}$ with additional higher-order rewrite rules such as the map function, the Ackermann function and primitive recursive functionals of finite type (§3). Note that the rewrite rules of the map function and the Ackermann function which we discuss are not purely recursive rules.

2 A Termination Proof of Simply Typed λ-Calculus by a Path Ordering with an Algebra

In this section, we first introduce a path ordering with an algebra, whose well foundedness follows from Kruskal's tree theorem (§2.1). Next, we give a mapping

[3] `https://www.win.tue.nl/rtaloop/problems/19.html`. Cf. also the TLCA open problem 26 (`http://tlca.di.unito.it/opltlca/opltlcasu33.html#x38-62000`).

[4] If one takes the sequent-style formulation of logic, a reasonably straightforward mapping from proofs to a recursive path ordering (for example, a recursive path ordering of size of the ordinal $\varphi\omega 0$ in Veblen hierarchy) is enough for termination proofs. Lévy's question is open only if one considers not normalization in sequent-calculus (cut-elimination) but normalization in natural deduction or typed λ-calculus.

from simply typed λ-calculus $\lambda_{\to}$ to our path ordering as Howard-style non-unique assignments of this ordering to λ-terms, and then verify that the β-rewrite relation decreases the order in the sense of our path ordering (§2.2).

2.1 Formulation of a Path Ordering with Howard Algebra

First of all, we outline Howard's termination proof of $\lambda_{\to}$ which can be found in [9], and then explain what our contribution is.[5] We follow [6, 7] with respect to the terminology of term rewrite theory such as recursive path orderings RPO, and in particular we use the recursive path ordering with the status multiset only. Let $\mathcal{T}$ be the set of typed λ-terms of $\lambda_{\to}$ (the definition of $\mathcal{T}$ will be given in §2.2). Below "λ-terms" always means typed λ-terms. The set of all finite sequences from a given set $\mathcal{A}$ is denoted by $\mathcal{A}^{<\omega}$. We call a finite sequence of some elements a *vector* as Howard did.

Howard's termination proof of $\lambda_{\to}$ in [9] proceeds as follows. First, he introduced the set of expressions, which we write as $\mathcal{N}$ in this paper (cf. [9, p.448]). The set $\mathcal{N}^{<\omega}$ of all vectors of $\mathcal{N}$-elements forms the base-set of Howard's vector system, and λ-terms of $\mathcal{T}$ were mapped to this vector system. He mapped λ-terms of $\mathcal{T}$ to $\mathcal{N}^{<\omega}$ non-uniquely: Multiple vectors were assigned to each λ-term including some λ-subterms of the form $\lambda X.A$. The reason Howard used non-unique assignments is that he exploited non-unique assignments to cope with the non-monotonicity of the delta operator on the vector system, which was introduced to interpret λ-abstraction (cf. [9, pp.456–457] and [14, Footnote 6]).

The ordering $\succeq$ on $\mathcal{N}^{<\omega}$ in Howard's vector system $(\mathcal{N}^{<\omega}, \succeq)$ was defined via the axiomatic theory $\mathcal{E}$ on $\mathcal{N}$, which consists of several equations and inequalities (cf. [9, p.448, p.457]). The theory $\mathcal{E}$ gave a quasi ordering on $\mathcal{N}$ which is not in path ordering-style but in axiomatic formulation, and $\succeq$ was obtained by extending this ordering to $\mathcal{N}^{<\omega}$ in a natural way. On the other hand, the well foundedness of $\succ$ does not follow from the theory $\mathcal{E}$. To show the well foundedness of $\succ$, Howard used an interpretation of expressions of $\mathcal{N}$ in the set $\mathbb{N}$ of natural numbers.[6] Then, by showing that any reduction in $\lambda_{\to}$ decreases the order in $(\mathcal{N}^{<\omega}, \succeq)$, Howard finished his termination proof of $\lambda_{\to}$. In sum, Howard's termination proof used the following two step mappings, where non-unique assignments above are expressed as the double arrow.

$$\mathcal{T} \Longrightarrow (\mathcal{N}^{<\omega}, \succeq) \longrightarrow \mathbb{N}^{<\omega}$$

Below we eliminate the right-hand mapping from the picture above: The resulting termination proof for $\lambda_{\to}$ has the structure below.

$$\mathcal{T} \Longrightarrow (\mathcal{N}^{<\omega}, \succeq_{\mathrm{po}})$$

[5] As stated in Section 1, it is not $\lambda_{\to}$ but primitive recursive functionals of finite type that Howard actually discussed in [9]. We restrict Howard's proof to $\lambda_{\to}$ because it did not prove the termination of primitive recursive functionals in the sense of strong normalization.

[6] Note that $\mathcal{N}$ is mapped to an ordinal notation system up to ε_0 if we do not restrict Howard's proof to $\lambda_{\to}$.

We define an RPO-style ordering $\succeq_{\rm po}$ on $\mathcal{N}^{<\omega}$ directly, by using a normalizing function which results from oriented equations of Howard's $\mathcal{E}$ above. A difficulty in defining an RPO-style ordering on $\mathcal{N}$ and $\mathcal{N}^{<\omega}$ directly will be revealed in Proposition 1 later. We first introduce the set H of rewrite rules on $\mathcal{N}$, which is obtained by orienting some crucial equations of $\mathcal{E}$. This set gives the convergent rewrite system, and we consider a normalizing function $\mathrm{nf}(t)$ with respect to the H-rewrite relation. The ordering $\succeq_{\rm nf}$ on $\mathcal{N}$, which we call a *path ordering with Howard algebra* below, is defined by means of this normalizing function. The well foundedness of $\succeq_{\rm nf}$ follows from Kruskal's tree theorem, and the ordering $\succeq_{\rm po}$ on $\mathcal{N}^{<\omega}$ is obtained from a straightforward extension of $\succeq_{\rm nf}$ to $\mathcal{N}^{<\omega}$. In sum, we simplify Howard's two step mappings into a single mapping by introducing the well founded RPO-style ordering $(\mathcal{N}^{<\omega}, \succeq_{\rm po})$ into which λ-abstraction/application can be embedded (cf. Lévy's open question mentioned Section 1). We, in §3, will simplify Wilken and Weiermann's two step mappings ([14]) for their termination proof of T in a similar manner.

Let $\mathcal{F}$ be a signature with a quasi ordering $\geq_{\mathcal{F}}$ as its precedence relation, and $\mathcal{X}$ be a set of variables. We denote the set of terms over $\mathcal{F}$ and $\mathcal{X}$ by $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Below the following notations for an arbitrary quasi ordering $\geq$ are used: $t \approx s$ denotes $t \geq s \& s \geq t$, and $t > s$ denotes $t \geq s \& s \not\geq t$. The recursive path ordering on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is denoted by $\succeq_{\rm rpo}$. For any term t of $\mathcal{T}(\mathcal{F}, \mathcal{X})$, the equivalence class of t with respect to an equivalence relation R is denoted by $[t]_R$. Below we fix an arbitrary term-set $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

Definition 1 (A Path Ordering with an Algebra). *Let E be a set of equations on $\mathcal{T}(\mathcal{F}, \mathcal{X})$, and R^E be the equivalence relation on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ induced by E. Moreover, let f be a choice function such that for any term t of $\mathcal{T}(\mathcal{F}, \mathcal{X})$, $f([t]_{R^E}) \in [t]_{R^E}$ holds.*

Then, the path ordering with the algebra E with respect to f is the ordering $(\mathcal{T}(\mathcal{F}, \mathcal{X}), \succeq_E)$, where $\succeq_E$ is defined as follows: For any two terms t and s of $\mathcal{T}(\mathcal{F}, \mathcal{X})$, $t \succeq_E s$ holds if and only if $f([t]_{R^E}) \succeq_{\rm rpo} f([s]_{R^E})$ holds.

We write $E \vdash t = s$ if $t = s$ is derivable from E by means of reflexivity, symmetry, transitivity and the substitution rule. A quasi ordering $\succeq$ on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ and a set E of equations on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ are *compatible on* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ if $E \vdash t = s$ implies $t \approx s$ in the sense of $\succeq$, otherwise they are *incompatible on* $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Theorem 1 below is shown by Kruskal's tree theorem, and Lemma 1 below is the key lemma to the construction of a path ordering with Howard algebra.

Theorem 1 ([5]). *If a quasi ordering $\geq$ on $\mathcal{F}$ is a well quasi ordering, then $\succeq_{\rm rpo}$ on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is a well quasi ordering.*

Lemma 1. *Let $(\mathcal{T}(\mathcal{F}, \mathcal{X}), \succeq_E)$ be the path ordering with an algebra E with respect to a choice function f, and assume that the precedence relation $\geq$ on $\mathcal{F}$ is a well quasi ordering. Then, $(\mathcal{T}(\mathcal{F}, \mathcal{X}), \succeq_E)$ is a well quasi ordering on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $\succeq_E$ and E are compatible on $\mathcal{T}(\mathcal{F}, \mathcal{X})$.*

Proof. First, we show that $\succeq_E$ and E are compatible on $\mathcal{T}(\mathcal{F}, \mathcal{X})$. If $E \vdash t = s$ holds, then we have $[t]_{R^E} = [s]_{R^E}$, where R^E is the equivalence relation induced

by E. Therefore, $f([t]_{R^E}) = f([s]_{R^E})$ holds, and so $t \approx s$ holds in the sense of $\succeq_E$. Next, we show that $(\mathcal{T}(\mathcal{F}, \mathcal{X}), \succeq_E)$ is a well quasi ordering. It is obvious that $(\mathcal{T}(\mathcal{F}, \mathcal{X}), \succeq_E)$ is a quasi ordering. Let $t_0, t_1, \ldots$ be an infinite sequence of terms of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We obtain the infinite sequence $f([t_0]_{R^E}), f([t_1]_{R^E}), \ldots$ and we have $f([t_i]_{R^E}) \succeq_{\mathrm{rpo}} f([t_j]_{R^E})$ for some i, j with $i > j$ by Theorem 1. Therefore, $t_i \succeq_E t_j$ holds by definition.

In the rest of this subsection, we modify Howard's set $\mathcal{N}$ of expressions to a path ordering with Howard algebra. Define $\mathcal{N}$ as the set $\mathcal{T}(\mathcal{X}, \mathcal{F})$ where $\mathcal{X}$ is an infinite set of term variables and $\mathcal{F}$ consists of the constant 1, the binary function symbols $\#$ and h.[7] In Howard's notations, $\#$ and $\mathsf{h}(\cdot, \cdot)$ are written as $+$ and $(\cdot, \cdot)$, respectively. By considering $\#$ to be a variadic function symbol and writing $\#(t_1, \ldots, t_n)$ as $t_1 \# \cdots \# t_n$, we define the set $\mathcal{N}^{\mathrm{fla}}$ of flattened terms.

Definition 2 (The Set $\mathcal{N}^{\mathrm{fla}}$ of Flattened Terms). *The set $\mathcal{N}^{\mathrm{fla}}$ of flattened terms are defined by induction.*

1. *Any term variable x belongs to $\mathcal{N}^{\mathrm{fla}}$, and we say x is a connected term.*
2. *The constant 1 belongs to $\mathcal{N}^{\mathrm{fla}}$, and we say 1 is a connected term.*
3. *If $t_1, t_2 \in \mathcal{N}^{\mathrm{fla}}$ holds, then $\mathsf{h}(t_1, t_2) \in \mathcal{N}^{\mathrm{fla}}$ holds and we say $\mathsf{h}(t_1, t_2)$ is a connected term.*
4. *If $t_1, \ldots, t_n \in \mathcal{N}^{\mathrm{fla}}$ holds such that $t_1, \ldots, t_n$ are all connected with $n > 1$, then $t_1 \# \cdots \# t_n \in \mathcal{N}^{\mathrm{fla}}$ holds and we say $t_1 \# \cdots \# t_n$ is an unconnected term.*

If no term variable occurs in t then we say t is *closed*. Hereafter, we call a term of the form $\mathsf{h}(t, s)$ an h-*term*. We define the ordering $\geq_{\mathcal{F}}$ on $\mathcal{F}$ as $\mathsf{h} >_{\mathcal{F}} \# >_{\mathcal{F}} 1$. The recursive path ordering on $\mathcal{N}^{\mathrm{fla}}$ is a well quasi ordering by Theorem 1. Below we denote the recursive path ordering on $\mathcal{N}^{\mathrm{fla}}$ by $\succeq_{\mathrm{rpo}}$ as well. Put $(t, s) := \mathsf{h}(t, s)$ and for any $t = t_1 \# \cdots \# t_n$ and $s = s_1 \# \cdots \# s_m$ $(n, m \geq 1)$, $t \underline{\#} s := t_1 \# \cdots \# t_n \# s_1 \# \cdots \# s_m$.

The ordering $\succeq_{\mathrm{rpo}}$ on $\mathcal{N}^{\mathrm{fla}}$ is incompatible with the set E_H of the following two equations on $\mathcal{N}^{\mathrm{fla}}$, which we call *Howard algebra* (cf. [9, Axioms 2.8 and 2.13 in p.448]): For any $n \geq 1$ and $m \geq 1$,

1. $(x, y_1 \# \cdots \# y_n) = (x, y_1) \# \cdots \# (x, y_n)$.
2. $(x_1 \# \cdots \# x_n, (y_1 \# \cdots \# y_m, z)) = (x_1 \# \cdots \# x_n \# y_1 \# \cdots \# y_m, z)$.

Proposition 1 (Incompatibility of $\succeq_{\mathrm{rpo}}$ And E_H). *The ordering $\succeq_{\mathrm{rpo}}$ and Howard algebra E_H are incompatible on $\mathcal{N}^{\mathrm{fla}}$.*

Proof. First, we consider the equation 1 above. For any $t, s_1, \ldots, s_n \in \mathcal{N}^{\mathrm{fla}}$ where each s_i is connected, we have $(t, s_1 \# \cdots \# s_n) \succ_{\mathrm{rpo}} (t, s_1) \# \cdots \# (t, s_n)$, so this is incompatible with 1. Next, consider the equation 2. In this case, we have $(1, (1, 1)) \succ_{\mathrm{rpo}} (1 \# 1, 1)$ while $E_H \vdash (1, (1, 1)) = (1 \# 1, 1)$ holds. We also have $((1, (1, 1)) \# 1, 1) \succ_{\mathrm{rpo}} ((1, (1, 1)), (1, 1))$ and $E_H \vdash ((1, (1, 1)) \# 1, 1) = ((1, (1, 1)), (1, 1))$.

[7] Though Howard also used the constant 0, we omit it since it is not necessary and this makes the formulation of our path ordering simple.

When a set E of equations on $\mathcal{T}(\mathcal{F},\mathcal{X})$ is given, we say a set of rewrite rule R is an *oriented set* of E if $R = \{F(t=s) \mid t=s \in E\}$ holds for some mapping F assigning a rewrite rule $t \to s$ or $s \to t$ to each equation $t = s$ in E. To define a path ordering with Howard algebra, we consider the oriented set H of E_H which is obtained by giving to the equations in E_H the direction from the left-hand side to the right-hand side. As usual, a rewrite relation R is *convergent* if and only if R is terminating and confluent. One can verify that the rewrite relation $\to_H$ is terminating by defining the polynomial interpretation $\tau(t)$ for any closed term t in $\mathcal{N}^{\mathrm{fla}}$ as $\tau(1) := 2$, $\tau((t,s)) := \tau(t)\cdot\tau(s)$, $\tau(t_1\#\cdots\#t_n) := \tau(t_1)+\cdots+\tau(t_n)+1$. The confluence of $\to_H$ can be shown by Knuth-Bendix's Critical Pair Lemma, so $\to_H$ is convergent. By the convergence of $\to_H$, any term has a unique H-normal form. We denote the H-normal form of t by $\mathrm{nf}(t)$. Note that the H-normal form s of a term $t \in \mathcal{N}^{\mathrm{fla}}$ is a term such that any h-subterm of s is of the form either (u,x) or $(u,1)$. By defining $f_H([t]_{R^H}) := \mathrm{nf}(t)$, where R^H is the equivalence relation induced by E_H, we have a choice function f_H for the set of equivalence classes with respect to R^H. By Lemma 1, we have the following proposition.

Proposition 2 (The Path Ordering $\succeq_{\mathrm{nf}}$ with Howard Algebra). *The path ordering $(\mathcal{N}^{\mathrm{fla}},\succeq_{\mathrm{nf}})$ with Howard algebra E_H with respect to f_H is a well quasi ordering on $\mathcal{N}^{\mathrm{fla}}$ such that $\succeq_{\mathrm{nf}}$ and E_H are compatible on $\mathcal{N}^{\mathrm{fla}}$.*

For any $t \in \mathcal{N}$, we define $\overline{t} \in \mathcal{N}^{\mathrm{fla}}$ as $\overline{x} := x$, $\overline{1} := 1$, $\overline{(t,s)} := (\overline{t},\overline{s})$, $\overline{t_1\#t_2} := \overline{t_1}\#\overline{t_2}$. Then, we extend $\succeq_{\mathrm{nf}}$ to $\mathcal{N}$ by defining $t \succeq_{\mathrm{nf}} s$ for $t,s \in \mathcal{N}$ as $\overline{t} \succeq_{\mathrm{nf}} \overline{s}$. The well quasi ordering $\succeq_{\mathrm{nf}}$ on $\mathcal{N}$ will be extended to the well founded ordering $\succeq_{\mathrm{po}}$ on $\mathcal{N}^{<\omega}$ further in the next subsection. Since the choice function above was defined via the normal form function, one can show that $\succeq_{\mathrm{nf}}$ satisfies several conditions which are listed in Lemma 2 below.[8] This guarantees that one can use $\succeq_{\mathrm{nf}}$ in a way similar to Howard's original ordering. For any term t, we define a substitution $t[x := s]$ of s for x in t in a usual way.

Lemma 2. *For any $t,t_1,s,u \in \mathcal{N}$, the following statements hold.*

1. *If $t \succ_{\mathrm{nf}} s$ and $t \succ_{\mathrm{nf}} u$ hold, then $(t,t_1) \succ_{\mathrm{nf}} (s,t_1)\#(u,t_1)$ holds.*
2. *If $t \succ_{\mathrm{nf}} s$ holds, then $(t,u) \succ_{\mathrm{nf}} (s,u)$ and $(u,t) \succ_{\mathrm{nf}} (u,s)$ hold.*
3. *If $t \succ_{\mathrm{nf}} s$ holds, then $t[x := u] \succ_{\mathrm{nf}} s[x := u]$ holds.*
4. *$(s,t) \succ_{\mathrm{nf}} t$ and $(t,s) \succ_{\mathrm{nf}} t$ hold.*

2.2 Non-Unique Assignments of the Vector System

In this subsection, we finish a termination proof of $\lambda_{\to}$ based on our path ordering $(\mathcal{N},\succeq_{\mathrm{nf}})$. Once our path ordering was defined, the remaining parts of our proof are essentially the same as Howard's and so almost all proofs are omitted. Let $\mathcal{S}$ be a non-empty set of base types. *Types* are defined by induction:

1. Any $\sigma \in \mathcal{S}$ is a type and the level $\mathrm{lv}(\sigma)$ of σ is defined as $\mathrm{lv}(\sigma) := 0$.

[8] The statement 1. of Lemma 2 below corresponds to Axiom 2.9 in [9, p.448], the statement 2. corresponds to Axioms 2.10, 2.11 and the statement 3. corresponds to Axiom 4.1 of [9, p.457]. The statement 4. has no counterpart in [9].

2. If σ and τ are types, then $\sigma \to \tau$ is a type and the level $\mathrm{lv}(\sigma \to \tau)$ is defined as $\max\{\mathrm{lv}(\sigma)+1, \mathrm{lv}(\tau)\}$.

Below we use a set of $\lambda_{\to}$-variables such that each $\lambda_{\to}$-variable has a unique type and for any type σ, there are infinitely many $\lambda_{\to}$-variables of type σ enumerated as $X^0, X^1, \ldots, X^r, \ldots$. We denote $\lambda_{\to}$-variables X, Y, Z possibly with suffixes. Then, *typed $\lambda_{\to}$-terms* are defined by induction:

1. For any type σ, every variable of type σ is a $\lambda_{\to}$-term of type σ.
2. If M is a $\lambda_{\to}$-term of type τ and X is a variable of type σ, then $\lambda X.M$ is a $\lambda_{\to}$-term of type $\sigma \to \tau$.
3. If M is a $\lambda_{\to}$-term of type $\sigma \to \tau$ and N is a $\lambda_{\to}$-term of type σ, then MN is a term of type τ.

If no free $\lambda_{\to}$-variable occurs in a $\lambda_{\to}$-term M then we say M is *closed*. We often write a $\lambda_{\to}$-term M of type σ as M^{σ} to indicate the type of M. We treat $\to$ as right associative and application-terms MN as left associative. Substitution $M[X := N]$ of N for X in M and its simultaneous version are defined in a usual way, and we assume that when we write $M[X := N]$, no free $\lambda_{\to}$-variable in N is bound in $M[X := N]$. We denote by $\to_{\beta}$ the β-rewrite relation on λ-terms defined as follows: $M \to_{\beta} N$ holds if and only if for some $\lambda_{\to}$-term $(\lambda X.L_1)L_2$ and some position p in M, we have $M|_p = (\lambda X.L_1)L_2$ and $M[L_1[X := L_2]]_p = N$.

A finite sequence of $\mathcal{N}$-terms is called a vector and denoted by $\boldsymbol{t}, \boldsymbol{s}, \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ possibly with suffixes. Let $\mathcal{N}^{<\omega}$ be the set of all vectors. Howard defined the square operator $\Box$ and the delta operator δ^r on $\mathcal{N}^{<\omega}$ to define the vectors assigned to application terms MN and abstraction terms $\lambda X.M$, respectively. Since we slightly modified Howard's original set of expressions by eliminating the constant 0, we define the square operator below. However, our definition is essentially the same as Howard's (cf. [9, p.449]). The delta operator can be defined in the exactly same way as Howard's, so its definition will be omitted.

Definition 3. *We define the following notions concerning vectors.*

1. *For any vector $\boldsymbol{t} = \langle t_0, \ldots, t_n \rangle$, the level $\mathrm{lv}(\boldsymbol{t})$ of $\boldsymbol{t}$ is defined as $\mathrm{lv}(\boldsymbol{t}) := n$.*
2. *Let ε be the auxiliary symbol not being a term in $\mathcal{N}$ and define $\varepsilon \# t := t \# \varepsilon := t$. If a vector $\boldsymbol{t}$ is of the form $\langle t_0, \ldots, t_n \rangle$, then we define the i-th component $(\boldsymbol{t})_i$ of $\boldsymbol{t}$ for any $i \in \mathbb{N}$ as follows:*

$$(\boldsymbol{t})_i := \begin{cases} t_i, & \text{if } 0 \le i \le n, \\ \varepsilon, & \text{if } i > n. \end{cases}$$

3. *For any two vectors $\boldsymbol{t}$ and $\boldsymbol{s}$ with $\mathrm{lv}(\boldsymbol{t}) = \mathrm{lv}(\boldsymbol{s}) = n$, we define the sum $\boldsymbol{t} \# \boldsymbol{s}$ of $\boldsymbol{t}$ and $\boldsymbol{s}$ as the vector of level n such that for any i with $0 \le i \le n$, $(\boldsymbol{t} \# \boldsymbol{s})_i := (\boldsymbol{t})_i \# (\boldsymbol{s})_i$ holds.*

The auxiliary symbol ε is used to make the following definition simple.

Definition 4 (Square Operator). *Let $\boldsymbol{t}$ and $\boldsymbol{s}$ be arbitrary two vectors. By downward induction on $n = \max\{\mathrm{lv}(\boldsymbol{t}), \mathrm{lv}(\boldsymbol{s})\}$, we define the vector $\boldsymbol{t}\Box\boldsymbol{s}$ of level n as*

$$(\boldsymbol{t}\Box\boldsymbol{s})_i := \begin{cases} (\boldsymbol{t})_i\#(\boldsymbol{s})_i, & \text{if } i = n, \\ ((\boldsymbol{t}\Box\boldsymbol{s})_{i+1}, (\boldsymbol{t})_i\#(\boldsymbol{s})_i), & \text{if } 0 \leq i < n. \end{cases}$$

For the sake of brevity, hereafter we write $\succeq_{\mathrm{nf}}$ and $\succ_{\mathrm{nf}}$ as $\succeq$ and $\succ$, respectively. The lemma below is provable by downward induction on i.

Lemma 3. *Let $\boldsymbol{t}$ and $\boldsymbol{s}$ be arbitrary two vectors such that $\mathrm{lv}(\boldsymbol{t}) = \mathrm{lv}(\boldsymbol{s}) = n$ holds and $t_i \succeq s_i$ holds for any i with $0 \leq i \leq n$. If there exists $j \leq n$ such that $t_i \succ s_i$ holds for any i with $0 \leq i \leq j$, then for any vector $\boldsymbol{u}$ and any i with $0 \leq i \leq j$, $(\boldsymbol{t}\Box\boldsymbol{u})_i \succ (\boldsymbol{s}\Box\boldsymbol{u})_i$ holds.*

We denote $t\#t$ by $2t$ for any term t. Lemma 4 below is essentially the same as [9, Lemma 2.5, 2.6], so we omit its proof. It is crucial for the proof of Lemma 4.(2) that $\succeq$ and E_H are compatible on $\mathcal{N}^{\mathrm{fla}}$ (cf. Proposition 2 above).

Lemma 4. *Let $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{t}, \boldsymbol{s}, \boldsymbol{u}$ be vectors with $\mathrm{lv}(\boldsymbol{a}) = \mathrm{lv}(\boldsymbol{b}) = \mathrm{lv}(\boldsymbol{t}) = \mathrm{lv}(\boldsymbol{s}) = n > \mathrm{lv}(\boldsymbol{u})$. Then, the following statements hold.*

1. *For any i with $0 \leq i \leq n$, $((\boldsymbol{a}\#\boldsymbol{b})\Box\boldsymbol{u})_i \succeq ((\boldsymbol{a}\Box\boldsymbol{u})\#(\boldsymbol{b}\Box\boldsymbol{u}))_i$ holds.*
2. *For any vector $\boldsymbol{v}$, if $(\boldsymbol{v})_n \succ 2(\boldsymbol{t})_n\#2(\boldsymbol{s})_n$ holds and $(\boldsymbol{v})_i \succeq (\boldsymbol{t})_i\#(\boldsymbol{s})_i$ holds for any i with $0 \leq i < n$, then $(\boldsymbol{v}\Box\boldsymbol{u})_i \succ 2((\boldsymbol{t}\Box\boldsymbol{u})\Box(\boldsymbol{s}\Box\boldsymbol{u}))_i$ holds for any i with $0 \leq i \leq n$.*

Below we assume an injection from $\lambda_{\to}$-variables to vectors of $\mathcal{N}$-variables which maps a $\lambda_{\to}$-variable X^r of type σ to a vector of $\mathcal{N}$-variables $\boldsymbol{x}^r := \langle x_0^r, \ldots, x_n^r\rangle$ with $n = \mathrm{lv}(\sigma)$. For any $t \in \mathcal{N}$ (resp. any vector $\boldsymbol{t}$), we define a simultaneous substitution $t[x_0 := s_0, \ldots, x_n := s_n]$ (resp. $\boldsymbol{t}[x_0 := s_0, \ldots, x_n := s_n]$) in a usual way. Let $\boldsymbol{s}$ be a vector with $\mathrm{lv}(\boldsymbol{s}) = n$. A substitution $t[\boldsymbol{x}^r := \boldsymbol{s}]$ with $\boldsymbol{x}^r = \langle x_0^r, \ldots, x_n^r\rangle$ is defined as $t[\boldsymbol{x}^r := \boldsymbol{s}] := t[(\boldsymbol{x}^r)_0 := (\boldsymbol{s})_0, \ldots, (\boldsymbol{x}^r)_n := (\boldsymbol{s})_n]$. A substitution $\boldsymbol{t}[\boldsymbol{x}^r := \boldsymbol{s}]$ with $\boldsymbol{x}^r = \langle x_0^r, \ldots, x_n^r\rangle$ is defined as $(\boldsymbol{t}[\boldsymbol{x}^r := \boldsymbol{s}])_i := t_i[\boldsymbol{x}^r := \boldsymbol{s}]$ for any i with $0 \leq i \leq \mathrm{lv}(\boldsymbol{t})$.

Take the subset $\mathcal{C}_i$ of $\mathcal{N}$ for each $i \in \mathbb{N}$ and the subset $\boldsymbol{C}$ of $\mathcal{N}^{<\omega}$, which are defined in [9, p.451]. By using these sets, one can define the delta operator $\delta^r t$ for each $t \in \mathcal{C}_i$ and the delta operator $\delta^r\boldsymbol{t}$ for each $\boldsymbol{t} \in \boldsymbol{C}$ in the same way as Howard's (cf. [9, p.452]). Here $\delta^r\boldsymbol{t}$ is intended to serve as the interpretation of $\lambda X^r.M$ where $\boldsymbol{t}$ is a vector assigned to M. Note that the factor 2 in the definition clause (d) of [9, p.452] is crucial to use Lemma 4.(2) in a proof of Lemma 5.(1) below, which is the key lemma to prove β-reduction decreases the order on vectors. Lemma 5 is provable as in [9], so its proof is omitted.[9]

Lemma 5. *We have the following propositions.*

1. *For any $t \in \mathcal{C}_i$, any $\boldsymbol{x}^r, \boldsymbol{s}$ with $\mathrm{lv}(\boldsymbol{x}^r) = \mathrm{lv}(\boldsymbol{s})$ and any $i \leq \mathrm{lv}(\boldsymbol{x}^r) + 1$, $((\delta^r t)\Box\boldsymbol{s})_i \succ t[\boldsymbol{x}^r := \boldsymbol{s}]$ holds.*

[9] Lemma 5.(1) and (2) correspond to Lemma 2.11 and its corollary in [9], respectively.

2. *For any* $\boldsymbol{t} \in \boldsymbol{C}$ *and any* $\boldsymbol{x}^r, \boldsymbol{s}$ *with* $\mathrm{lv}(\boldsymbol{x}^r) = \mathrm{lv}(\boldsymbol{s})$, $((\delta^r \boldsymbol{t})\Box \boldsymbol{s})_i \succ (\boldsymbol{t}[\boldsymbol{x}^r := \boldsymbol{s}])_i$ *holds for any* i *with* $0 \leq i \leq \mathrm{lv}(\boldsymbol{t})$.

We define our path ordering $\succeq_{\mathrm{po}}$ on $\mathcal{N}^{<\omega}$ as follows: For any two vectors $\boldsymbol{t}$ and $\boldsymbol{s}$ with $\mathrm{lv}(t) \geq \mathrm{lv}(s)$, $\boldsymbol{t} \succ_{\mathrm{po}} \boldsymbol{s}$ holds if and only if $(\boldsymbol{t})_0 \succ (\boldsymbol{s})_0$ holds and $(\boldsymbol{t})_i \succeq (\boldsymbol{s})_i$ holds for any i with $1 \leq i \leq \mathrm{lv}(s)$. Then, put $\boldsymbol{t} \succeq_{\mathrm{po}} \boldsymbol{s}$ as $\boldsymbol{t} \succ_{\mathrm{po}} \boldsymbol{s} \vee \boldsymbol{t} = \boldsymbol{s}$. For any vector $\boldsymbol{t}$ of level n and any i with $0 \leq i \leq n$, the restriction $\boldsymbol{t} \restriction_i$ of $\boldsymbol{t}$ up to i is defined as $\boldsymbol{t} \restriction_i := \langle t_0, \ldots, t_i \rangle$. For any $\boldsymbol{x}^r = \langle x_0^r, \ldots, x_n^r \rangle$, we denote $\boldsymbol{t}[x_0^r := 1, \ldots, x_n^r := 1]$ by $\boldsymbol{t}[\boldsymbol{x}^r := \boldsymbol{1}]$. Below we define the non-unique assignments of vectors in $\boldsymbol{C}$ to $\lambda_{\to}$-terms by adapting Wilken and Weiermann's *assignment derivations* (cf. [14, Definition 3.1]) to our setting.[10]

Definition 5 (Assignment Derivations). *Assignment derivations, which assign vectors in* $\boldsymbol{C}$ *to* $\lambda_{\to}$*-terms non-uniquely, are derivations constructed by the following axioms and rules.*

Axioms. *The following axiom is an assignment derivation for* $X^r \mapsto \boldsymbol{x}^r$.

$$\frac{}{X^r \mapsto \boldsymbol{x}^r}$$

Application Rule. *If* d_0 *and* d_1 *are assignment derivations for* $M^{\sigma\to\tau} \mapsto \boldsymbol{a}$ *and* $N^\sigma \mapsto \boldsymbol{b}$, *respectively, then*

$$\frac{\begin{array}{cc}\vdots\, d_0 & \vdots\, d_1 \\ M \mapsto \boldsymbol{a} & N \mapsto \boldsymbol{b}\end{array}}{MN \mapsto (\boldsymbol{a}\Box\boldsymbol{b}) \restriction_{\mathrm{lv}(\tau)}}$$

is an assignment derivation for $MN \mapsto (\boldsymbol{a}\Box\boldsymbol{b}) \restriction_{\mathrm{lv}(\tau)}$.

Abstraction Rule. *If for some* $k \geq 0$, $d_0, \ldots, d_k$ *are assignment derivations for* $D_0 \mapsto \boldsymbol{d}^0, \ldots, D_k \mapsto \boldsymbol{d}^k$, *respectively, such that* $D_0 \to_\beta \cdots \to_\beta D_k = L$ *and* $\boldsymbol{d}^0 \succ_{\mathrm{po}} \cdots \succ_{\mathrm{po}} \boldsymbol{d}^k$ *hold, then*

$$\frac{\begin{array}{ccc}\vdots\, d_0 & & \vdots\, d_k \\ D_0 \mapsto \boldsymbol{d}^0 & \cdots & D_k \mapsto \boldsymbol{d}^k\end{array}}{\lambda X^r.L \mapsto (\delta^r \boldsymbol{d}^0)\#\boldsymbol{d}^k[\boldsymbol{x}^r := \boldsymbol{1}]}$$

is an assignment derivation for $\lambda X^r.L \mapsto (\delta^r \boldsymbol{d}^0)\#\boldsymbol{d}^k[\boldsymbol{x}^r := \boldsymbol{1}]$.

We say $\boldsymbol{a}$ *is a vector of a* $\lambda_{\to}$*-term* M *if and only if there is an assignment derivation for* $M \mapsto \boldsymbol{a}$.

[10] Assignment derivations were introduced to formulate Howard-style non-unique assignments of vectors perspicuously. Note that Wilken and Weiermann not only introduced assignment derivations but also refined Howard's original non-unique assignments to cope with arbitrary recursor reduction rules of T. We restrict ourself to $\lambda_{\to}$ here, so we need not use the refined part by Wilken and Weiermann.

Because of Abstraction Rule, a $\lambda_{\to}$-term M with λ-abstraction subterms can have several assignment derivations which give to M different vectors.

Lemma 6 below corresponds to [9, Lemma 3.1] (cf. also [14, Lemma 3.3]), and Proposition 3 below corresponds to [9, Theorem 4.1] (cf. also [14, Theorem 3.5]). Though one can prove these lemma and proposition in the same way as [14], we prove the assertion in the crucial cases of Proposition 3 for readers' convenience.

Lemma 6. *Let $\boldsymbol{t}$ and $\boldsymbol{s}$ be vectors of M and N, respectively. If M and a $\lambda_{\to}$-variable X^r have the same type σ, then $\boldsymbol{s}[\boldsymbol{x}^r := \boldsymbol{t}]$ is a vector of $N[X^r := M]$.*

Proposition 3. *Let M and N be arbitrary two $\lambda_{\to}$-terms of the same type σ with $M \to_\beta N$. For any vector $\boldsymbol{t}$ of M, there is a vector $\boldsymbol{s}$ of N such that $\boldsymbol{t} \succ_{\mathrm{po}} \boldsymbol{s}$ holds.*

Proof. By induction on the build-up of M. First, assume that $M = (\lambda X^r.L_1)L_2$ and $L_1[X^r := L_2] = N$ holds. Let a given vector of $\lambda X^r.L_1$ be $(\delta^r \boldsymbol{d}^0)\#\boldsymbol{d}^k[\boldsymbol{x}^r := \boldsymbol{1}]$ and a given vector of L_2 be $\boldsymbol{a}$. Then, we have

$$\begin{aligned}((\delta^r \boldsymbol{d}^0)\#\boldsymbol{d}^k[\boldsymbol{x}^r := \boldsymbol{1}])\Box\boldsymbol{a} &\succ_{\mathrm{po}} (\delta^r \boldsymbol{d}^0)\Box\boldsymbol{a} \quad \text{By Lemma 3}\\ &\succ_{\mathrm{po}} \boldsymbol{d}^0[\boldsymbol{x}^r := \boldsymbol{a}] \quad \text{By Lemma 5.(2)}\\ &\succ_{\mathrm{po}} \boldsymbol{d}^k[\boldsymbol{x}^r := \boldsymbol{a}] \quad \text{By Lemma 2.(3),}\end{aligned}$$

hence we have the assertion since $\boldsymbol{d}^k[\boldsymbol{x}^r := \boldsymbol{a}]$ is a vector of N by Lemma 6.

Next, assume that $M = \lambda X^r.L_1 \to_\beta \lambda X^r.L_2 = N$ holds and an assignment derivation d for $M \mapsto \boldsymbol{t}$ is given. Then, d consists of the immediate subderivations d_i of $D_i \mapsto \boldsymbol{d}^i$ for all i with $0 \le i \le k$ such that $D_k = L_1$ and $\boldsymbol{d}^0 \succ_{\mathrm{po}} \cdots \succ_{\mathrm{po}} \boldsymbol{d}^k$ hold. Since we have $L_1 \to_\beta L_2$, there is an assignment derivation d' of $L_2 \mapsto \boldsymbol{a}$ with $\boldsymbol{d}^k \succ_{\mathrm{po}} \boldsymbol{a}$ by IH. Therefore, by Lemma 2.(3), we can construct an assignment derivation of $\lambda X^r.L_2 \mapsto \boldsymbol{s}$ from $d_0, \ldots, d_k$ and d' such that the assertion $\boldsymbol{t} = (\delta^r \boldsymbol{d}^0)\#\boldsymbol{d}^k[\boldsymbol{x}^r := \boldsymbol{1}] \succ_{\mathrm{po}} (\delta^r \boldsymbol{d}^0)\#\boldsymbol{a}[\boldsymbol{x}^r := \boldsymbol{1}] = \boldsymbol{s}$ holds.

Corollary 1. *Simply typed λ-calculus $\lambda_{\to}$ is terminating.*

Proof. We can easily assign a vector to a given $\lambda_{\to}$-term M by putting $k = 0$ in all cases of Abstraction Rule. Therefore, any reduction sequence terminates by Propositions 2 and 3.

3 Extension to Some Combined Systems

In this section, by using our path ordering $(\mathcal{N}^{<\omega}, \succeq_{\mathrm{po}})$, we first sketch termination proofs of two combined systems: The first is combined with the map function and the second is combined with the Ackermann function. Next, we apply our path ordering method to Wilken and Weiermann's termination proof for the system T of primitive recursive functionals of finite type.

We start from a sketch of termination proof for the combined system of $\lambda_{\to}$ and rewrite rules for the map function. In this case, the set $\mathcal{S}$ of base types consists of the type $\mathbb{N}$ of natural numbers and the type List of lists of natural numbers. We add the following clauses to extend the set of λ-terms:

i. $\mathsf{0}$ is a λ-term of type $\mathbb{N}$, and if M is a λ-term of type $\mathbb{N}$ then $\mathsf{S}(M)$ is a λ-term of type $\mathbb{N}$,
ii. nil is a λ-term of type List, and if M is a λ-term of type $\mathbb{N}$ and F, G are λ-terms of type List then $\mathsf{cons}(M, F)$ and $\mathsf{append}(F, G)$ are λ-terms of type List,
iii. if M is a λ-term of type $\mathbb{N} \to \mathbb{N}$ and N is a closed λ-term of type List then $\mathsf{map}(M, N)$ is a λ-term of type List.

Note that $\mathsf{S}(M), \mathsf{cons}(M, F), \mathsf{append}(F, G)$ and $\mathsf{map}(M, N)$ are constructed not by λ-application but by application of the function symbols $\mathsf{S}, \mathsf{cons}, \mathsf{append}$ and map. Our set of rewrite rules for the map function consists of the following rules:

1. $\mathsf{append}(\mathsf{nil}, M) \to M$,
2. $\mathsf{append}(\mathsf{cons}(M, N), L) \to \mathsf{cons}(M, \mathsf{append}(N, L))$,
3. $\mathsf{append}(\mathsf{append}(M, N), L) \to \mathsf{append}(M, \mathsf{append}(N, L))$,
4. $\mathsf{map}(M, \mathsf{nil}) \to \mathsf{nil}$,
5. $\mathsf{map}(M, \mathsf{cons}(N, L)) \to \mathsf{cons}(MN, \mathsf{map}(M, L))$.

Due to the algebraic rule 3. for append, these rules are not purely recursive ones. By the definition of λ-terms, N and L in the rule 5. are always closed. This restriction is needed to guarantee that a vector of $\mathsf{map}(M, N)$ below belongs to $\boldsymbol{C}$ and so Lemma 6 holds. Then, we add the following axioms and rules of assignment derivations.

$$\frac{}{\mathsf{0} \mapsto \langle 1 \rangle} \qquad \frac{M \mapsto \boldsymbol{a}}{\mathsf{S}(M) \mapsto \langle (\boldsymbol{a})_0 \# 1 \rangle} \qquad \frac{}{\mathsf{nil} \mapsto \langle 1 \rangle} \qquad \frac{M \mapsto \boldsymbol{a} \quad N \mapsto \boldsymbol{b}}{\mathsf{cons}(M, N) \mapsto \langle (\boldsymbol{a})_0 \# (\boldsymbol{b})_0 \rangle}$$

$$\frac{M \mapsto \boldsymbol{a} \quad N \mapsto \boldsymbol{b}}{\mathsf{append}(M, N) \mapsto \langle 2(\boldsymbol{a})_0 \# (\boldsymbol{b})_0 \rangle} \qquad \frac{M \mapsto \boldsymbol{a} \quad N \mapsto \boldsymbol{b}}{\mathsf{map}(M, N) \mapsto \langle ((\boldsymbol{a})_1 \# (\boldsymbol{b})_0, (\boldsymbol{a})_0 \# (\boldsymbol{b})_0) \rangle}$$

Since the rewrite rules 1.–5. decrease the order $\succ_{\mathrm{po}}$ and the monotonicity holds for $\mathsf{S}, \mathsf{cons}, \mathsf{append}$ and map on these assignments, one can prove the termination of the combined system above by the results of the previous section.

Next, we sketch a termination proof of the combined system of $\lambda_{\to}$ and rewrite rules for the Ackermann function without recursors. Take the set $\mathcal{S}$ of base types as $\{\mathbb{N}\}$. In addition to the clause i. above, we also consider the following clause: If M, N are λ-terms with M closed, then $\mathsf{ack}(M, N)$ is a λ-term of type $\mathbb{N}$. We call a λ-term of the form $\underbrace{\mathsf{S}(\cdots \mathsf{S}}_{n\ times}(\mathsf{0}) \cdots)$ a *numeral* and denote it by n. We consider the following rewrite rules for the Ackermann function[11]:

1. $\mathsf{ack}(\mathsf{0}, N) \to \mathsf{S}(N)$,
2. $\mathsf{ack}(\mathsf{S}(M), \mathsf{n}) \to$
$(\lambda X_0.\mathsf{ack}(M, X_0))\Big(\cdots (\lambda X_{n-1}.\mathsf{ack}(M, X_{n-1}))\Big((\lambda X_n.\mathsf{ack}(M, X_n))\mathsf{1}\Big)\cdots\Big)$.

[11] The rule 2. below is a higher-order version of a usual rule $\mathsf{ack}_{m+1}(n) \to \mathsf{ack}_m^{n+1}(1)$ with $f^0(k) := k$ and $f^{n+1}(k) := f(f^n(k))$.

These rules are not purely recursive ones in the respect that they are written without any recursor and iterator. By using the following rule of assignment derivations, one can prove the termination of the combined system in this case as well.

$$\frac{M \mapsto \boldsymbol{a} \quad N \mapsto \boldsymbol{b}}{\mathsf{ack}(M,N) \mapsto \langle((1,(\boldsymbol{a})_0),(\boldsymbol{b})_0)\rangle}$$

Finally, we apply our path ordering method to Wilken and Weiermann's termination proof for the system T of primitive recursive functionals of finite type ([14]). Their proof is a refinement of Howard's proof in [9]: A collapsing function was incorporated into Wilken-Weiermann's vector system and they used this machinery to give the strong normalization theorem of T and an optimal derivation lengths classification of T, both of which were not given in Howard's proof. On the other hand, Wilken and Weiermann maintained Howard's two step mappings. The system T is embedded into their vector system $\mathcal{O}^{<\omega}$, and $\mathcal{O}^{<\omega}$ was interpreted in an ordinal notation system up to ε_0 to guarantee the well foundedness of the ordering on $\mathcal{O}^{<\omega}$. Below we note that these two step mappings can be simplified into a single mapping as well.

Let $\mathcal{X}_{\mathcal{O}}$ be an infinite set of ordinal variables, and $\mathcal{F}_{\mathcal{O}}$ be a signature consists of the constants $0, 1, \omega$, the unary function symbols $2^x, \omega^x, \Psi(x)$, the binary function symbols $x+y, x\#y, x\otimes y$. First of all, we define the subset $\mathcal{O}$ of $\mathcal{T}(\mathcal{F}_{\mathcal{O}}, \mathcal{X}_{\mathcal{O}})$ in a way similar to [14, Definition 2.1]: (i) Any ordinal variable belongs to $\mathcal{O}$, (ii) the constants $0, 1, \omega$ belong to $\mathcal{O}$ and (iii) if $t, s, u \in \mathcal{O}$ holds with $t, s \neq 0$, then $(t\#s), (2^u \otimes s), \Psi((\omega \otimes t)\#s) \in \mathcal{O}$ holds. Here we write $x+y$, $x\cdot y$ and $\psi(x)$ in [14] as $x\#y$, $x\otimes y$ and $\Psi(x)$, respectively. Terms of the form $2^u \otimes s$ correspond to Howard's original interpretation of his function symbol $(\cdot,\cdot)$ ([9, p.449]).

Next, we define the two other subsets $\mathcal{O}^{\mathrm{fla}}, \mathcal{O}^{\mathrm{nf}}$ of $\mathcal{T}(\mathcal{F}_{\mathcal{O}}, \mathcal{X}_{\mathcal{O}})$, treating $\#$ as a variadic function symbol. As in the previous section, we write $\#(t_1, \ldots, t_n)$ as $t_1\# \cdots \#t_n$. Note that $\mathcal{O}^{\mathrm{fla}}$ and $\mathcal{O}^{\mathrm{nf}}$ do not contain any ordinal variable and any term whose form is either $0 \otimes t$, $t \otimes 0$, $0 + t$, $t + 0$ or $\cdots\#0\#\cdots$.

Definition 6 (The Sets $\mathcal{O}^{\mathrm{fla}}$ and $\mathcal{O}^{\mathrm{nf}}$). *The set $\mathcal{O}^{\mathrm{fla}}$ of flattened ordinal terms is defined by induction.*

1. *The constant 0 belongs to $\mathcal{O}^{\mathrm{fla}}$, and we say 0 is an unconnected term.*
2. *If $t, s \in \mathcal{O}^{\mathrm{fla}}$ holds with $t, s \neq 0$, then $t+s, t\otimes s \in \mathcal{O}^{\mathrm{fla}}$ holds and we say they are connected terms.*
3. *If $t \in \mathcal{O}^{\mathrm{fla}}$ holds, then $\omega^t \in \mathcal{O}^{\mathrm{fla}}$ holds and we say ω^t is a connected term.*
4. *If $t \in \mathcal{O}^{\mathrm{fla}}$ holds, then $\Psi(t), 2^t \in \mathcal{O}^{\mathrm{fla}}$ holds and we say they are connected terms.*
5. *If $t_1, \ldots, t_n \in \mathcal{O}^{\mathrm{fla}}$ holds such that $t_1, \ldots, t_n$ are all connected with $n > 1$, then $t_1\# \cdots \#t_n \in \mathcal{O}^{\mathrm{fla}}$ holds and we say it is an unconnected term.*

The set $\mathcal{O}^{\mathrm{nf}}$ of normal flattened ordinal terms are defined by induction with the clauses 1., 3. and 5. above.

Let $\succeq_{\mathrm{rpo}}$ be the recursive path ordering on $\mathcal{O}^{\mathrm{nf}}$ with the precedence relation $\omega^x > \# > 0$. For any finite set $K = \{\alpha_1, \ldots, \alpha_n\} \subseteq \mathcal{O}^{\mathrm{nf}}$, we denote by π_K

a fixed permutation on K such that $\pi_K(\alpha_1) \succeq_{\mathrm{rpo}} \cdots \succeq_{\mathrm{rpo}} \pi_K(\alpha_n)$ holds. We treat $+$ as left associative, and abbreviate ω^0 as 1, and ω^1 as ω, hence $1, \omega \in \mathcal{O}$ are constants while $1, \omega \in \mathcal{O}^{\mathrm{nf}}$ are abbreviations of ω^0 and ω^{ω^0}, respectively. In addition to this, $\widetilde{n}$ denotes $\underbrace{\omega^0 + \omega^0 + \cdots + \omega^0}_{n \text{ times}}$ for any positive integer n, $\omega^{1+\widetilde{0}}$ denotes ω^1 and $\omega^{\widetilde{0}}$ denotes ω^0. Define $\mathcal{O}^{\mathrm{nf}} \restriction \omega := \{t \in \mathcal{O}^{\mathrm{nf}} \mid \omega \succ_{\mathrm{rpo}} t\}$.

Following [14, p.15], we define a norm function and a collapsing function on $\mathcal{O}^{\mathrm{nf}}$. For any $t \in \mathcal{O}^{\mathrm{nf}}$, the norm function $\mathrm{no} : \mathcal{O}^{\mathrm{nf}} \to \mathbb{N}$ is defined by induction on the build-up of t: $\mathrm{no}(0) := 0$ and $\mathrm{no}(\omega^{t_1} \# \cdots \# \omega^{t_n}) := n + \mathrm{no}(t_1) + \cdots + \mathrm{no}(t_n)$ with $n \geq 1$. Then, define the number theoretic functions $F_n : \mathbb{N} \to \mathbb{N}$ by $F_0(x) := 2^x$ and $F_{n+1}(x) := F_n^{x+1}(x)$. We fix a fast growing number theoretic function $\Phi : \mathbb{N} \to \mathbb{N}$ such that Φ is bounded by some suitable F_k. The collapsing function $\psi : \mathcal{O}^{\mathrm{nf}} \to (\mathcal{O}^{\mathrm{nf}} \restriction \omega)$ is defined by induction on the order type of $(\mathcal{O}^{\mathrm{nf}}, \succ_{\mathrm{rpo}})$: $\psi(t) := \max(\{0\} \cup \{\psi(s) \underline{\#} 1 \mid t \succ_{\mathrm{rpo}} s,\ \mathrm{no}(s) \leq \Phi(\mathrm{no}(t))\})$.

Definition 7 (The Set E of Ordinal Equations). *The set E of ordinal equations on $\mathcal{O}^{\mathrm{fla}}$ consists of the following equations:*

1. $(t + s) + u = t + (s + u)$.
2. *For any* $K = \{t_1, \ldots, t_n\} \subseteq \mathcal{O}^{\mathrm{nf}}$,
   $\omega^{t_1} \# \cdots \# \omega^{t_n} = \omega^{\pi_K(t_1)} + \cdots + \omega^{\pi_K(t_n)}$.
3. *For any* $n, m \geq 1$, $(\omega^{t_1} \# \cdots \# \omega^{t_n}) \otimes (\omega^{s_1} \# \cdots \# \omega^{s_m}) =$
   $(\omega^{t_1 \# s_1} \# \cdots \# \omega^{t_1 \# s_m}) \# \cdots \# (\omega^{t_n \# s_1} \# \cdots \# \omega^{t_n \# s_m})$.
4. $2^0 = \omega^0$.
5. $2^{\widetilde{n}} = \widetilde{2^n}$.
6. *For any* k, m *with* $k + m \geq 1$ *and any* $t_1, \ldots, t_k$ *such that the form of each* t_i *is neither* 0 *nor* $\widetilde{n}$,
   $2^{\omega^{t_1} + \cdots + \omega^{t_k} + \omega^{1+\widetilde{n_1}} + \cdots + \omega^{1+\widetilde{n_m}} + \widetilde{n}} = \omega^{\omega^{t_1} + \cdots + \omega^{t_k} + \omega^{\widetilde{n_1}} + \cdots + \omega^{\widetilde{n_m}}} \otimes \widetilde{2^n}$.

We denote by E_W the set which consists of the equation 1. and 2. above.

Let W be the oriented set W of the equation 3.–6. on $\mathcal{O}^{\mathrm{fla}}$ which gives to these equations the direction from the left-hand side to the right-hand side. We define the W/E_W-rewrite relation $\to_{W'}$ on $\mathcal{O}^{\mathrm{fla}}$ as follows: For any $t, s \in \mathcal{O}^{\mathrm{fla}}$, $t \to_{W'} s$ holds if and only if for some $t', s' \in \mathcal{O}^{\mathrm{fla}}$, we have $E_W \vdash t = t'$, $t' \to_W s'$ and $E_W \vdash s' = s$, where $\to_W$ is the W-rewrite relation on $\mathcal{O}^{\mathrm{fla}}$. We consider the following interpretation of $\mathcal{O}^{\mathrm{fla}}$ in $\mathbb{N}$:

$$\tau(0) := 3, \quad \tau(2^t) := 2^{\tau(t)}, \quad \tau(\omega^t) := \tau(t) + 1, \quad \tau(\Psi(t)) := \tau(t),$$
$$\tau(t + s) := \tau(t) + \tau(s), \quad \tau(t \otimes s) := \tau(t) \cdot \tau(s),$$
$$\tau(t_1 \# \cdots \# t_n) := \tau(t_1) + \cdots + \tau(t_n).$$

Then, one can show that $\tau(t) = \tau(s)$ holds for any equation $t = s$ in E_W and that $\tau(t) > \tau(s)$ holds for any rule $t \to s$ from W. Since we have $\tau(u[t]_p) > \tau(u[s]_p)$ whenever $\tau(t) > \tau(s)$ holds, the rewrite relation $\to_{W'}$ is terminating. This rewrite relation is also convergent, because it is locally confluent and hence it is confluent.

For any $t \in \mathcal{O}$, let $\bar{t} \in \mathcal{O}^{\mathrm{fla}}$ be the flattened term of $t \in \mathcal{O}$ defined as in the previous section. Note that the occurrences of 1 and ω in t are replaced by ω^0 and ω^{ω^0} in $\bar{t}$, respectively. Due to the collapsing function ψ, we need to use the following normalizing function.

Definition 8 (Normalizing Function). *The normalizing function* $[\![\cdot]\!]$ *from closed terms of* $\mathcal{O}$ *to terms of* $\mathcal{O}^{\mathrm{nf}}$ *is defined as follows.*

- *The mapping* * *from* $\mathcal{O}^{\mathrm{fla}}$*-terms without* $+$ *to* $\mathcal{O}^{\mathrm{nf}}$ *is defined as follows.*
  $0^* := 0, \quad (2^t)^* := \mathrm{nf}(2^{(t^*)}), \quad (\omega^t)^* := \omega^{(t^*)}, \quad (\Psi(t))^* := \psi(t^*),$
  $(t \otimes s)^* := \mathrm{nf}(t^* \otimes s^*), \quad (t_1 \# \cdots \# t_n)^* := t_1^* \underline{\#} \cdots \underline{\#} t_n^*,$
  where $\mathrm{nf}(t)$ *is a fixed* $\to_{W'}$*-normal form* s *of* t *such that* $s \in \mathcal{O}^{\mathrm{nf}}$.
- *For any closed* $t \in \mathcal{O}$, $[\![t]\!] := \overline{t^*}$.

One can see that the definition above indeed gives $[\![t]\!] \in \mathcal{O}^{\mathrm{nf}}$ for any closed $t \in \mathcal{O}$, and this normalizing function gives the following well quasi ordering.

Definition 9. *For any closed* $t, s \in \mathcal{O}$, $t \succeq s$ *holds if and only if* $[\![t]\!] \succeq_{\mathrm{rpo}} [\![s]\!]$ *holds.*

For any $t, s \in \mathcal{O}$ such that t, s are not closed, one can define $t \succeq s$ by using substitutions as in [14, Definition 2.7], and the ordering $\succeq$ can be extended to $\mathcal{O}^{<\omega}$ as in [14, Definition 2.8]. Then, we have the well-founded ordering $(\mathcal{O}^{<\omega}, \succeq)$ in which the system T can be embedded by one step.

For higher-order termination proofs, which we discussed in this section, there are some RPO-style orderings such as Jouannaud and Rubio's higher-order recursive path ordering (HORPO) [12] and Blanqui, Jouannaud and Rubio's computability path ordering (CPO) [4]. The ordering HORPO was introduced as an extension of RPO to λ-terms: It is an RPO-style ordering defined for λ-terms directly. Then, CPO was introduced as a syntax-oriented counterpart of the combination of HORPO and the computability closure construction (as to the computability closure construction, see [2]), so the domain of CPO contains λ-terms as well. On the other hand, the domain of our path ordering with an algebra does not contain λ-terms and consists of equivalence classes of first-order terms. In addition to this difference, as far as we know, the well foundedness proofs of HORPO and CPO need sophisticated computability arguments: Variants of computability predicates were defined, and it was shown that HORPO and CPO satisfy certain properties similar to the ones satisfied by the β-reduction relation in a usual computability argument. In the case of our path ordering, its well foundedness immediately follows from the well-foundedness of RPO on which our path ordering is defined, hence one can see the well-foundedness of our path ordering more easily.

4 Concluding Remark and Future Work

We have introduced a first-order RPO-style ordering with Howard algebra, and we have given some examples of our method. Termination proofs by our method could be complementary to ones by HORPO or CPO when an interpretation of λ-abstraction and application is helpful. We shall investigate such examples so that our path ordering is useful to complement the HORPO-CPO method. We would also plan to see how our method is related to the theory of rewriting modulo equations, especially of rewriting logic (cf. [13]).

As we explained in Section 1, our motivation was to view Howard's work in [9] as a case study towards Lévy's open question. Our work in this paper was still a very small step to bridge Howard's work to this problem. We intend to work further on this topic by investigating other case studies.

References

1. Barbanera, F., Fernández, M., Geuvers, H.: Modularity of strong normalization and confluence in the algebraic-lambda-cube. In: Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994. pp. 406–415 (1994). https://doi.org/10.1109/LICS.1994.316049
2. Blanqui, F., Jouannaud, J.P., Okada, M.: Inductive-data-type systems. Theoretical Computer Science **272**(1), 41–68 (2002). https://doi.org/10.1016/S0304-3975(00)00347-9
3. Blanqui, F., Jouannaud, J.P., Okada, M.: Corrigendum to "inductive-data-type systems" [theoret. comput. sci. 272 (1–2) (2002) 41–68]. Theoretical Computer Science (2018). https://doi.org/10.1016/j.tcs.2018.01.010
4. Blanqui, F., Jouannaud, J., Rubio, A.: The computability path ordering: The end of a quest. In: Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings. pp. 1–14 (2008). https://doi.org/10.1007/978-3-540-87531-4_1
5. Dershowitz, N.: Orderings for term-rewriting systems. Theoretical Computer Science **17**, 279–301 (1982). https://doi.org/10.1016/0304-3975(82)90026-3
6. Dershowitz, N.: Termination of rewriting. Journal of Symbolic Computation **3**(1), 69–115 (1987). https://doi.org/10.1016/S0747-7171(87)80022-6
7. Dershowitz, N., Jouannaud, J.P.: Rewrite systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science (Vol. B), pp. 243–320. MIT Press, Cambridge, MA, USA (1990)
8. Dershowitz, N., Okada, M.: Proof-theoretic techniques for term rewriting theory. In: Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988. pp. 104–111 (1988). https://doi.org/10.1109/LICS.1988.5108
9. Howard, W.A.: Assignment of ordinals to terms for primitive recursive functionals of finite type. In: Kino, A., Myhill, J., Vesley, R.E. (eds.) Intuitionism and Proof Theory: Proceedings of the Summer Conference at Buffalo N.Y. 1968, Studies in Logic and the Foundations of Mathematics, vol. 60, pp. 443–458. Elsevier (1970). https://doi.org/10.1016/S0049-237X(08)70770-5
10. Jouannaud, J., Okada, M.: A computation model for executable higher-order algebraic specification languages. In: Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991. pp. 350–361 (1991). https://doi.org/10.1109/LICS.1991.151659
11. Jouannaud, J., Okada, M.: Abstract data type systems. Theoretical Computer Science **173**(2), 349–391 (1997). https://doi.org/10.1016/S0304-3975(96)00161-2
12. Jouannaud, J., Rubio, A.: Polymorphic higher-order recursive path orderings. J. ACM **54**(1), 2:1–2:48 (2007). https://doi.org/10.1145/1206035.1206037
13. 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
14. Wilken, G., Weiermann, A.: Derivation Lengths Classification of Gödel's T Extending Howard's Assignment. Logical Methods in Computer Science **8**(1) (2012). https://doi.org/10.2168/LMCS-8(1:19)2012

Strategies, model checking and branching-time properties in Maude*

Rubén Rubio, Narciso Martí, Isabel Pita, Alberto Verdejo

Universidad Complutense de Madrid, Spain
{rubenrub,narciso,ipandreu,jalberto}@ucm.es

Abstract. Maude 3.0 includes as a new feature an object-level strategy language. Rewriting strategies can now be used to easily control how rules are applied, restricting the rewriting systems behavior. This new specification layer would not be useful if there were no tools to execute, analyze and verify its creatures. For that reason, we extended the Maude LTL model checker to systems controlled by strategies, after studying their model-checking problem. Now, we widen the range of properties that can be checked in Maude models, both strategy-aware and strategy-free, by implementing a module for the language-independent model checker LTSmin that supports logics like CTL* and μ-calculus.

1 Introduction

The Maude [9] specification language has recently reached its 3.0 version, integrating new features developed during the last years, including a full implementation of the Maude strategy language [9, §10]. Although rewriting logic owes its natural representation of concurrency to the possibility that different rules can be executed in different positions at each step of the rewriting process, there are situations in which it is convenient to control such nondeterminism. This is the purpose of strategies, which have traditionally been expressed in Maude at the metalevel by means of its reflective features [10,11,25], but since the complexity and learning curve of programming metalevel computations is hard, an object-level strategy language design was proposed [19,13], exercised with different examples [28,24,20,26, ...], and finally added to the Core Maude functionality. Strategies can be described compositionally using strategy modules on top of system modules, and different commands are provided to rewrite a term following a strategy.

However, this new feature would be worthless without convenient tools to analyze the specifications using it. One of the most useful tools for verifying regular Maude modules is its LTL model checker [15]. In a previous work [23], we have studied the model-checking problem for rewriting systems controlled by strategies and presented an extension of the model checker to deal with them.

* Research partially supported by MCIU Spanish project *TRACES* (TIN2015-67522-C3-3-R). Rubén Rubio is partially supported by MCIU grant FPU17/02319.

However, since the original Maude model checker is limited to LTL properties, these are the only ones that our extension can handle and the discussion was mainly centered on linear-time properties. In this paper, we further discuss branching-time properties and show an implementation of a language plugin for the language-independent model checker LTSmin [16] that widens the range of logics in which properties can be expressed to CTL* and μ-calculus, for both the strategy-aware specifications and the regular ones. It can be downloaded from http://maude.ucm.es/strategies.

In the following sections, we briefly introduce the strategy language, the model-checking problem in this context, and the plugin we have developed. But let us first introduce a motivational example: the *river crossing* puzzle. In this classical game, a shepherd needs to cross a river carrying a wolf, a goat and a cabbage. The only way to cross it is using a boat that only the shepherd can operate and with room for only one more being. The shepherd could ship their companions to the other side one by one, but the wolf would eat the goat and the goat would eat the cabbage as soon as the shepherd is not present to impede it. The Maude signature of the problem is specified in a functional module:

```
fmod RIVER is
  sorts River Side Group .
  subsort Side < Group .

  op  _|_ : Group Group → River [ctor comm] .
  ops left right : → Side [ctor] .
  ops shepherd wolf goat cabbage : → Group [ctor] .
  ops __ : Group Group → Group [ctor assoc comm] .

  op initial : → River .
  eq initial = left shepherd wolf goat cabbage | right .
endfm
```

The system module `RIVER-CROSSING` completes the equational specification with rules: `alone`, `wolf`, `goat` and `cabbage` cause the shepherd to cross the river with the mentioned passenger, while `wolf-eats` and `goat-eats` make such animal eat its prey, which vanishes from the scene.

```
mod RIVER-CROSSING is
  protecting RIVER .

  vars G G' : Group .

  rl [wolf-eats] : goat wolf G | G' shepherd ⇒
                   wolf G | G' shepherd .
  rl [goat-eats] : cabbage goat G | G' shepherd ⇒
                      goat G | G' shepherd .

  rl [alone] : shepherd G | G' ⇒
               G | G' shepherd .
  rl [wolf] : shepherd wolf G | G' ⇒
```

```
                              G | G' shepherd wolf .
  rl [goat] : shepherd goat G | G' ⇒
                              G | G' shepherd goat .
  rl [cabbage] : shepherd cabbage G | G' ⇒
                           G | G' shepherd cabbage .
endm
```

The rules of the game tell that no character will miss the chance to claim its prey, so the eating rules must be applied before any other crossing if possible. This is not guaranteed in the system module, but expressing this restriction using strategies is easy, and we will see how in the following section.

In a previous specification of this problem in Maude [22], the eating rules were written as equations. While this alternative also ensures the discussed property according to the operational semantics of the Maude rewriting engine, it yields a rewrite theory where rules and equations are not coherent[1].

2 The Maude strategy language

As we have said in the Introduction, the Maude strategy language was born to allow expressing rewriting strategies without the difficulties of the metalevel. Its design is based on the experience with reflective computations, and on earlier strategy languages like ELAN [4] and Stratego [6].

A strategy α can be seen, if we look at its results, as a transformation from a term t into a set of terms, since the rewriting process controlled by α may still be nondeterministic. These results can be obtained within the interpreter using the `srewrite` t `using` α command. The most elementary strategy is rule application

$$\mathsf{top}(\mathit{label}[x_1 \texttt{<-} t_1, \ldots, x_n \texttt{<-} t_n]\{\alpha_1, \ldots, \alpha_m\}),$$

that executes any available rules with label *label* on any subterm of the subject term. An optional substitution can be specified between brackets to instantiate any occurrence of the variables x_k in the rule and its condition with t_k before matching, and to apply rules with rewriting conditions, strategies α_l must be provided to control each rewriting condition fragment. To restrict the application of the rule to the top of the subject term, top is available. A more powerful tool for selecting to which subterm a strategy is applied is the matchrew operator

$$\textbf{matchrew } P \textbf{ s.t. } C \textbf{ by } x_1 \textbf{ using } \alpha_1 \textbf{ , } \ldots \textbf{ , } x_n \textbf{ using } \alpha_n$$

It matches the pattern P on top of the subject term, and for each match satisfying the condition C, the subterms corresponding to the variables $x_1, \ldots, x_n$ are rewritten using the strategies $\alpha_1, \ldots, \alpha_n$, and reassembled again. The `matchrew`

[1] A rewrite theory is *coherent* if for all term t rewritten by a rule to a term t', its canonical form u modulo equations and axioms can be rewritten to a term u' that is equationally equivalent to t', see [9, §5.3]. Coherence is assumed by Maude, which reduces terms to their canonical forms before applying a rule, not to miss any rewrite.

keyword can be prefixed by `a` to match anywhere or `x` to match modulo structural axioms. The same variants exist for the tests **match** P **s.t.** C, to check if P matches the subject term and satisfies C. Regular expressions are included in the strategy language by means of the alternation $\alpha\,|\,\beta$, the concatenation $\alpha\,;\,\beta$, the Kleene star α^*, and the constants **idle** and **fail**. A conditional strategy $\alpha\,?\,\beta\,:\,\gamma$ is also available. It executes α and then β on its results, but if α does not produce any, it applies γ to the initial term. The language includes some other derived operators like α **or-else** β defined as α `? idle :` β or `not(`α`)` as α `? fail : idle`.

Using these combinators, we can guarantee that eating happens eagerly before traveling in the river crossing puzzle with the following strategy:

```
((wolf-eats | goat-eats)
 or-else (alone | cabbage | goat | wolf)) *
```

In each step of the iteration, which can stop nondeterministically at any time, the **or-else** combinator ensures that the crossing rules of its second argument are tried only if the eating rules in its first argument do not succeed. However, when strategies become more complex, writing long self-contained expressions is not practical. For example, the previous will be easier to understand if we name the first union of the expression as `eating` and the second as `oneCrossing`, `(eating or-else oneCrossing) *`. Strategy modules allow defining strategies, which can take parameters and call themselves recursively, extending the expressive power of the language. They are introduced by the `smod` keyword and may contain strategy declarations **strat** `sname : T1 ... Tn @ T` specifying its name and signature, and (possibly conditional) strategy definitions like **sd** `sname(`$t_1, \ldots, t_n$`) :=` α. A strategy call will execute all strategy definitions whose left-hand side matches the call term, instantiating the right-hand side expression with the variables bound in the left-hand side and the optional condition.

The following strategy module gives some strategy definitions for the river crossing problem:

```
smod RIVER-CROSSING-STRAT is
  protecting RIVER-CROSSING .

  var G : Group .

  strats oneCrossing eating @ River .
  sd oneCrossing := alone | wolf | goat | cabbage .
  sd eating := wolf-eats | goat-eats .

  strats solution eagerEating safe @ River .

  sd solution := goat ; alone ; cabbage ; goat ;
                 wolf ; alone ; goat .

  sd eagerEating := match left | G cabbage goat ? idle
      : ((eating or-else oneCrossing) ; eagerEating) .
```

```
  sd safe := match left | G ? idle
      : (oneCrossing ; not(eating) ; safe) .
endsm
```

In addition to the `oneCrossing` and `eating` strategies described before, there is also a deterministic strategy `solution` that simply applies a choice of steps that are known to solve the problem. The `eagerEating` strategy applies any rule in each recursive call respecting their precedence, and continues indefinitely or until a solution is found. Observe that the definition is recursive and nonterminating. This will not pose a problem since the execution engine and the model checker will be able to detect this loop and finish, and it is a useful resource to specify the behavior of reactive systems. The last strategy `safe` discards all rewriting paths where some being can be swallowed by concatenating the `not(eating)` strategy that fails whenever `eating` succeeds. Note that the stop condition only checks whether the left side of the river is empty, which is enough provided no one dies, while in `eagerEating` it is necessary to check that the goat and cabbage are still alive. We can execute the strategy to see how the solution is reached:

```
Maude> srew initial using safe .

Solution 1
rewrites: 33
result River: left | right shepherd wolf goat cabbage

No more solutions.
rewrites: 33
```

More details about the strategy language and examples can be found in its chapter in the Maude manual [9], in [12], and the strategy language website [14].

3 Model checking

Model checking [7,8] is an automated verification technique based on the exhaustive exploration of a system model to check a property describing aspects of its intended behavior. Multiple variants and algorithms exist, but traditionally the model is represented as a state and transition system, and the property in some temporal logic.

A *transition system* or *abstract reduction system* is a set of states S endowed with a binary transition relation $(\to) \subseteq S \times S$. It is usually required that every state has at least a successor to avoid dealing with finite executions. In the abstract context of an $\mathcal{A} = (S, \to)$, strategies can be seen as subsets E of the set of all execution paths $\Gamma^{\omega}_{\mathcal{A}} = \{(s_n)_{n=0}^{\infty} : s_n \to s_{n+1}\}$ of the system. In the following, we will write $\Gamma^{\omega}_{\mathcal{A},s}$ for the set of executions starting at $s \in S$ and $\Gamma^{*}_{\mathcal{A}}$ for the set of finite executions. This definition of strategy is sometimes called *abstract* or *extensional* [5] in contrast with an *intensional* characterization in terms of partial functions $\lambda : \Gamma^{*}_{\mathcal{A}} \to \mathcal{P}(S)$ that limit the possible next steps for a given execution prefix. Although these two definitions are not equivalent [5,23], most common strategies can be expressed intensionally.

The properties about the system are expressed in terms of some tags declared for each state. This yields a Kripke structure $\mathcal{K} = (S, \to, AP, I, \ell)$ with a finite set of such atomic propositions AP, a finite set of initial states $I \subseteq S$, and a labeling function $\ell : S \to \mathcal{P}(AP)$. Temporal logics combine these properties with operators that describe how they occur in time. Well-known examples of such logics are CTL* and its sublogics LTL (Linear Temporal Logic) and CTL (Computational Tree Logic).

However, some other logics like the μ-calculus do not only refer to state properties but also to the transitions. The abstract setting needs then to be enriched with labels for them: *labeled transition systems* (LTS) are defined as triples (S, A, R) where A is a set of edge labels or actions and $R \subseteq S \times A \times S$ is a tagged relation. Strategies and executions are defined similarly, but in this case interleaving states with edge labels, i.e., $\Gamma^{\omega}_{\mathcal{A},s_0} = \{s_0(a_n s_n)_{n=1}^{\infty} : s_n \to^{a_{n+1}} s_{n+1}\}$.

Maude supports on-the-fly LTL model checking since its 2.0 version [15]. The mapping of a rewriting system to the model-checking framework is natural: its states are its terms and its transitions are rule applications. All executions are assumed to be infinite, by repeating the last state of finite executions, adding a loop transition to deadlock states, like in Spin and other verification tools. In order to prepare a Maude module for model checking, users need to extend it including the predefined `SATISFACTION` module, declaring the state sort, and the atomic propositions as regular Maude operators of sort `Prop`, and defining them equationally for all terms using the satisfaction relation symbol `_|=_`. Here is an example for the river crossing puzzle:

```
mod RIVER-CROSSING-PREDS is
  protecting RIVER-CROSSING .
  including SATISFACTION .

  subsort River < State .

  ops goal death bad : → Prop [ctor] .

  var  R    : River .
  vars G G' : Group .

  eq left | G goat cabbage |= goal = true .
  eq R |= goal = false [owise] .

  eq cabbage G | G' goat |= death = false .
  eq cabbage goat G | G' |= death = false .
  eq R |= death = true [owise] .

  eq wolf goat G | G' shepherd |= bad = true .
  eq goat cabbage G | G' shepherd |= bad = true .
  eq R |= bad = false [owise] .
endm
```

Three properties are defined: `goal` that is only satisfied by the puzzle solution, `death` that tags states where someone has already been eaten, and `bad` that

signals states in which eating is possible but not yet accomplished. Finally, the user should import the predefined `MODEL-CHECKER` module giving access to a special operator `modelCheck` that reduces to the verification result, assuming some decidability requirements [15].

```
Maude> red modelCheck(initial, [] (bad → <> death)) .
rewrites: 44
result ModelCheckResult: counterexample(
   {right | left shepherd wolf goat cabbage,'alone}
    ...
   {left shepherd cabbage | right wolf goat,'cabbage},
   {left | right shepherd wolf goat cabbage,'alone}
   {left shepherd | right wolf goat cabbage,'alone})
```

In this case, the property is not satisfied and a counterexample execution is obtained, described by a cycle and a path to it.

Recently, we have extended the model checker to rewrite theories controlled by strategies [23]. From an abstract point of view, a system $\mathcal{K}$ controlled by a strategy $E \subseteq \Gamma^{\omega}_{\mathcal{K}}$ is said to satisfy a linear property φ if $\mathcal{K}, \pi \vDash \varphi$ for all $\pi \in E$. This definition is natural and almost unavoidable, since linear-time properties refer to individual executions quantified universally. The fundamental issue is which are the executions E allowed by a Maude strategy language expression α.

This question has been answered by defining a nondeterministic structural operational semantics for the strategy language. Its execution states $q \in \mathcal{XS}$ are terms augmented with a continuation for the strategy execution, and its steps $q \twoheadrightarrow q'$ correspond to single rule rewrites $\mathrm{cterm}(q) \to^{1}_{R} \mathrm{cterm}(q')$ on the underlying terms, denoted by $\mathrm{cterm}(q)$. States are usually of the form $t \,@\, s$ where s is a stack of strategy expressions whose execution is pending and substitutions defining the variable contexts of the active strategy calls, but more complex constructs are required for operators involving subsearches. Projecting the term part of the semantics executions leads to well-defined abstract strategies for the underlying system,

$$E(\alpha, t) = \{(\mathrm{cterm}(q_n))_{n=0}^{\infty} : q_0 = t \,@\, \alpha,\ q_n \twoheadrightarrow q_{n+1}\}.$$

Moreover, the abstract definition of model checking for this $E(\alpha, t)$ is equivalent to model checking the Kripke structure given by the semantics graph

$$\mathcal{B} := (\mathcal{XS}, \twoheadrightarrow, \{t \,@\, \alpha\}, AP, \ell \circ \mathrm{cterm})$$

under some decidability assumptions [23].

The strategy-aware model checker shares a great part of its infrastructure with the strategy execution engine and the original model checker. Their usage is similar, but in this case the `STRATEGY-MODEL-CHECKER` module, whose `modelCheck` symbol receives an additional argument to indicate the name of the strategy that controls the system, should be imported instead.

```
Maude> red modelCheck(initial, [] ~ bad, 'safe) .
rewrites: 54
```

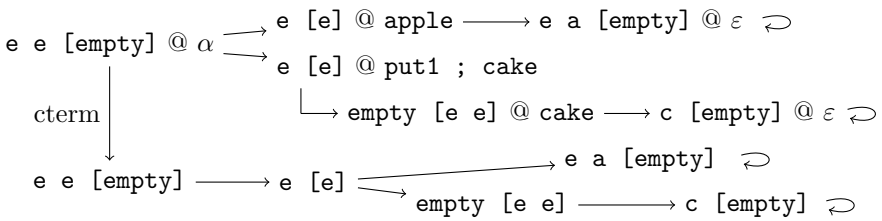
```
result Bool: true

Maude> red modelCheck(initial, [] (bad → 0 death),
                      'eagerEating) .
rewrites: 121
result Bool: true
```

A version of Maude including this model checker, its source code, and detailed documentation can be downloaded from [14].

4 Model checking branching-time properties

While the abstract definition for checking linear-time properties using strategies was very simple, the case of branching-time properties is not so clear. The main difficulty can be observed in the following example of a vending machine, which admits one-euro coins `e` and sells apples `a` and cakes `c` for one and two euros respectively. According to the semantics, the execution tree of the strategy $\alpha \equiv$ `(put1 ; apple) | (put1 ; put1 ; cake)` from the term `e e [empty]` is:



We can see that the tree structure is not preserved by the projection. Its effects are observed in the satisfaction of the CTL property $\mathbf{A}\bigcirc\mathbf{E}\Diamond \mathit{hasApple}$ where *hasApple* is only satisfied if an apple has been bought. Moreover, expressions denoting the same abstract strategy, like `put1 ; (apple | put1 ; cake)` and the previous one, would satisfy different properties. Fortunately, the problem can be solved by simply merging successor states whose terms coincide, like `e [e] @ apple` and `e [e] @ put1 ; cake` in the example above.

In abstract terms, we suggested in [23] that the satisfaction of a branching-time property φ on a system $\mathcal{A}$ controlled by a strategy can be understood as the satisfaction of φ in its *unwinding*, the transition system whose states are the finite executions $\Gamma^*_{\mathcal{A},s}$ of the model and whose transitions are those allowed by the (intensional) strategy. However, since this construct is not finite, the practical usage of this definition goes through finding a bisimilar finite transition system. For the Maude strategy language, this can be the one derived from its nondeterministic semantics ($\mathcal{B}$ in Section 3) after merging states. We define it formally as $\mathcal{M} := (\mathcal{P}(\mathcal{XS}), [\twoheadrightarrow], \{\{t_0 \mathbin{@} \alpha\}\}, AP, \ell \circ \mathrm{cterm})$ where

$$Q[\twoheadrightarrow]Q' \iff \exists\, t \in T_\Sigma \quad Q' = \{q' : q \twoheadrightarrow q', \mathrm{cterm}(q') = t, q \in Q\}.$$

When considering the labeled transition system, a slightly different Kripke structure $\mathcal{M}'$ should be defined, in which only the successors for a given rule label are

included in each Q' state. The following proposition states that $\mathcal{M}$ is bisimilar to the strategy expansion:

Proposition 1. *Given a strategy expression α and a term t_0, the initial state $\{t_0 @ \alpha\}$ of $\mathcal{M}$ is bisimilar to the state t_0 in $\mathcal{U} = (T^*_\Sigma, U)$, where $(wt)\,U\,(wtt') \iff \exists w' \in S^\omega\ wtt'w' \in E(\alpha, s)$.*

Hence, to model check state-based properties of system controlled by strategies, we propose applying standard algorithms on $\mathcal{M}$. A similar proposition holds for the labeled variant $\mathcal{M}'$ and $\mathcal{U}' = ((S \cup A)^*, U')$ where $(wt)\,U'\,(wtat')$ iff $\exists w' \in (S \cup A)^\omega\ wtat'w' \in E_{\text{labeled}}(\alpha, s)$. For action-based or doubly-labeled logics, we propose using $\mathcal{M}'$ instead. In the following sections, to justify that the proposed procedure is meaningful, we give reasonable generalizations of two specific branching-time logics for strategy-controlled systems, and show how their notions of satisfaction coincide. CTL* and μ-calculus are chosen because they are well-known and because the tool used only handles those, but the procedure is general and can be applied to other logics.

4.1 CTL*

The proposed generalized definition is similar to those we can find in most reference textbooks [7,8] and coincides with a previous definition for trees [27]. We identify abstract strategies with trees since they are in univocal relation as long as trees only branch to distinct children, as it is the case. For an execution $\pi = (\pi_n)_{n=0}^\infty$, we denote the suffix that starts at the position k by $\pi^k = (\pi_{k+n})_{n=0}^\infty$, the prefix that stops at k by $\pi^{-k} = \pi_0 \cdots \pi_k$, and all the executions of a given abstract strategy E continuing a given prefix by $E \restriction ws = \{s\pi : ws\pi \in E\}$ for all $w \in S^*$ and $s \in S$.

1. $E \vDash p$ iff $\forall\, \pi \in E \quad p \in \ell(\pi_0)$
2. $E \vDash \neg\Phi$ iff $E \nvDash \Phi$
3. $E \vDash \Phi_1 \wedge \Phi_2$ iff $E \vDash \Phi_1$ and $E \vDash \Phi_2$
4. $E \vDash \mathbf{A}\,\phi$ iff $\forall\, \pi \in E \quad E \restriction \pi_0, \pi \vDash \phi$
5. $E \vDash \mathbf{E}\,\phi$ iff $\exists\, \pi \in E \quad E \restriction \pi_0, \pi \vDash \phi$
6. $E, \pi \vDash \Phi$ iff $E \vDash \Phi$
7. $E, \pi \vDash \neg\varphi$ iff $E, \pi \nvDash \varphi$
8. $E, \pi \vDash \varphi_1 \wedge \varphi_2$ iff $E, \pi \vDash \varphi_1$ and $E, \pi \vDash \varphi_2$
9. $E, \pi \vDash \bigcirc \varphi$ iff $E \restriction \pi_0\pi_1, \pi^1 \vDash \varphi$
10. $E, \pi \vDash \Diamond \varphi$ iff $\exists\, n \geq 0 \quad E \restriction \pi^{-n}, \pi^n \vDash \varphi$
11. $E, \pi \vDash \Box \varphi$ iff $\forall\, n \geq 0 \quad E \restriction \pi^{-n}, \pi^n \vDash \varphi$
12. $E, \pi \vDash \varphi_1\, \mathbf{U}\, \varphi_2$ iff $\exists\, n \geq 0\ E \restriction \pi^{-n}, \pi^n \vDash \varphi_2 \wedge \forall\, 0 \leq k < n\ E \restriction \pi^{-k}, \pi^k \vDash \varphi_1$

Observe that it only differs from the classical definition in the fact that the strategy is carried on. Path formulae φ are understood similarly, but here, a state property Φ does not only depend on the state but on the full state history. The extended and classical relations are linked by the essential property that $\mathcal{K}, s \vDash \varphi \iff \Gamma^\omega_{\mathcal{K},s} \vDash \varphi$ for every formula φ. The following proposition justifies that model checking can be solved by the classical procedures applied on $\mathcal{M}$:

Proposition 2. $E(\alpha, t) \vDash \varphi \iff \mathcal{M}, \{t @ \alpha\} \vDash \varphi$

4.2 μ-calculus

We present a generalized definition of μ-calculus for strategies that mimics the original one. While in the original μ-calculus a valid formula φ is given meaning $[\![\varphi]\!]_\eta$ as the set of states in which it is satisfied, here, a formula will denote instead a set $\langle\!\langle\varphi\rangle\!\rangle_\eta$ of subtrees (in other words, strategies) in which φ is satisfied. Let η be an assignment from variables Z in the formula to subsets of $\mathcal{P}(\Gamma^\omega_\mathcal{K})$:

1. $\langle\!\langle p\rangle\!\rangle_\eta = \{T \subseteq \Gamma^\omega_\mathcal{K} : \forall sa\pi \in T \quad p \in \ell(s)\}$
2. $\langle\!\langle \neg\varphi\rangle\!\rangle_\eta = \mathcal{P}(\Gamma^\omega_\mathcal{K}) \setminus \langle\!\langle\varphi\rangle\!\rangle_\eta$
3. $\langle\!\langle \varphi_1 \wedge \varphi_2\rangle\!\rangle_\eta = \langle\!\langle\varphi_1\rangle\!\rangle_\eta \cap \langle\!\langle\varphi_2\rangle\!\rangle_\eta$
4. $\langle\!\langle Z\rangle\!\rangle_\eta = \eta(Z)$
5. $\langle\!\langle \langle a\rangle\varphi\rangle\!\rangle_\eta = \{T \subseteq \Gamma^\omega_\mathcal{A} : \exists\, sa\pi \in T \quad T \restriction sa\pi_0 \in \langle\!\langle\varphi\rangle\!\rangle_\eta\}$
6. $\langle\!\langle \nu Z.\varphi\rangle\!\rangle_\eta = \bigcup \{F \subseteq \mathcal{P}(\Gamma^\omega_\mathcal{A}) : F \subseteq \langle\!\langle\varphi\rangle\!\rangle_{\eta[Z/F]}\}$

Other constructors like $[a]\varphi$ and $\mu Z.\varphi$ are defined by their usual equivalences to these. Provided that every variable is under an even number of negations, the definition is monotone and the fixpoints are well-defined. When the formula φ is ground, i.e. it does not have free variables, we omit the valuation subscript η. This generalization is connected with the original definition by the property that $\langle\!\langle\varphi\rangle\!\rangle \ni \Gamma^\omega_{\mathcal{K},s} \iff s \in [\![\varphi]\!]$ for any state $s \in S$ and ground formula φ.

As for CTL*, the following proposition claims that a formula is satisfied for a strategy in the generalized sense iff it is satisfied in the merged labeled transition system generated by the nondeterministic semantics:

Proposition 3. *For any ground formula φ, $E(\alpha, t) \in \langle\!\langle\varphi\rangle\!\rangle$ in $\mathcal{K}$ iff $t \in [\![\varphi]\!]$ in the transition system $\mathcal{M}'$.*

5 The Maude language module for LTSmin

According to the previous section, to check CTL* or μ-calculus properties on Maude specifications we should take the Kripke structure $\mathcal{B}$, already generated for the LTL model checker, merge its states as in $\mathcal{M}$ or $\mathcal{M}'$ and apply the standard algorithms on them. To avoid programming these algorithms for scratch, we have developed instead a language module for the language-independent model checker LTSmin [16]. Oversimplifying, this software allows defining *language frontends* that expose programs in a specification language like Maude as a labeled transition system to some builtin *algorithmic backends*, including model checkers for different logics. A Kripke-like C interface called PINS (Partitioned Next State Interface) is used, which promotes sharing additional information about the internal structure of the models to speed up algorithms. Frontends are included for various modeling formalisms like Promela, PNML, DIVINE, UPPAAL, etc., and custom language modules, like ours, can also be loaded by the LTSmin tools using the POSIX's `dlopen` API.

The language module is the C library `libmaudemc.so` illustrated in Fig. 1. On the one hand, the module is linked with the C++ implementation of Maude 3.0

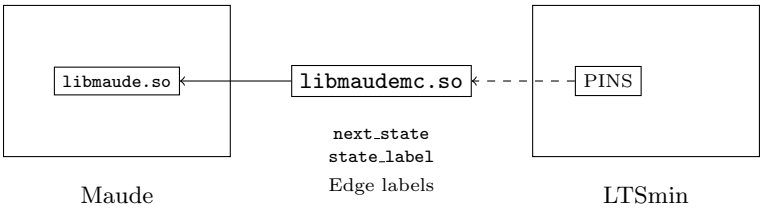


Fig. 1. Architecture of the Maude LTSmin plugin

including the extended LTL model checker for strategy-controlled systems, which processes the Maude files and gives access to the transition system used for LTL model checking. On the other hand, the plugin implements the PINS interface by exporting some C functions that the LTSmin model-checking algorithms will call to introspect the model[2]: the `next_state` function provides the successors of a given state, including their edge labels, and `state_label` tests whether an atomic proposition holds in a state. The module itself takes care of merging states as required for CTL* and μ-calculus, and also removes states in which the strategy has failed, which were ignored automatically by the nested depth-first search of LTL model checking, but must be explicitly purged here. LTSmin lets frontend designers represent states as vectors of integers, which can be partitioned and whose dependencies can be declared as matrices that the algorithms may use to improve their efficiency and allow distributed implementations. However, in our case the state is a single integer that represents an internal state of the Maude model checker, since partitioning and inferring relations about arbitrary Maude specifications seems unpractical.

LTSmin includes different commands like `pins2lts-seq` for explicit-state LTL model checking or `pins2lts-sym` for symbolic CTL/CTL*/μ-calculus that, once our module is loaded with `--loader=libmaudemc.so`, are ready to handle Maude specifications. The Maude source file, the initial term, and an optional strategy expression have to be provided as arguments to the command, along with the temporal formula to be checked using LTSmin's syntax. In order to simplify its usage, a helper utility `umaudemc` has been written to call the appropriate tool and configuration for the appropriate formula among those supported by LTSmin. It also allows expressing the temporal formulae using Maude's LTL syntax extended with the quantifiers `A_` and `E_` for CTL*, and the operators `<_>_`, `[_]_`, `mu_._` and `nu_._` for μ-calculus.

For example, after downloading the plugin from `http://maude.ucm.es/strategies` and LTSmin from `https://ltsmin.utwente.nl`, we can check the CTL property that every state of the river crossing puzzle can be continued to a solution $\mathbf{A}\,\Box\,\mathbf{E}\,\Diamond\,\mathit{goal}$. This formula is satisfied when the system is controlled by the `safe` strategy, but not when using the `eagerEating` strategy or when the system runs uncontrolled.

[2] LTSmin loads `libmaudemc.so` using the `dlopen` API, which allows loading dynamic libraries at runtime, accessing their symbols, and calling their C functions.

```
$ umaudemc check river.maude initial 'A [] E <> goal' safe
maude-mc: selected module is RIVER-CROSSING-SCHECK
pins2lts-sym: Formula A [] E <> goal == true holds
              for the initial state
maude-mc: 16 system states explored, 264 rewrites

$ umaudemc check river initial 'A [] E <> goal' eagerEating
pins2lts-sym: Formula ... does not hold ...
maude-mc: 43 system states explored, 4012 rewrites

$ umaudemc check river.maude initial 'A [] E <> goal'
pins2lts-sym: Formula ... does not hold ...
maude-mc: 36 system states explored, 3058 rewrites
```

However, the property $\mathbf{A}\,\Box\,(bad \vee death \vee \mathbf{E}\,\Diamond\,goal)$ holds under the `eagerEating` strategy.

```
$ umaudemc check river.maude initial \
    'A [] (bad \/ death \/ E <> goal)' eagerEating
pins2lts-sym: Formula ... holds ...
maude-mc: 43 system states explored, 1088 rewrites
```

LTL properties can be checked both directly in Maude or using the LTSmin plugin. Against the model-checking examples available in our web page [14], LTSmin is 10,73% slower in average (or 11,21% using its builtin caching) and requires more memory. However, the communication costs and the partially redundant representation of the state can explain this difference. Moreover, since the PINS interface asks for all the successors of a state at once, the on-the-fly state space expansion is lazier in Maude and the order in which children are processed is reversed. The size of the property automata generated from the formulae by both tools coincide, except in one case when Maude's is one state smaller.

Other alternatives to bring CTL* and μ-calculus model checking to Maude have been considered like generating an equivalent model for a specific tool or exporting it to a somehow standard representation. For example, the Model Checking Contest uses the Petri Net Markup Language (PNML) to state the problems for all the competitor tools. In fact, we first wrote a metalevel prototype that outputs a model for the NuSMV model checker. Finally, we decided to use LTSmin because its interface is closer to our description of the transition system, and because of its live connection that allows generating the space state and checking propositions on the fly. Only LTL model checking, which was already covered, can benefit from the first advantage, but the second is always useful.

6 Related work

Each section includes references to related work for its topic, but we should also mention that other model checkers have been developed for Maude. A timed CTL model checker is included as part of Real-Time Maude [18], and the builtin

one was also extended to support the more expressive Linear Temporal Logic of Rewriting [3] (LTLR), as well as models based on narrowing instead of rewriting in the abstract logical model checker [2].

The combination of strategies and model checking is not original. In the field of multiplayer games, various logics like ATL* [1] and *strategy logic* [21] have been proposed to reason about player strategies. Other logics like mCTL* [17] are extended to take past actions into account. However, our approach is different, since strategies are part of the specification of the model, keeping the property specification unaltered.

7 Conclusions

In this paper, the study of model checking for systems controlled by strategies is extended to branching-time properties, and a tool is presented that allows CTL* and μ-calculus properties to be checked on standard Maude specifications and strategy-controlled ones. In a wider sense, this work aims to make strategies a more useful and convenient choice to specify and verify systems. While strategy-free models can be fully explored at the metalevel using the `metaXApply` function, there were no resources in the current metalevel to follow step by step the execution of a strategy, without implementing them from scratch. Our plugin exposes these Maude models to external tools for verification, visualization and other types of analysis.

References

1. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. Journal of the ACM **49**(5), 672–713 (2002). https://doi.org/10.1145/585265.585270
2. Bae, K., Escobar, S., Meseguer, J.: Abstract logical model checking of infinite-state systems using narrowing. In: van Raamsdonk, F. (ed.) 24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands. LIPIcs, vol. 21, pp. 81–96. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013). https://doi.org/10.4230/LIPIcs.RTA.2013.81
3. Bae, K., Meseguer, J.: Model checking linear temporal logic of rewriting formulas under localized fairness. Science Computer Programming **99**, 193–234 (2015). https://doi.org/10.1016/j.scico.2014.02.006
4. Borovanský, P., Kirchner, C., Kirchner, H., Ringeissen, C.: Rewriting with strategies in ELAN: A functional semantics. Int. J. Found. Comput. Sci. **12**(1), 69–95 (2001). https://doi.org/10.1142/S0129054101000412
5. Bourdier, T., Cirstea, H., Dougherty, D.J., Kirchner, H.: Extensional and intensional strategies. In: Fernández, M. (ed.) Proceedings Ninth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2009, Brasilia, Brazil, 28th June 2009. EPTCS, vol. 15, pp. 1–19 (2009). https://doi.org/10.4204/EPTCS.15.1
6. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Programming **72**(1-2), 52–70 (2008). https://doi.org/10.1016/j.scico.2007.11.003

7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The Mit Press (1999)
8. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8
9. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.: Maude Manual v3.0 (2019-12), http://maude.lcc.uma.es/manual271/maude-manual.html
10. Clavel, M., Meseguer, J.: Reflection and strategies in rewriting logic. In: Meseguer, J. (ed.) Proceedings of the First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3-6, 1996. Electronic Notes in Theoretical Computer Science, vol. 4, pp. 126–148. Elsevier (1996). https://doi.org/10.1016/S1571-0661(04)00037-4
11. Clavel, M., Meseguer, J.: Internal strategies in a reflective logic. In: Gramlich, B., Kirchner, H. (eds.) Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction. pp. 1–12 (1997)
12. 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 Computer Programming **110** (2020). https://doi.org/10.1016/j.jlamp.2019.100497
13. Eker, S., Martí-Oliet, N., Meseguer, J., Verdejo, A.: Deduction, strategies, and rewriting. In: Archer, M., de la Tour, T.B., Muñoz, C. (eds.) Proceedings of the 6th International Workshop on Strategies in Automated Deduction, STRATEGIES 2006, Seattle, WA, USA, August 16, 2006. Electronic Notes in Theoretical Computer Science, vol. 174(11), pp. 3–25. Elsevier (2007). https://doi.org/10.1016/j.entcs.2006.03.017
14. Eker, S., Martí-Oliet, N., Meseguer, J., Pita, I., Rubio, R., Verdejo, A.: Strategy language for Maude, http://maude.ucm.es/strategies
15. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: Gadducci, F., Montanari, U. (eds.) Proceedings of the Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19-21, 2002. Electronic Notes in Theoretical Computer Science, vol. 71, pp. 162–187. Elsevier (2004). https://doi.org/10.1016/S1571-0661(05)82534-4
16. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: Ltsmin: High-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9035, pp. 692–707. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_61
17. Kupferman, O., Vardi, M.Y.: Memoryful branching-time logic. In: Alur, R. (ed.) 21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings. pp. 265–274. IEEE Computer Society (2006). https://doi.org/10.1109/LICS.2006.34
18. Lepri, D., Ábrahám, E., Ölveczky, P.C.: Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories. Sci. Comput. Program. **99**, 128–192 (2015). https://doi.org/10.1016/j.scico.2014.06.006
19. Martí-Oliet, N., Meseguer, J., Verdejo, A.: Towards a strategy language for Maude. In: Martí-Oliet, N. (ed.) Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27-April 4, 2004. Electronic Notes in Theoretical Computer Science, vol. 117, pp. 417–441. Elsevier (2004). https://doi.org/10.1016/j.entcs.2004.06.020

Strategies, model checking and branching-time properties in Maude*

Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, Alberto Verdejo

Universidad Complutense de Madrid, Spain
{rubenrub,narciso,ipandreu,jalberto}@ucm.es

Abstract. Maude 3.0 includes as a new feature an object-level strategy language. Rewriting strategies can now be used to easily control how rules are applied, restricting the rewriting systems behavior. This new specification layer would not be useful if there were no tools to execute, analyze and verify its creatures. For that reason, we extended the Maude LTL model checker to systems controlled by strategies, after studying their model-checking problem. Now, we widen the range of properties that can be checked in Maude models, both strategy-aware and strategy-free, by implementing a module for the language-independent model checker LTSmin that supports logics like CTL* and μ-calculus.

1 Introduction

The Maude [9] specification language has recently reached its 3.0 version, integrating new features developed during the last years, including a full implementation of the Maude strategy language [9, §10]. Although rewriting logic owes its natural representation of concurrency to the possibility that different rules can be executed in different positions at each step of the rewriting process, there are situations in which it is convenient to control such nondeterminism. This is the purpose of strategies, which have traditionally been expressed in Maude at the metalevel by means of its reflective features [10,11,25], but since the complexity and learning curve of programming metalevel computations is hard, an object-level strategy language design was proposed [19,13], exercised with different examples [28,24,20,26, ...], and finally added to the Core Maude functionality. Strategies can be described compositionally using strategy modules on top of system modules, and different commands are provided to rewrite a term following a strategy.

However, this new feature would be worthless without convenient tools to analyze the specifications using it. One of the most useful tools for verifying regular Maude modules is its LTL model checker [15]. In a previous work [23], we have studied the model-checking problem for rewriting systems controlled by strategies and presented an extension of the model checker to deal with them.

* Research partially supported by MCIU Spanish project *TRACES* (TIN2015-67522-C3-3-R). Rubén Rubio is partially supported by MCIU grant FPU17/02319.

Verification of the IBOS Browser Security Properties in Reachability Logic

Stephen Skeirik[1], José Meseguer[1], and Camilo Rocha[2]

[1] University of Illinois at Urbana-Champaign
skeirik2@illinois.edu
meseguer@illinois.edu
[2] Pontificia Universidad Javeriana Cali
camilo.rocha@javerianacali.edu.co

Abstract. This paper presents a rewriting logic specification of the Illinois Browser Operating System (IBOS) and defines several security properties, including the *same-origin policy* (SOP) in reachability logic. It shows how these properties can be deductively verified using our constructor-based reachability logic theorem prover. This paper also highlights the reasoning techniques used in the proof and three modularity principles that have been crucial to scale up and complete the verification effort.

1 Introduction

Rationale and Origins. Web browsers have in fact become operating systems for a myriad of web-based applications. Given the enormous user base and the massive increase in web-based application areas, browsers have for a long time been a prime target for security attacks, with a seemingly unending sequence of browser security violations. One key reason for this problematic state of affairs is the enormous size (millions of lines of code) and sheer complexity of conventional browsers, which make their formal verification a daunting task. An early effort to substantially improve browser security by formal methods was jointly carried out by researchers at Microsoft Research and the University of Illinois at Urbana-Champaign (UIUC), who formally specified Internet Explorer (IE) in Maude [11], and model checked that formalization finding 13 new types of unknown address bar or status bar spoofing attacks in it [8]. To avoid attacks on those newly found vulnerabilities, they were all corrected in IE *before* [8] was published. But the research in [8] just uncovered *some* kinds of possible attacks, and the sheer size and complexity of IE made full verification unfeasible. This stimulated a team of systems and formal methods researchers at UIUC to ask the following question: *could formal methods be used from the very beginning in the design of a secure browser with a very small trusted code base (TCB) whose design could be verified?* The answer given to this question was the Maude-based design, model checking verification, and implementation of the IBOS Browser cum operating system, [37, 38, 48, 49], with a 42K line trusted code base (TCB), several orders of magnitude smaller than the TCBs of commodity browsers.

Why this Work. As further explained in Section 6, only a model checking verification of the IBOS security properties relying on a hand-proof abstraction argument for

its full applicability was possible at the time IBOS was developed [34, 37, 38]. A subsequent attempt at a full deductive verification of IBOS in [34] had to be abandoned due to the generation of thousands of proof obligations. In retrospect, this is not surprising for two reasons. (1) Many of the symbolic techniques needed to scale up the IBOS deductive verification effort, including variant unification and narrowing [14], order-sorted congruence closure module axioms [30], and variant-based satisfiability [32, 41], did not exist at the time. In the meantime, those symbolic techniques have been developed and implemented in Maude. (2) Also missing was a *program logic* generalizing Hoare logic for Maude specifications in which properties of concurrent systems specified in Maude could be specified and verified. This has been recently addressed with the development of a *constructor-based reachability logic* for rewrite theories in [43, 44], which extends prior reachability logic research on verification of conventional programs using K in [35, 36, 46, 47]. In fact, what has made possible the deductive proof of the IBOS security properties presented in this paper is precisely the combination of the strengths from (1) and (2) within the reachability logic theorem prover that we have developed for carrying out such a proof. Implicit in both (1) and (2) are two important proof obligations. First, both our symbolic reasoning and reachability logic engines take as input a rewrite theory $\mathcal{R}$. However, the correctness of the associated deductions depends on the theory being *suitable* for symbolic reachability analysis, i.e., its equations should be ground convergent and sufficiently complete; therefore, these properties are proof obligations that must be discharged. Second, the previous model-checking-based verification that the IBOS design satisfies certain security properties [37, 38] was based on an invariant I_0. Our deductive verification uses a slightly different invariant I that is also *inductive* (as explained in Section 3). Thus, we require that I is *at least* as strong as or stronger than I_0 to ensure that our specification of the IBOS security properties does not miss any cases covered by the prior work. Both of these important proof obligations have been fully checked as explained in [40]. Last, but not least, as we further explain in Section 6, the IBOS browser security goals remain as relevant and promising today as when IBOS was first developed, and this work bring us closer to achieving those goals.

Main Contributions. They include:

- The first full *deductive verification of the IBOS browser* as explained above.
- A general *modular proof methodology* for scaling up reachability logic proofs of object-based distributed systems that has been invaluable for verifying IBOS, but has a much wider applicability to general distributed system verification.
- A substantial and useful *case study* that can be of help to other researchers interested in both browser verification and distributed system verification.
- New capabilities of the *reachability logic prover*, which in the course of this research has evolved from the original prototype reported in [43] to a first prover version to be released in the near future.

Plan of the Paper. Preliminaries are gathered in Section 2. Reachability Logic and invariant verification are presented in Section 3. IBOS, its rewriting logic Maude specification, and the specification of its security properties are explained in Section 4. The deductive proof of those IBOS properties and the modular proof methodology used are described in Section 5. Section 6 discusses related work and concludes the paper.

2 Preliminaries on Equational and Rewriting Logic

We present some preliminaries on order-sorted equational logic and rewriting logic. The material is adapted from [15, 28, 29].

Order-Sorted Equational Logic. We assume the basic notions of order-sorted (abbreviated OS) signature Σ, Σ-term t, Σ-algebra A, and Σ-homomorphism $f : A \to B$ [15, 28]. Intuitively, Σ defines a partially ordered set of sorts $(S, \leqslant)$, which are interpreted in a Σ-algebra A with carrier family of sets $A = \{A_s\}_{s \in S}$ as sort containments. For example, if we have a sort inclusion $Nat < Int$, then we must have $A_{Nat} \subseteq A_{Int}$. An operator, say $+$, in Σ may have several related typings, e.g., $+ : Nat\ Nat \to Nat$ and $+ : Int\ Int \to Int$, whose interpretations in an algebra A must agree when restricted to subsorts. The OS algebras over signature Σ and their homomorphisms form a category $\mathbf{OSAlg}_\Sigma$. Furthermore, under mild syntactic conditions on Σ, the term algebra T_Σ is initial [28]; all signatures are assumed to satisfy these conditions.

An S-sorted set $X = \{X_s\}_{s \in S}$ of *variables*, satisfies $s \neq s' \Rightarrow X_s \cap X_{s'} = \emptyset$, and the variables in X are always assumed disjoint from all constants in Σ. The *Σ-term algebra* on variables X, $T_\Sigma(X)$, is the *initial algebra* for the signature $\Sigma(X)$ obtained by adding to Σ the variables X *as extra constants*. Given a Σ-algebra A, an *assignment* a is an S-sorted function $a \in [X \to A]$ mapping each variable $x \in X_s$ to a value $a(x) \in A_s$ for each $s \in S$. Each such assignment uniquely extends to a Σ-homomorphism $\_a : T_\Sigma(X) \to A$, so that if $x \in X_s$, then $x\,a = a(x)$. In particular, for $A = T_\Sigma(X)$, an assignment $\sigma \in [X \to T_\Sigma(X)]$ is called a *substitution* and uniquely extends to a Σ-homomorphism $\_\sigma : T_\Sigma(X) \to T_\Sigma(X)$. Define $dom(\sigma) = \{x \in X \mid x \neq x\sigma\}$ and $ran(\sigma) = \bigcup_{x \in dom(\sigma)} vars(x\sigma)$.

We assume familiarity with the language of first-order logic with equality. In particular, given a Σ-formula φ, we assume familiarity with the satisfaction relation $A, a \models \varphi$ for a Σ-algebra A and assignment $a \in [fvars(\varphi) \to A]$ for the *free variables* $fvars(\varphi)$ of φ. Then, φ is *valid* in A, denoted $A \models \varphi$, iff $\forall\ a \in [fvars(\varphi) \to A]\ A, a \models \varphi$, and is *satisfiable* in A iff $\exists\ a \in [fvars(\varphi) \to A]\ A, a \models \varphi$. Let $Form(\Sigma)$ (resp. $QFForm(\Sigma)$) denote the set of Σ-formulas (resp. quantifier free Σ-formulas).

An OS *equational theory* is a pair $T = (\Sigma, E)$, with E a set of (possibly conditional) Σ-equations. $\mathbf{OSAlg}_{(\Sigma,E)}$ denotes the full subcategory of $\mathbf{OSAlg}_\Sigma$ with objects those $A \in \mathbf{OSAlg}_\Sigma$ such that $A \models E$, called the *(Σ, E)-algebras*. The inference system in [28] is *sound and complete* for OS equational deduction. *E-equality*, i.e., provability $E \vdash u = v$, is written $u =_E v$. $\mathbf{OSAlg}_{(\Sigma,E)}$ has an *initial algebra* $T_{\Sigma/E}$ [28]. Given a system of Σ equations $\phi = u_1 = v_1 \wedge \ldots \wedge u_n = v_n$, an *$E$-unifier* for ϕ is a substitution σ such that $u_i\sigma =_E v_i\sigma$, $1 \leqslant i \leqslant n$; an *$E$-unification algorithm* for (Σ, E) generates a *complete set* of E-unifiers $Unif_E(\phi)$ for any system ϕ in the sense that, up to E-equality, any E-unifier σ of ϕ is a substitution instance of some unifier $\theta \in Unif_E(\phi)$.

Rewriting Logic. A *rewrite theory* $\mathcal{R} = (\Sigma, E \cup B, R)$, with $(\Sigma, E \cup B)$ an OS-equational theory with equations E and structural axioms B (typically any combination of associativity, commutativity, and identity), and R a collection of rewrite rules, specifies a *concurrent system* whose states are elements of the initial algebra $T_{\Sigma/E \cup B}$ and whose *concurrent transitions* are specified by the rewrite rules R. The concurrent system thus specified is the *initial reachability model* $\mathcal{T}_\mathcal{R}$ associated to $\mathcal{R}$ [6, 29].

Maude [11] is a declarative programming language whose programs are exactly rewrite theories. To be executable in Maude, a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$ should satisfy some *executability conditions* spelled out below. Recall the notation for term positions, subterms, and replacement from [12]: (i) positions in a term are marked by strings $p \in \mathbb{N}^*$ specifying a path from the root, (ii) $t|_p$ denotes the subterm of term t at position p, and (iii) $t[u]_p$ denotes the result of *replacing* subterm $t|_p$ at position p by u.

Definition 1. *An* executable rewrite theory *is a 3-tuple $\mathcal{R} = (\Sigma, E \cup B, R)$ with $(\Sigma, E \cup B)$ an OS equational theory with E possibly conditional and R a set of possibly conditional Σ-*rewrite rules*, i.e., sequents $l \to r$ if ϕ, with $l, r \in T_\Sigma(X)_s$ for some $s \in S$, and ϕ a quantifier-free Σ-formula. We further assume that:*

1. *B is a collection of associativity and/or commutativity and/or identity axioms and Σ is B-*preregular *[11].*
2. *Equations E, oriented as rewrite rules $\vec{E}$, are* convergent *modulo B [26].*
3. *Rules R are* ground coherent *with the equations E modulo B [13].*

The one-step R,B-rewrite relation $t \to_{R,B} t'$ *holds iff there is a rule $l \to r$ if $\phi \in R$, a ground substitution $\sigma \in [Y \to T_\Sigma]$ with Y the rule's variables, and position p where $t|_p =_B l\sigma$, $t' = t[r\sigma]_p$, and $T_{\Sigma/E\cup B} \models \phi\sigma$. Let $\to^*_{R,B}$ denote the reflexive-transitive closure of the rewrite relation $\to_{R,B}$.*

Intuitively, conditions (1)–(2) ensure that the initial algebra $T_{\Sigma/E\cup B}$ is isomorphic to the *canonical term algebra* $C_{\Sigma/E,B}$, whose elements are B-equivalence classes of $\vec{E},B$-*canonical* ground Σ-terms, where v is the $\vec{E},B$-*canonical form* of a term t, denoted $u = t!_{\vec{E},B}$, iff: (i) $t \to^*_{\vec{E},B} u$, and (ii) $(\nexists v \in T_\Sigma)\ u \to_{\vec{E},B} v$. By $\vec{E}$ convergent modulo B, $t!_{\vec{E},B}$ is unique up to B-equality [26]. Adding (3) ensures that "computing $\vec{E},B$-canonical forms before performing R,B-rewriting" is a *complete* strategy for rewriting with the rules R module equations E. That is, if $t \to_{R,B} t'$ and $t!_{\vec{E},B} = u$, then there exists a u' such that $u \to_{R,B} u'$ and $t'!_{\vec{E},B} =_B u'!_{\vec{E},B}$. We refer to [13, 26, 29] for more details.

Conditions (1)–(3) allow a simple and intuitive description of the *initial reachability model* $\mathcal{T}_\mathcal{R}$ [6] of $\mathcal{R}$ as the *canonical reachability model* $C_\mathcal{R}$ whose states are the elements of the *canonical term algebra* $C_{\Sigma/E,B}$, and where the one-step transition relation $[u] \to_\mathcal{R} [v]$ holds iff $u \to_{R,B} u'$ and $[u'!_{\vec{E},B}] = [v]$. Finally, if $u \to_{R,B} u'$ via rule $(l \to r \text{ if } \phi) \in R$ and a ground substitution $\sigma \in [Y \to T_\Sigma]$, then checking if condition $T_{\Sigma/E\cup B} \models \phi\sigma$ holds is *decidable* by reducing terms in $\phi\sigma$ to $\vec{E},B$-canonical form.

An OS-subsignature $\Omega \subseteq \Sigma$ is called a *constructor subsignature* for an OS equational theory $(\Sigma, E \cup B)$ where $\vec{E}$ is convergent modulo B iff $\forall t \in T_\Sigma\ t!_{\vec{E},B} \in T_\Omega$. Furthermore, the constructors Ω are then called *free* modulo axioms $B_\Omega \subseteq B$ iff, as S-sorted sets, $C_{\Sigma/E,B} = T_{\Omega/B_\Omega}$. This assumption gives a particularly simple description of the states of the canonical reachability model $C_\mathcal{R}$ as B_Ω-equivalence classes of ground Ω-terms. As explained in Section 3, this simple *constructor-based* description is systematically exploited in reachability logic.

An executable rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$ with constructor subsignature Ω is called *topmost* iff the poset of sorts $(S, \leqslant)$ has a maximal element, call it *State*, such that: (i) for all rules $(l \to r \text{ if } \phi) \in R$, $l, r \in T_\Omega(X)_{State}$; and (ii) for any $f : s_1 \dots s_n \to s$ in Ω

with $s \leqslant State$ we have $s_i \not\leq State$, $1 \leqslant i \leqslant n$. This ensures that if $[u] \in C_{\mathcal{R}} = T_{\Omega/B_\Omega}$, and $[u] \in C_{\mathcal{R},State}$, then all rewrites $u \rightarrow_{R,B} u'$ happen at the top position ϵ. This topmost requirement is easy to achieve in practice. In particular, it can *always* be achieved for *object-based rewrite theories*, which we explain next.

Object-Based Rewrite Theories. Most distributed systems, including the IBOS browser, can be naturally modeled by *object-based rewrite theories*. We give here a brief introduction and refer to [11, 27] for more details. The *distributed state* of an object-based system, called a *configuration*, is modeled as a *multiset* or "soup" of objects and messages built up by an *associative-commutative* binary multiset union operator (with juxtaposition syntax) $\_\ \_ : Conf\ \ Conf \rightarrow Conf$ with identity *null*. The sort *Conf* has two subsorts: a sort *Object* of *objects* and a sort *Msg* of *messages* "traveling" in the configuration from a sender object to a receiver object. The syntax for messages is user-definable, but it is convenient to adopt a conventional syntax for objects as record-like structures of the form: $\langle\, o \mid a_1(v_1), \ldots, a_n(v_n) \,\rangle$, where o is the object's name or *object identifier*, belonging to a subsort of a general sort *Oid*, and $a_1(v_1), \ldots, a_n(v_n)$ is a *set* of object *atributes* of sort *Att* built with an associative-commutative union operator $\_\ ,\ \_ : Atts\ \ Atts \rightarrow Atts$, with *null* as identity element and with $Att < Atts$. Each a_i is a constructor operator $a_i : s_i \rightarrow Att$ so that the *data value* v_i has sort s_i. Objects can be classified in *object classes*, so that a class C has an associated subsort $C.Oid < Oid$ for its object identifiers and associated attribute constructors $a_i : s_i \rightarrow Att$, $1 \leqslant i \leqslant n$. Usually, a configuration may have many objects of the same class, each with a different object identifier; but some classes (e.g., the *Kernel* class in IBOS) are *singleton classes*, so that only one object of that class, with a fixed name, will apear in a configuration. Another example in the IBOS specification is the singleton class *Display*. The single display object represents the rendering of the web page shown to the user and has the form: $< display \mid displayContent(D), activeTab(WA) >$, where the *activeTab* attribute constructor contains a reference to the web process that the user has selected (each tab corresponds to a different web process) and the *displayContent* constructor encapsulates the web page content currently shown on the display. Not all configurations of objects and messages are sensible. A configuration is *well-formed* iff it satisfies the following two requirements: (i) *unique object identifiers*: each object has a unique name different from all other object names; and (ii) *uniqueness of object attributes*: within an object, each object attribute *appears only once*; for example, an object like $< display \mid displayContent(D), activeTab(WA), activeTab(WA') >$ is nonsensical.

The *rewrite rules* R of an object-based rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$ have the general form $l \rightarrow r\ \ if\ \ \phi$, where l and r are terms of sort *Conf*. Intuitively, l is a pattern describing a *local fragment* of the overall configuration, so that a substitution instance $l\sigma$ describing a concrete such fragment (e.g., two objects, or an object and a message) can be rewritten to a new subfragment $r\sigma$, provided the rule's condition $\phi\sigma$ holds (see Section 4.1 for an example rule). Classes can be structured in *multiple inheritance hierarchies*, where a subclass C' of class C, denoted $C' < C$, may have additional attributes and has a subsort $C'.Oid < C.Oid$ for its object identifiers. By using extra variables $Attrs_j$ of sort *Atts* for "any extra attributes" that may appear in a subclass of any object o_j in the left-hand side patterns l of a rule, rewrite rules can be

automatically inherited by subclasses [27]. Furthermore, a subclass $C' < C$ may have *extra rules*, which may modify both its superclass attributes and its additional attributes.

An object-based rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$ *can easily be made topmost* as follows: (i) we add a fresh now sort *State* and an "encapsulation operator," say, $\{\_\} : Conf \rightarrow State$, and (ii) we trasform each rule $l \rightarrow r$ *if* ϕ into the rule $\{l\ C\} \rightarrow \{r\ C\}$ *if* ϕ, where C is a fresh variable of sort *Conf* modeling "the rest of the configuration," which could be empty by the identity axiom for *null*.

3 Constructor-Based Reachability Logic

Constructor-based reachability logic [43, 44] is a partial correctness logic generalizing Hoare logic in the following sense. In Hoare logic we have state predicates $A, B, C, \ldots$ and formulas are Hoare triples $\{A\}\, p\, \{B\}$ where A is the *precondition*, B is the *post-condition*, and p is the *program* we are reasoning about. Since a Maude program is a rewrite theory $\mathcal{R}$, a Hoare triple in Maude has the form $\{A\}\, \mathcal{R}\, \{B\}$, with the expected *partial correctness semantics*. But we can be more general and consider *reachability logic formulas* of the form: $A \rightarrow^{\circledast} B$, with A the *precondition* (as in Hoare logic) but with B what we call the formula's *midcondition*. That is, B need not hold *at the end* of a terminating computation, as in Hoare logic, but just *somewhere in the middle* of such a computation. A topmost rewrite theory $\mathcal{R}$ *satisfies* $A \rightarrow^{\circledast} B$, written $\mathcal{R} \models A \rightarrow^{\circledast} B$, iff along any terminating computation from an initial state $[u]$ satisfying A there is some intermediate state satisfying B. More precisely:

Definition 2. *Let $\mathcal{R} = (\Sigma, E \cup B, R)$ be topmost with top sort State and with free constructors Ω modulo B_Ω, and let A and B be state predicates for states of sort State, so that $[\![A]\!]$ and $[\![B]\!]$ denote the respective subsets of $T_{\Omega/B_\Omega,State}$ defined by A and B. Furthermore, let T be a state predicate of* terminating states *so that $[\![T]\!] \subseteq Term_{\mathcal{R}}$, where $Term_{\mathcal{R}} = \{[u] \in T_{\Omega/B_\Omega,State} \mid (\nexists [v])\ [u] \rightarrow_{\mathcal{R}} [v]\}$. Then, a reachability formula $A \rightarrow^{\circledast} B$ holds for the canonical reachability model $\mathcal{C}_{\mathcal{R}}$ of $\mathcal{R}$ relative to T in the* all paths satisfaction relation*, denoted $\mathcal{R} \models^{\forall}_{T} A \rightarrow^{\circledast} B$, iff for every sequence $[u_0] \rightarrow_{\mathcal{R}} [u_1] \ldots [u_{n-1}] \rightarrow_{\mathcal{R}} [u_n]$ with $[u_0] \in [\![A]\!]$ and $[u_n] \in [\![T]\!]$ there exists k, $0 \leqslant k \leqslant n$ such that $[u_k] \in [\![B]\!]$.*

When $[\![T]\!] = Term_{\mathcal{R}}$, we abbreviate the relation $\models^{\forall}_{T}$ to just $\models$.

We can define the satisfaction of a Hoare triple $\{A\}\, \mathcal{R}\, \{B\}$ as syntactic sugar for the satisfaction relation $\mathcal{R} \models A \rightarrow^{\circledast} (B \wedge Term_{\mathcal{R}})$. As explained in [44], the case of a Hoare logic for an *imperative* programming language $\mathcal{L}$ is obtained as syntactic sugar for a special class of reachability logic formulas for a rewrite theory $\mathcal{R}_{\mathcal{L}}$ giving a rewriting logic semantics to the programming language $\mathcal{L}$.

But in what sense is such a reachability logic *constructor-based*? In the precise sense that the state predicates $A, B, C, \ldots$ used in the logic are *constrained constructor pattern predicates* that exploit for symbolic purposes the extreme simplicity of the constructor theory (Ω, B_Ω), which is much simpler than the equational theory $(\Sigma, E \cup B)$.

Definition 3. *Let $(\Sigma, E \cup B)$ be convergent modulo B and sufficiently complete with respect to (Ω, B_Ω), i.e., $C_{\Sigma/E,B} = T_{\Omega/B_\Omega}$. A* constrained constructor pattern *is an expression $(u \mid \phi)$ such that $(u, \phi) \in T_\Omega(X) \times QFForm(\Sigma)$. The set of* constrained constructor

pattern predicates $PatPred(\Omega,\Sigma)$ *is defined inductively over* $T_\Omega(X)\times QFForm(\Sigma)$ *by adding* $\bot$ *and closing under* $(\vee)$ *and* $(\wedge)$*. We let capital letters* $A,B,\ldots,P,Q,\ldots$ *range over* $PatPred(\Omega,\Sigma)$*. The* semantics *of* $A\in PatPred(\Omega,\Sigma)$ *is the subset* $[\![A]\!]\subseteq C_{\Sigma/E,B}$ *such that:*

(i) $[\![\bot]\!]=\emptyset$,
(ii) $[\![u\mid\varphi]\!]=\{[(u\rho)!]_{B_\Omega}\in C_{\Sigma/E,B}\mid \rho\in[X\to T_\Omega]\ \wedge\ C_{\Sigma/E,B}\models\varphi\rho\}$,
(iii) $[\![A\vee B]\!]=[\![A]\!]\cup[\![B]\!]$, *and*
(iv) $[\![A\wedge B]\!]=[\![A]\!]\cap[\![B]\!]$.

For any sort s*, let* $PatPred(\Omega,\Sigma)_s\subseteq PatPred(\Omega,\Sigma)$ *where* $A\in PatPred(\Omega,\Sigma)_s$ *iff each subpattern* $(u\mid\phi)$ *of* A *has* $u\in T_\Omega(X)_s$*.* $A\in PatPred(\Omega,\Sigma)$ *is* normal *if it has no subpattern of the form* $P\wedge Q$*. If* B_Ω *has a finitary unification algorithm, normalization is effectively computable by (disjoint)* B_Ω *unification [43, 44].*

We can now fully specify our constructor-based reachability logic: it is a reachability logic for topmost rewrite theories $\mathcal{R}=(\Sigma,E\cup B,R)$ with top sort *State* and with free constructors Ω modulo B_Ω whose set of state predicate formulas is $PatPred(\Omega,\Sigma)_{State}$.

Reachability Logic Proof Rules [43, 44]. We review our proof system for reachability logic that was proved sound with respect to the all-paths satisfaction relation in [43, 44]. The proof rules derive sequents of the form $[\mathcal{A},\ C]\ \vdash_T\ A\to^{\circledast}B$, where $\mathcal{A}$ and C are finite sets of reachability formulas, $T\subseteq Term_{\mathcal{R}}$, and reachability formula $A\to^{\circledast}B$ is normalized. Formulas in $\mathcal{A}$ are called *axioms*; those in C are called *circularities*. The proof system has three rules: Step, Axiom, and Subsumption as well as derived rules Case, Split, and Substitution. See the brief overview in Table 1. The derived rules are explained in [42].

Table 1: Overview of Proof Rules

Assumptions:
1. $\mathcal{R}=(\Sigma,E\cup B,R)$ is suff. comp. w.r.t. (Ω,B_Ω) with $R=\{l_i\to r_i \text{ if } \phi_i\}_{i\in I}$
2. $P\to^{\circledast}(\bigvee_j v_j\mid\psi_j)\in\mathcal{A}$

Name	Rule	Condition
Step*	$\dfrac{\bigwedge_{i\in I,\alpha\in Y(i)}[\mathcal{A}\cup C,\ \emptyset]\ \vdash_T\ (r_i\mid\varphi'\wedge\phi_i)\alpha\to^{\circledast}B\alpha}{[\mathcal{A},\ C]\ \vdash_T\ u\mid\varphi\to^{\circledast}B}$	
Axiom	$\dfrac{\bigwedge_j[\mathcal{A},\ C]\ \vdash_T\ (v_j\alpha\mid\varphi\wedge\psi_j\alpha)\to^{\circledast}B}{[\mathcal{A},\ C]\ \vdash_T\ (u\mid\varphi)\to^{\circledast}B}$	$[\![u\mid\varphi]\!]\subseteq[\![P\alpha]\!]$
Sub.	$\dfrac{}{[\mathcal{A},\ C]\ \vdash_T\ A\to^{\circledast}B}$	$[\![A]\!]\subseteq[\![B]\!]$

*$\varphi'=\varphi\wedge\bigwedge\{\neg\psi\beta\ \mid\ (w\mid\psi)\in B\wedge\exists\beta\, w\beta=u\}$, $Y(i)=Unif_{B_\Omega}(u,l_i)$

This inference system has been mechanized using the Maude rewriting engine [43, 44]. Let us say a few words about its *automation*. The Step rule can be automated by

narrowing, i.e., symbolic rewriting where the lefthand side of a rewrite rule $l \to r$ if ϕ, instead of being *matched* by a term t to be rewritten, only B_Ω*-unifies* with it. Assuming B_Ω has a finitary unification algorithm, the narrowing relation lifted to constrained constructor patterns is *decidable*. The Axiom and Subsumption rules require automating a constrained constructor pattern subsumption check of the form $[\![u \mid \varphi]\!] \subseteq [\![v \mid \phi]\!]$; this can be shown to hold by B_Ω-matching $v\alpha =_{B_\Omega} u$ and checking whether $T_{\Sigma/E\cup B} \models \varphi \Rightarrow \phi\alpha$. Since B_Ω-matching is decidable, the only undecidable check is the implication's validity. Our tool employs multiple heuristics to check whether $T_{\Sigma/E\cup B} \models \varphi \Rightarrow \phi\alpha$. Given a goal $G = [\mathcal{A},\ C]\ \vdash_T\ A \to^{\circledast} B$, there are two final operations needed: (i) checking whether the proof failed, i.e., whether $[\![A]\!] \cap [\![T]\!] \neq \varnothing$ but not $[\![A]\!] \subseteq [\![B]\!]$, (ii) as an optimization, discarding a goal G where A's constraint is unsatisfiable. These two checks are also undecidable in general, so we rely on best-effort heuristics. Interestingly, thanks to their use of the symbolic techniques for variant unification and satisfiability [14, 32, 41], and for order-sorted congruence closure module axioms [30] mentioned in the Introduction, these heuristics were sufficient to complete all reachability logic proofs of the IBOS properties without relying on an external prover.

Proving Inductive Invariants [31, 44]. For a transition system $\mathcal{Q} = (Q, \to_Q)$ and a subset $Q_0 \subseteq Q$ of *initial states*, a subset $I \subseteq Q$ is called an *invariant* for $\mathcal{Q}$ from Q_0 iff for each $a \in Q_0$ and $b \in Q$, $a \to_Q^* b$ implies $b \in I$, where $\to_Q^*$ denotes the reflexive-transitive closure of $\to_Q$. A subset $A \subseteq Q$ is called *stable* in $\mathcal{Q}$ iff for each $a \in A$ and $b \in Q$, $a \to_Q b$ implies $b \in A$. An invariant I for $\mathcal{Q}$ from Q_0 is called *inductive* iff I is stable. We instantiate this generic framework to prove invariants over a topmost rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$ with $(\Sigma, B, \vec{E})$ convergent and sufficiently complete with respect to (Ω, B_Ω) and consider the transition system induced by $\mathcal{R}$ over sort *State*, i.e., $(C_{\Sigma/E\cup B,State}, \to_{\mathcal{R}})$.

To prove an invariant I from Q_0 over $\mathcal{R}$, we use a simple theory transformation mapping topmost theory $\mathcal{R}$ to a theory $\mathcal{R}_{stop}$ having a fresh operator $[\_]$ and a rule *stop* : $c(\vec{x}) \to [c(\vec{x})]$ for each constructor c of sort *State*. Then, by Corollary 1 in [43], to prove I is an invariant from Q_0 over $\mathcal{R}$ we prove $Q_0 \subseteq I$ and that the reachability formula $I\sigma \to^{\circledast} [I]$ holds over $\mathcal{R}$, where: (i) $I\sigma$ is a renaming of I with $vars(I) \cap vars(I\sigma) = \varnothing$ and (ii) if $I = (u \mid \varphi)$ then $[I] = ([u] \mid \varphi)$. If I is inductive and $I = u \mid \varphi$, the proof of $I\sigma \to^{\circledast} [I]$ proceeds as follows:

1. The initial sequent is $[\varnothing, I\sigma \to^{\circledast} [I]] \vdash_{[]} I\sigma \to^{\circledast} [I]$.
2. Apply the Step rule; based on which rule $\zeta : l \to r$ if ϕ was used, obtain:
   (a) if ζ is *stop*, $[I\sigma \to^{\circledast} [I], \varnothing] \vdash_{[]} [I\sigma] \to^{\circledast} [I]$;
   (b) otherwise, $\bigwedge_{\alpha\in Unif_{B_\Omega}(u,l)} [I\sigma \to^{\circledast} [I], \varnothing] \vdash_{[]} (r \mid \varphi \wedge \phi)\alpha \to^{\circledast} [I]$.
3. For case (2a), apply Subsumption. For case (2b), apply zero or more derived rules to obtain sequents of the form: $[I\sigma \to^{\circledast} [I], \varnothing] \vdash_{[]} A \to^{\circledast} [I]$.
4. Since I is assumed inductive, we have $[\![A]\!] \subseteq [\![I\sigma]\!]$. Thus, apply Axiom to derive $[I\sigma \to^{\circledast} [I], \varnothing] \vdash_{[]} [I] \to^{\circledast} [I]$.
5. Finally, apply Subsumption.

Since all the IBOS security properties *are invariants*, all our proofs follow steps (1)–(5) above.

4 IBOS and its Security Properties

One important security principle adopted by all modern browsers is the same-origin policy (SOP): it isolates web apps from different origins, where an origin is represented as a tuple of a protocol, a domain name, and a port number. For example, if a user loads a web page from origin (`https`,`mybank.com`,`80`) in one tab and in a separate tab loads a web page from origin (`https`,`mybnk.com`,`80`), i.e., a spoofed domain that omits the 'a' in 'mybank,' any code originating from the spoofed domain in the latter tab will not be able to interact with the former tab. SOP also ensures that asynchronous JavaScript and XML (AJAX) network requests from a web app are routed to its origin. Unfortunately, browser vendors often fail to correctly implement the SOP policy [9, 39].

The Illinois Browser Operating System (IBOS) is an operating system and web browser that was designed and modeled in Maude with security and compatibility as primary goals [38, 48, 49]. Unlike commodity browsers, where security checks are implemented across millions of lines of code, in IBOS a small trusted computing base of only 42K code lines is established in the *kernel* through which all network interaction is filtered. What this means in practice is that, even if highly complex HTML rendering or JavaScript interpretation code is compromised, the browser *still* cannot violate SOP (and several other security properties besides). The threat model considered here allows for much of the browser code itself to be compromised while still upholding the security invariants we describe below.

In the following subsections we survey the IBOS browser and SOP, how the IBOS browser can be formally specified as a rewriting logic theory, and how the SOP and other IBOS security properties can be formally specified as invariants.

4.1 IBOS System Specification

IBOS System Design. The Illinois Browser Operating System is an operating system and web browser designed to be highly secure as a browser while maintaining compatibility with modern web apps. It was built on top of the micro-kernel L4Ka::Pistachio [22, 23], which embraces the principles of least privilege and privilege separation by separating operating subsystems into separate daemons that communicate through the kernel via checked inter-process communication (IPC). IBOS directly piggybacks on top of this micro-kernel design by implementing various browser abstractions, such as the browser chrome and network connections, as separate components that communicate using L4ka kernel message passing infrastructure. Figure 1 gives an overview of the IBOS architecture; as an explanatory aid we highlight a few key objects:

- **Kernel.** The IBOS kernel is built on top of the L4Ka::Pistachio micro-kernel which, as noted previously, can check IPC messages against security policies.
- **Network Process.** A network process is responsible for managing a network connection (e.g., HTTP connections) to a specific origin. It understands how to encode and decode TCP datagrams and Ethernet frames and can send and receive frames from the network interface card (NIC).
- **Web Application.** A web application represents a specific instance of a web page loaded in a particular browser tab (e.g., when a link is clicked or a URL is entered

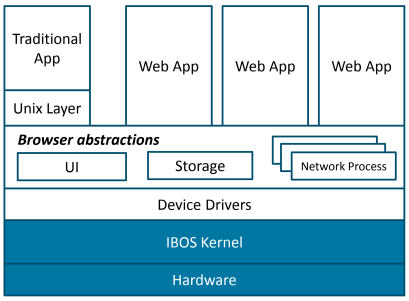


Fig. 1: IBOS System Architecture [38].

into the address bar). Web applications know how to render HTML documents. As per SOP, each web page is labeled by its origin.

– **Browser UI.** The browser user inferface (UI) minimally includes the address bar and the mouse pointer and extends to any input mechanism.
– **Display.** The display represents the rendered web app shown to the user; it is blank when no web app has loaded. For security, it cannot modify the UI.

IBOS System Specification in Maude. We present an overview of the IBOS formal executable specification as a Maude rewrite theory, which closely follows previous work [34, 37, 38]; a more detailed explanation can be found in [42]. We model IBOS as an object-based rewrite theory (see Section 2). We use *italics* to write Maude rewrite rules and *CamelCase* for variable or sort names. Some objects of interest include the singleton objects *kernel*, *ui*, and *display* (in classes *Kernel*, *UI*, and *Display*, respectively). We also have non-singleton classes *WebApp* and *NetProc* representing web apps and network processes in IBOS, respectively.

As an example, let us consider the specification of the *change-display* rewrite rule shown in Figure 2.

$$\{\langle \mathit{display} \mid \mathit{displayContent}(D), \mathit{activeTab}(WA), \mathit{Atts}\rangle \langle \mathit{WA} \mid \mathit{rendered}(D'), \mathit{Atts}'\rangle \mathit{Conf}\}$$
$$\rightarrow \{\langle \mathit{display} \mid \mathit{displayContent}(D'), \mathit{activeTab}(WA), \mathit{Atts}\rangle \langle \mathit{WA} \mid \mathit{rendered}(D'), \mathit{Atts}'\rangle \mathit{Conf}\}$$
$$\text{if } D \neq D'$$

Fig. 2: *change-display* rule

This rule involves the *display* object and the *WebApp* designated as the display's *activeTab*. In our model, the *rendered* attribute of a web app represents its current rendering of the HTML document located at its origin. When the web app is first created, its *rendered* attribute has the value *about-blank*, i.e., nothing has yet been rendered. Thus, this rule essentially states that, at any time, the displayed content can be replaced by currently rendered HTML document of the active tab, *only if* it is different from the currently displayed content.

Our IBOS browser specification contains 23 rewrite rules and is about 850 lines of Maude code; it is available at `https://github.com/sskeirik/ibos-case-study`.

4.2 Specification of the IBOS Security Properties

We first describe at a high level the security properties that we will formally specify and verify. The key property that we verify is the *same-origin policy* (SOP), but we also specify and verify the *address bar correctness* (ABC) property. Our discussion follows that of [37, 38], based on invariants P_1-P_{11}:

- P_1 Network requests from web page instances must be handled by the proper network process.
- P_2 Ethernet frames from the network interface card (NIC) must be routed to the proper network process.
- P_3 Ethernet frames from network processes to the NIC must have an IP address and TCP port that matches the origin of the network process.
- P_4 Data from network processes to web page instances must be from a valid origin.
- P_5 Network processes for different web page instances must remain isolated.
- P_6 The browser chrome and web page content displays are isolated.
- P_7 Only the current tab can access the screen, mouse, and keyboard.
- P_8 All components can only perform their designated functions.
- P_9 The URL of the active tab is displayed to the user.
- P_{10} The displayed web page content is from the URL shown in the address bar.
- P_{11} All configurations are well-formed, i.e., non-duplication of *Oid*s and *Att*s.

We define same-origin policy as SOP $= \bigwedge_{1 \leqslant i \leqslant 7} P_i$; address-bar correctness is specified as ABC $= P_{10}$. Note that $P_9 \wedge P_{10} \Rightarrow P_7$. Since P_5, P_6, and P_8 follow directly from the model design, it is sufficient to prove $\bigwedge_{i \in I} P_i$ for $I = \{1, 2, 3, 4, 9, 10\}$. Invariant P_{11} is new to our current formalization, but is implicitly used in prior work; it forbids absurd configurations, e.g., having two *kernel*s or a *WebApp* that has two *URL*s (see Section 2). Due to its fundamental relation to how object-based rewrite theories are defined, we need P_{11} in the proof of *all* other invariants. As an example of how these invariant properties can be formalized in our model as constrained pattern predicates, we show how the address bar correctness invariant can be specified in our system below:

$$\{\langle \mathit{kernel} \mid \mathit{addrBar}(U), \mathit{Atts} \rangle \\ \langle \mathit{display} \mid \mathit{displayContent}(U'), \mathit{Atts}' \rangle \; \mathit{Conf}\} \mid U' \trianglelefteq U \quad \text{(ABC)}$$

where $U' \trianglelefteq U \Leftrightarrow_{\text{def}} (U' \neq \mathit{about\text{-}blank} \Rightarrow U = U')$, i.e., the *display* is either blank *or* its contents' origin matches the address bar. Note that, for simplicity, in our model, we identify displayed content with its origin URL. As a second example, consider how to formalize invariant P_9:

$$\big\{\langle \mathit{kernel} \mid \mathit{addrBar}(U), \mathit{Atts} \rangle \\ \langle \mathit{display} \mid \mathit{activeTab}(\mathit{WA}), \mathit{Atts}' \rangle \\ \langle \mathit{WA} \mid \mathit{URL}(U''), \mathit{Atts}'' \rangle \; \mathit{Conf}\big\} \mid U \trianglelefteq U'' \quad (P_9)$$

i.e., the address bar must match the URL of the active tab, unless the address shown is *about-blank*, i.e., nothing is shown. Finally, P_{11} has a trivial encoding: $\{\mathit{Conf}\} \mid$

$WF(Conf)$, where $WF : Conf \rightarrow Bool$ is the well-formedness predicate. Our specification has 200 lines of Maude code to specify the pattern predicates used in our invariants and another 900 lines of code specifying all of the auxiliary functions and predicates. As stated in the Introduction, we additionally must prove that: (a) the IBOS system specification extended by our security property specification is *suitable* for symbolic reachability analysis; and (b) our ABC and SOP invariants are at least as strong as the corresponding invariants in prior work [37, 38] ABC_0 and SOP_0. Proof obligation (a) can be met by using techniques for proving *ground convergence* and *sufficient completeness* of conditional equational theories, while proof obligation (b) can be reduced to proving associated implications, e.g., $SOP \Rightarrow SOP_0$. We have carried out full proofs of (a) and (b); due to space limitations, the full details are available at [40].

5 Proof of IBOS Security Properties

In this section, due to space limitations, we give a high-level overview of our proof methodology for verifying the SOP and ABC properties for IBOS, and show a subproof used in deductively verifying ABC. Each proof script has roughly 200 lines of code and another 20 to specify the reachability logic sequent being proved.

Modular Proof Methodology. In this subsection we survey our modular proof methodology for proving invariants using reachability logic and comment on how we exploit modularity in three key ways, i.e., we show how we efficiently structured and carried out our proofs by decomposing them into composable, independent, and reusable pieces.

Most of the IBOS proof effort was spent strengthening an invariant I into an inductive invariant I_{ind}, where I is either SOP or ABC. Typically, I_{ind} is obtained by iteratively applying the proof strategy given in Section 2. In each round, assume candidate I' is inductive and attempt to complete the proof. If, after applying the Step rule (and possibly some derived rules), an application of Axiom is impossible, examine the proof of failed pattern subsumption $[\![A]\!] \subseteq [\![I']\!]$ in the side-condition of Axiom. If pattern formula C (possibly using new functions and predicates defined in theory Δ) can be found that might enable the subsumption proof to succeed, try again with candidate $I' \wedge C$. In parallel, our system specification $\mathcal{R}$ is enriched by extending the underlying *convergent* equational theory $\mathcal{E}$ to $\mathcal{E} \cup \Delta$ to obtain the enriched rewrite theory $\mathcal{R}_\Delta$.

The first kind of modularity we exploit is *rule modularity*. Recall that any reachability logic proof begins with an application of the Step rule. Since Step must consider the result of symbolically rewriting the initial sequent with all possible rewrite rules, we can equivalently construct our proof on a "rule-by-rule" basis, i.e., if $\mathcal{R} = (\Sigma, E \cup B, \{l_j \rightarrow r_j \text{ if } \phi_j\}_{j \in J})$, we can consider $|J|$ separate reachability proofs using respective theories $\mathcal{R}_j = (\Sigma, E \cup B, \{l_j \rightarrow r_j \text{ if } \phi_j\})$ for $j \in J$. Thus, we can focus on strengthening invariant I' for each rule $j \in J$ separately.

Another kind of modularity that we can exploit is *subclass modularity*. Because we are reasoning in an object-based rewrite theory, each rule mentions one or more objects in one or more classes, and describes how they evolve. Recall from Sect. 2 that subclasses must contain all of the attributes of their superclass, but may define additional attributes and have additional rewrite rules which affect them. The upshot of

all this is that if we *refine* our specification by instantiating objects in one class into some subclass, any invariants proved for the original rules immediately hold for the same rules in the refined specification. Because of rule modularity, we need only prove our invariants hold for the newly defined rules. Even among newly defined rules, all *non-interfering* rules trivially satisfy any already proved invariants, where non-interfering rules do not directly or indirectly influence the state of previously defined attributes.

Lastly, we exploit what we call *structural modularity*. Since our logic is constructor-based and we assume that B_Ω-matching is decidable, by pattern matching we can easily specify a set S of sequents to which we can apply the *same* combination of derived proof rules. This is based on the intuition that syntactically similar goals typically can be proved in a similar way. More concretely, given a set of reachability formulas S and pattern $p \in T_\Omega(X)$, we can define the subset of formulas $S_p = \{(u \mid \varphi) \rightarrow^{\circledast} B \in S \mid \exists \alpha \in [X \rightarrow T_\Omega(X)]\, p\alpha =_{B_\Omega} u\}$. Although the simplified example below does not illustrate structural modularity, we have heavily exploited this principle in our formal verification of IBOS.

Address Bar Correctness Proof Example. Here we show a snippet of the ABC invariant verification, namely, we prove that the invariant holds for the *change-display* rule by exploiting rule modularity as noted above. As mentioned in our description of invariant P_{11}, well-formedness is required for all invariants. Therefore, we begin with ABC $\wedge$ P_{11} as our candidate inductive invariant, which, as mentioned in Sect. 2, normalizes by disjoint B_Ω-unification to the invariant:

$$\{\langle \mathit{kernel} \mid \mathit{addrBar}(U), \mathit{Atts}\rangle \\ \langle \mathit{display} \mid \mathit{displayContent}(U'), \mathit{Atts}'\rangle\ \mathit{Conf}\} \mid U' \trianglelefteq U \wedge \mathit{WF}(\ldots)$$

where for brevity ... expands to the entire term of sort *Conf* wrapped inside operator $\{\_\}$, i.e., the entire configuration is *well-formed*. Recall the definition of the *change-display* rule in Section 4.1. We can see that our invariant only mentions the *kernel* and *display* processes, whereas in rule *change-display* the value of *displayContent* depends on the *rendered* attribute of a *WebApp*, i.e., the one selected as the *activeTab*. Clearly, our invariant seems too weak. How can we strengthen it? The reader may recall P_9, which states "the URL of the active tab is displayed to the user." Thus, by further disjoint B_Ω-unification, the strengthened invariant ABC $\wedge$ P_{11} $\wedge$ P_9 normalizes to:

$$\{\langle \mathit{kernel} \mid \mathit{addrBar}(U), \mathit{Atts}\rangle\, \langle \mathit{display} \mid \mathit{displayContent}(U'), \mathit{activeTab}(\mathit{WA}), \mathit{Atts}'\rangle \\ \langle \mathit{WA} \mid \mathit{URL}(U''), \mathit{Atts}''\rangle\ \mathit{Conf}\} \mid U' \trianglelefteq U \wedge U \trianglelefteq U'' \wedge \mathit{WF}(\ldots)$$

This new invariant is closer to what we need, since the pattern now mentions the particular web app we want. Unfortunately, since our invariant still knows nothing about the *rendered* attribute, at least one further strengthening is needed.

At this point, we can enrich our theory with a new predicate stating that the *rendered* and *URL* attributes of any *WebApp* always agree[3]. Let us declare it as $R : \mathit{Conf} \rightarrow \mathit{Bool}$. We can define it inductively over configurations by:

[3] Note that, in a very real sense, this requirement is at the heart of the SOP, since it means that any *WebApp* has indeed obtained content from its claimed origin.

$$R(\langle WA \mid rendered(U), URL(U'), Atts\rangle\ Conf) = U \trianglelefteq U' \wedge R(Conf) \qquad (R_1)$$
$$R(\langle P \mid Atts\rangle\ Conf) = R(Conf)\ if\ \neg WA(P) \qquad (R_2)$$
$$R(none) = \top \qquad (R_3)$$

where $WA : Oid \rightarrow Bool$ ambiguously denotes a predicate that holds iff an *Oid* refers to a web app. Intuitively, it says that whatever a *WebApp* has *rendered* is either blank or has been loaded from its *URL*. The strengthened invariant becomes:

$$\{\langle kernel \mid addrBar(U), Atts\rangle\ \langle display \mid displayContent(U'), activeTab(WA), Atts'\rangle$$
$$\langle WA \mid URL(U''), Atts''\rangle\ Conf\ \} \mid$$
$$U' \trianglelefteq U \wedge U \trianglelefteq U'' \wedge WF(\ldots) \wedge R(\ldots)$$

where, as before, the ... represents the entire term of sort *Conf* enclosed in $\{\_\}$. Now, all of the required relationships between variables in the rewrite rule seem to be accounted for. Uneventfully, with the strengthened invariant $ABC \wedge P_{11} \wedge P_9 \wedge R$, the subproof for the *change-display* rule now succeeds.

6 Related Work and Conclusions

Related Work on IBOS Verification. In this paper, we have presented the first full deductive verification of SOP and ABC for IBOS. Note that in [37, 38], SOP and ABC were also verified. Their approach consisted of a hand-written proof that any counterexample must appear on some trace of length n plus bounded model checking showing that such counterexamples are unreachable on all traces of length n. In [34], an attempt was made to deductively verify these same invariants via the Maude invariant analyzer. Though a few basic invariants were proved, due to thousands of generated proof obligations, none of the properties listed in Sect. 4.2 were verified. Compared to previous work, this paper presents the first full deductive verification of IBOS security properties.

Related Work on Browser Security. In terms of computer technology, the same-origin policy is quite old: it was first referenced in Netscape Navigator 2 released in 1996 [1]. As [39] points out, different browser vendors have different notions of SOP. Here, we situate IBOS and our work into this larger context. Many papers have been written on policies for enforcing SOP with regards to frames [2], third-party scripts [18,21], cached content [19], CSS [17], and mobile OSes [50]. Typically, these discussions assume that browser code is working as intended and then show existing policies are insufficient. Instead, IBOS attacks the problem taking a different tack: even if the browser itself is compromised, can SOP still be ensured? What the IBOS verification demonstrates is that —by using a minimal trusted computing base in the kernel, implementing separate web frames as separate processes, and requiring all IPC to be kernel-checked— one can in fact enforce the standard SOP notions, *even if* the complex browser code for rendering HTML or executing JavaScript is *compromised*. Although our model does not treat JavaScript, HTML, or cookies, explicitly, since it models system calls which are used

for process creation, network access, and inter-process communication, code execution and resource references can be abstracted away into the communication primitives they ultimately cause to be invoked, allowing us to perform strong verification in a high-level fashion. [7] surveys many promising lines of research in the formal methods web security landscape. Prior work on formal and declarative models of web browsers includes [3] as well as the *executable* models [4, 5]. Our work complements the Quark browser design and implementation of [20]: Quark, like IBOS, has a small trusted kernel (specified in Coq). In addition to proving tab non-interference and address bar correctness theorems, the authors use Coq code extraction to produce a verified, functional browser. Unlike Quark, whose TCB includes the entire Linux kernel and Coq code extraction tools, the TCB of the IBOS browser consists of only 42K lines of C/C++ code.

Related Work on Reachability Logic. Our work on *constructor-based* reachability logic [43, 44] builds upon previous work on reachability logic [35, 36, 46, 47] as a language-generic approach to program verification, parametric on the operational semantics of a programming language. Our work extends in a non-trivial manner reachability logic from a programming-language-generic logic of programs to a rewrite-theory-generic logic to reason about *both* distributed system designs and programs based on rewriting logic semantics. Our work in [43, 44] was also inspired by the work in [25], which for the first time considered reachability logic for rewrite theories, but went beyond [25] in several ways, including more expressive input theories and state predicates, and a simple inference system as opposed to an algorithm. Also related to our work in [43, 44], but focusing on *coinductive reasoning*, we have the recent work in [10, 24, 33], of which, in spite of various substantial differences, the closest to our work regarding the models assumed, the kinds of reachability properties proved, and the state predicates and inference systems proposed is the work in [10].

Conclusion and Future Work. We have presented a full deductive proof of the SOP and ABC properties of the IBOS browser design, as well as a prover and a modular reachability logic verification methodology making proofs scalable to substantial proof efforts like that of IBOS. Besides offering a case study that can help other distributed system verification efforts, this work should also be seen as a useful step towards incorporating the IBOS design ideas into future fully verified browsers. The web is alive and evolving; these evolution necessitates that formal approaches evolve as well. Looking towards the future of IBOS, two goals stand out: (i) extending the design of IBOS to handle some recent extensions of the SOP, e.g., cross-origin resource sharing (CORS) to analyze potential cross-site scripting (XSS) and cross-site request forgery attacks (XSRF) [16], and to check for incompatible content security policies (CSP) [45] in relation to SOP; by exploiting subclass and rule modularity, the verification of an IBOS extension with such new functionality could reuse most of the current IBOS proofs, since extra proofs would only be needed for the new, functionality-adding rules; and (ii) exploiting the intrinsic concurrency of Maude rewrite theories to transform them into correct-by-construction, deployable Maude-based distributed system implementations, closing the gap between verified designs and correct implementations. Our work on IBOS takes one more step towards demonstrating that a formally secure web is possible in a connected world where security is needed more than ever before.

References

1. JavaScript Guide (1.2). Netscape Communications Corporation (1997), originally http://developer.netscape.com/docs/manuals/communicator/jsguide4/index.htm; accessed at https://www.cs.rit.edu/~atk/JavaScript/manuals/jsguide/
2. Barth, A., Jackson, C., Mitchell, J.C.: Securing frame communication in browsers. Communications of the ACM **52**(6), 83–91 (2009)
3. Bauer, L., Cai, S., Jia, L., Passaro, T., Stroucken, M., Tian, Y.: Run-time monitoring and formal analysis of information flows in chromium. In: NDSS (2015)
4. Bohannon, A.: Foundations of web script security. Citeseer (2012)
5. Bohannon, A., Pierce, B.C.: Featherweight Firefox: formalizing the core of a web browser. In: Proceedings of the 2010 USENIX conference on Web application development. pp. 11–11. Usenix Association (2010)
6. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theoretical Computer Science **360**(1-3), 386–414 (2006)
7. Bugliesi, M., Calzavara, S., Focardi, R.: Formal methods for web security. Journal of Logical and Algebraic Methods in Programming **87**, 110–126 (2017)
8. Chen, S., Meseguer, J., Sasse, R., Wang, H.J., Wang, Y.M.: A systematic approach to uncover security flaws in gui logic. In: IEEE Symposium on Security and Privacy. pp. 71–85. IEEE (2007)
9. Chen, S., Ross, D., Wang, Y.M.: An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In: ACM Conference on Computer and Communications Security. pp. 2–11. ACM (2007)
10. Ştefan Ciobâcă, Lucanu, D.: A coinductive approach to proving reachability properties in logically constrained term rewriting systems. In: Proc. IJCAR 2018. Lecture Notes in Computer Science, vol. 10900, pp. 295–311. Springer (2018)
11. 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)
12. Dershowitz, N., Jouannaud, J.P.: Rewrite systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Vol. B, pp. 243–320. North-Holland (1990)
13. Durán, F., Meseguer, J.: On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. The Journal of Logic and Algebraic Programming **81**(7-8), 816–850 (2012)
14. Escobar, S., Sasse, R., Meseguer, J.: Folding variant narrowing and optimal variant termination. The Journal of Logic and Algebraic Programming **81**, 898–928 (2012)
15. 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)
16. Gollmann, D.: Problems with same origin policy: Know thyself. In: Security Protocols XVI. pp. 84–85. Springer, Berlin, Heidelberg (2011)
17. Huang, L.S., Weinberg, Z., Evans, C., Jackson, C.: Protecting browsers from Cross-origin CSS attacks. pp. 619–629. CCS '10, ACM, New York, NY, USA (2010)
18. Jackson, C., Barth, A.: Beware of finer-grained origins. Web (2008)
19. Jackson, C., Bortz, A., Boneh, D., Mitchell, J.C.: Protecting browser state from web privacy attacks. In: Proceedings of the 15th international conference on World Wide Web. pp. 737–744. ACM (2006)
20. Jang, D., Tatlock, Z., Lerner, S.: Establishing browser security guarantees through formal shim verification. In: Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12). pp. 113–128 (2012)

21. Karlof, C., Shankar, U., Tygar, J.D., Wagner, D.: Dynamic pharming attacks and locked same-origin policies for web browsers. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 58–71. ACM (2007)
22. Klein, G., Tuch, H.: Towards verified virtual memory in L4. TPHOLs Emerging Trends **4**, 16 (2004)
23. Kolanski, R., Klein, G.: Formalising the L4 microkernel API. In: Proceedings of the 12th Computing: The Australasian Theroy Symposium-Volume 51. pp. 53–68. Australian Computer Society, Inc. (2006)
24. Lucanu, D., Rusu, V., Arusoaie, A.: A generic framework for symbolic execution: A coinductive approach. Journal of Symbolic Computing **80**, 125–163 (2017)
25. Lucanu, D., Rusu, V., Arusoaie, A., Nowak, D.: Verifying reachability-logic properties on rewriting-logic specifications. In: Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday. vol. 9200, pp. 451–474. Springer LNCS (2015)
26. Lucas, S., Meseguer, J.: Normal forms and normal theories in conditional rewriting. Journal of Logical and Algebraic Methods in Programming **85**(1), 67–97 (2016)
27. 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)
28. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Proc. WADT'97. pp. 18–61. Springer LNCS 1376 (1998)
29. Meseguer, J.: Twenty years of rewriting logic. J. Algebraic and Logic Programming **81**, 721–781 (2012)
30. Meseguer, J.: Order-sorted rewriting and congruence closure. In: Proc. FOSSACS 2016. Lecture Notes in Computer Science, vol. 9634, pp. 493–509. Springer (2016)
31. Meseguer, J.: Generalized rewrite theories, coherence completion and symbolic methods. Tech. Rep. http://hdl.handle.net/2142/102183, University of Illinois Computer Science Department (December 2018)
32. Meseguer, J.: Variant-based satisfiability in initial algebras. Sci. Comput. Program. **154**, 3–41 (2018)
33. Moore, B.: Coinductive Program Verification. Ph.D. thesis, University of Illinois at Urbana-Champaign (2016), http://hdl.handle.net/2142/95372
34. Rocha, C.: Symbolic Reachability Analysis for Rewrite Theories. Ph.D. thesis, University of Illinois at Urbana-Champaign (2012)
35. Rosu, G., Stefanescu, A.: Checking reachability using matching logic. In: Proc. OOPSLA 2012. pp. 555–574. ACM (2012)
36. Rosu, G., Stefanescu, A.: From Hoare logic to matching logic reachability. In: Giannakopoulou, D., Méry, D. (eds.) FM. Lecture Notes in Computer Science, vol. 7436, pp. 387–402. Springer (2012)
37. Sasse, R.: Security models in rewriting logic for cryptographic protocols and browsers. Ph.D. thesis, University of Illinois at Urbana-Champaign (2012), http://hdl.handle.net/2142/34373
38. Sasse, R., King, S.T., Meseguer, J., Tang, S.: IBOS: A correct-by-construction modular browser. In: FACS 2012. Lecture Notes in Computer Science, vol. 7684, pp. 224–241. Springer (2013)
39. Schwenk, J., Niemietz, M., Mainka, C.: Same-origin policy: Evaluation in modern browsers. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 713–727. USENIX Association, Vancouver, BC (2017)
40. Skeirik, S.: Rewriting-Based Symbolic Methods for Distirbuted System Analysis. Ph.D. thesis, University of Illinois at Urbana-Champaign (2019)

41. Skeirik, S., Meseguer, J.: Metalevel algorithms for variant satisfiability. Journal of Logical and Algebraic Methods in Programming **96**, 81–110 (2018)
42. Skeirik, S., Meseguer, J., Rocha, C.: Verification of the IBOS browser security properties in reachability logic. Tech. rep. (2020), arxiv:2005.12232
43. Skeirik, S., Stefanescu, A., Meseguer, J.: A constructor-based reachability logic for rewrite theories. In: Proc. Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017. Lecture Notes in Computer Science, vol. 10855, pp. 201–217. Springer (2017)
44. Skeirik, S., Stefanescu, A., Meseguer, J.: A constructor-based reachability logic for rewrite theories. Fundam. Inform. **173**(4), 315–382 (2020)
45. Some, D.F., Bielova, N., Rezk, T.: On the content security policy violations due to the same-origin policy. pp. 877–886. WWW '17, Republic and Canton of Geneva, Switzerland (2017)
46. Stefanescu, A., Ştefan Ciobâcă, Mereuta, R., Moore, B.M., Serbanuta, T., Rosu, G.: All-path reachability logic. In: Proc. RTA-TLCA 2014. vol. 8560, pp. 425–440. Springer LNCS (2014)
47. Stefanescu, A., Park, D., Yuwen, S., Li, Y., Rosu, G.: Semantics-based program verifiers for all languages. In: Proc. OOPSLA 2016. pp. 74–91. ACM (2016)
48. Tang, S.: Towards Secure Web Browsing. Ph.D. thesis, University of Illinois at Urbana-Champaign (2011), 2011-05-25, http://hdl.handle.net/2142/24307
49. Tang, S., Mai, H., King, S.T.: Trust and protection in the illinois browser operating system. In: 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings. pp. 17–32. USENIX Association (2010)
50. Wang, R., Xing, L., Wang, X., Chen, S.: Unauthorized origin crossing on mobile platforms: Threats and mitigation. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security. pp. 635–646. CCS '13, ACM, New York, NY, USA (2013)

Automated Construction of Security Integrity Wrappers for Industry 4.0 Applications

Carolyn Talcott[1] and Vivek Nigam[2,3]

[1] SRI International, Menlo Park, USA, clt@csl.sri.com
[2] fortiss, Munich, Germany, nigam@fortiss.org
[3] Federal University of Paraíba, João Pessoa, Brazil

Abstract. Industry 4.0 (I4.0) refers to the trend towards automation and data exchange in manufacturing technologies and processes which include cyber-physical systems, where the internet of things connect with each other and the environment via networking. This new connectivity opens systems to attacks, by, *e.g.*, injecting or tampering with messages. The solution supported by standards such as OPC-UA is to sign and/or encrypt messages. However, given the limited resources of devices, instead of applying crypto algorithms to all messages in the network, it is better to focus on the messages that if tampered with or injected, could lead to undesired configurations.
This paper describes a framework for developing and analyzing formal executable specifications of I4.0 applications in Maude. The framework supports the engineering design workflow using theory transformations that include algorithms to enumerate network attacks leading to undesired states, and to determine wrappers preventing these attacks. In particular, given a deployment map from application components to devices we define a theory transformation that models execution of the application on the given set of (networked) devices. Given an enumeration of attacks (message flows) we define a further theory transformation that wraps each device with policies for signing/signature checking for just those messages needed to prevent the attacks.

1 Introduction

Manufacturing technologies and processes are increasingly automated with highly interconnected components that may include simple sensors and controllers as well as cyber-physical systems and the Internet of Things (IoT) components. This trend is sometimes referred to Industry 4.0 (I4.0). Among other benefits, I4.0 enables process agility and product specialization. This increase of interconnectivity has also enabled cyber-attacks. These attacks can lead to catastrophic events possibly leading to material and human damages. For example, after an attack on a steel mill, the factory had to stop its production leading to great financial loss [1].

A recent BSI report on the security of OPC-UA (machine to machine communication protocol for industrial automation) [7], points out that the lack of signed and encrypted messages on sensitive parts of the factory network can lead to high risk attacks. For example, attackers can inject or tamper with messages, confusing factory controllers and ultimately leading to a stalled or fatal state. Given the limited bandwidth

and processing power of I4.0 elements, instead of signing all messages, it is much better to only sign the messages that when not protected could be modified or injected by an intruder to lead to an undesirable situation. This leads to the question of how to determine critical communications.

This paper presents a formal framework for specifying I4.0 applications and analyzing safety and security properties using Maude [4]. The engineering development process from application design and testing to systems deployment is captured by theory transformations with associated theorems showing that results of analysis carried out at the abstract application level hold for models of deployed systems.

Our key contributions are as follows:

- **I4.0 Application Behavior:** We present a formal executable model of the behavior of I4.0 applications in the rewriting logic system Maude [4]. An application is composed of interacting state transition machines which, following the IEC 61499 standard [20], we call function blocks. Maude's search capability is used to formally check such applications for logical defects, which may lead to unsafe conditions.
- **Bounded Symbolic Intruder Model:** To evaluate the security of an application, we formalize a family of bounded intruders parameterized by the number of messages the intruder can inject. Our intruder can generate any clear text message, but can not generate (or read) messages signed by honest devices. We reduce the search space by defining a symbolic intruder. Proof of the *Intruder Theorem* shows that the two intruder models yield the same attacks.
- **Deployment transformation:** The application model is suited to reason about functionality and message flows, but does not support reasoning about resources and communication issues that arise when function blocks run on different devices. We define a theory transformation from an application executable specification to a specification of a deployment of that application using a map from application function blocks to a given set of devices. We prove that it is enough to carry out security verification in the application level as their properties can be transfered to deployed applications.
- **Security Integrity Wrappers:** Use of security wrappers is a mechanism to protect communications [3]. Here it is used to secure message integrity between devices using signing. Since signing is expensive, it is important to minimize message signing. We define a transformation from a specification of a deployed application to one in which devices are wrapped with a policy enforcement layer where the policies are computed from a set of message flows that must be protected as determined by the enumeration of possible attacks. The proof of the *Wrapping theorem* shows that the wrapping transformation protects the deployed system against identified attacks.

We have implemented the framework and carried out a number of experiments demonstrating analysis, deployment, and wrapping for variations of a PickNPlace application. The Maude code along with documentation, scenarios, sample runs and a technical report with details and proofs can be found at `https://github.com/SRI-CSL/WrapPat.git`. An early version of the framework was presented in [15] where we demonstrated the use of the search command to find logical defects and enumerate attacks, and proposed the idea of device wrappers. That paper contains a number of experiments, including scalability results. The new contributions include the the-

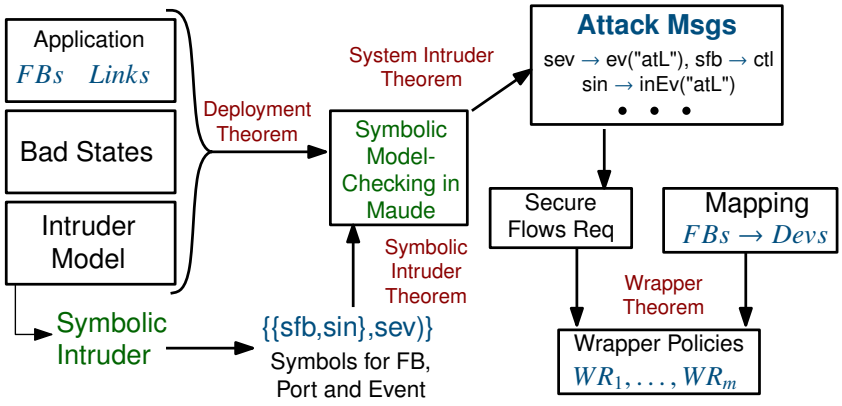


Fig. 1. Methodology Overview

orems and proofs, implementation of the deployment and wrapping functions, and a simplified version of the symbolic intruder model.

2 Overview

Threat Model We assume that devices have their pair of secret and public key. Moreover, that devices can be trusted and that a secret key is only known by its corresponding device. However, the communication channels shared by devices are not trusted. An intruder can, for example, inject and tamper (unsigned) messages in any communication channel. This intruder model reflects the critical types of attacks in Industry 4.0 applications as per the BSI report [7].

To protect communications between function blocks on different devices we use the idea of formal wrapper [3] to transform a system `S` into a system, `wrap(S,emsgs)`, in which system devices are wrapped in a policy layer protecting communications between devices by signing messages and checking signatures on flows.

Intuitively, a security integrity wrapper specifies which incoming events a device will accept only if they are correctly signed and which outgoing events should be signed. By using security integrity wrappers it is possible to prevent injection attacks. For example, if all possible incoming events expected in a device are to be signed, then any message injected by an intruder would be rejected by the device. However, more messages in security integrity wrappers means greater computational and network overhead. One goal of our work is to derive security integrity wrappers, $WR_1, \ldots, WR_n$, for devices, $Dev_1, \ldots, Dev_n$ in which software, called function blocks, are to be executed, to ensure the security of an application without needing to sign all events.

Figure 1 depicts the key components in achieving this goal with the inputs:

- **Application (App):** a set, $\{FB_1, \ldots, FB_n\}$, of function blocks (FBs) and links, $Links$ between output and target ports. An FB is a finite state machine similar to a Mealy Machine. An App executes its function blocks in cycles. In each cycle, the input pool is delivered to function block inputs and each function block fires one transition if possible.
- **Bad State:** a predicate (`badstate`) specifying which FB states should be avoided, for example, states that correspond to catastrophic situations.

– **Intruder Capabilities:** The intruder is given a set of all possible messages deliverable in the given application. For up to n times the intruder can pick a message from this set and inject it into the application input pool at any moment of execution.

We use a symbolic representation of intruder messages and Maude's search capability to determine which messages, called *attack messages*, that an intruder can inject to drive the system to a bad state. Due to the finiteness of the FBs, applications either get stuck or are periodic. Thus, due to Maude's loop detection, the search is finite, as a search path is interrupted whenever a state that has been visited is re-visited.

Deploying an application is a theory transformation [13]. The function `deployApp` takes an application and a deployment mapping from FBs to devices and returns a system that is the deployed version of the application corresponding to the mapping.

From the enumerated attack messages, we derive which flows between function blocks need to have their events signed. Finally, from these flows, we are able to derive the security integrity wrapper policies for a given mapping of function blocks to devices.

Notice that we are able to capture multi-stage attacks, where the system is moved to multiple states before reaching a bad state. This is done by using stronger intruders that can use a greater number of messages.

Challenges To achieve our goal, we encounter a number of challenges.

– **Challenge 1 (Deployment Agnostic):** As pointed out above, the deployment of FBs on devices can affect the security requirements of flows. Analysis at the system level is more complex than at the application level. Thus it is important to understand how analysis on the application level can be transferred to the system level.
– **Challenge 2 (Symbolic Intruder):** Our intruder possess a set of concrete messages and a bound n on the number of injections. The search space grows rapidly with the bound. To reduce the search space, the concrete messages and bound n is replaced by n distinct symbolic messages. The symbols are instantiated only when required. The question is whether the symbolic model is equivalent to the concrete model.
– **Challenge 3 (Complete Set of Attack Messages):** Given an intruder, how do we know that at the end the set of attack messages found is a complete set for any deployment?
– **Challenge 4 (System Security by Wrapping):** How do we know that the wrappers constructed from identified flows and deployment mapping ensure the security of the system assuming our threat model?

To address these challenges, we prove the following theorems:

Symbolic Intruder Theorem (Theorem 1) states that each execution of an application `A` in a symbolic intruder environment has a corresponding execution of `A` in the concrete intruder environment with the same bound, and conversely. Thus, using the symbolic intruder is sound and complete with respect to the concrete intruder–enumeration of attacks gives the same result in both cases. The key to this result is the soundness and completeness of the symbolic match generation.

Deployment Theorem (Theorem 2) states that executions of an application `A` and a deployment `S` of `A` are in close correspondence. In particular the underlying function block transitions are the same and thus properties that depend only on function block states are preserved.

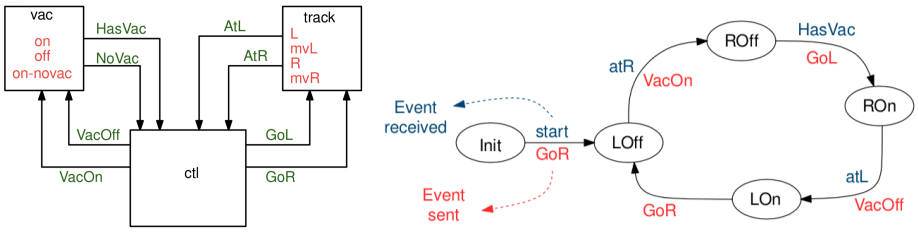


Fig. 2. PnP Function Blocks, ctl, vac, and track. The internal states of vac and track are shown in their corresponding boxes and their transitions are elided. The complete specification is shown in the finite machine to the right.

System Intruder Theorem (Theorem 3) states that, letting `A`, `S` be as in the Deployment Theorem, (1) for any execution of `S` in an intruder environment there is a corresponding execution of `A` in that environment; and (2) for any execution of `A` in an intruder environment that does not deliver any intruder messages that should flow on links internal to some device, has a corresponding execution from `S` in that environment. Corresponding executions preserve attacks and FB properties.

Wrapper Theorem (Theorem 4) Let `A` be an application, `S` a deployment of `A`, and `emsgs` a set of messages containing the attack messages enumerated by symbolic search with an n bounded intruder. The wrapper theorem says that the wrapped system `wrap(S,emsgs)` is resistant to attacks by an n bounded intruder.

2.1 Example

Consider an I4.0 unit, called Pick and Place (PnP),[4] used to place a cap on a cylinder. The cylinder moving on the conveyor belt is stopped by the PnP at the correct location. Then an arm picks a cap from the cap repository, by using a suction mechanism that generates a vacuum between the arm gripper and the cap. The arm is then moved, so that the cap is over the cylinder and then placed on the cylinder. Finally, the cylinder with the cap moves to the next factory element, e.g., storage element.

An application implementing the PnP logic has three function blocks that communicate using the channels as shown in Figure 2. The controller, ctl, coordinates with the vac and track function blocks as specified by the finite machine in Figure 2. For example, after starting, it sends the message GoR to the Arm that then moves to the right-most position where the caps are to be picked. When the Arm reaches this position, it informs the controller by sending the message atR. The controller then sends the message VacOn to the vac function block that starts its vacuum mechanism. If a vacuum is formed indicating that a cap has been picked, vac sends the message onhasVac to the ctl. The controller then sends GoL to the track that then moves to the left-most position where the cylinder is located on which the cap has to be placed. The

[4] See https://www.youtube.com/watch?v=Tkcv-mbhYqk starting at time 55 seconds for a very small scale version of the PnP.

track sends the message atL. The controller then sends the message VacOff to the vac to turn off the vacuum mechanism causing the cap to be placed over the cylinder.

For larger scale PnP, the hazard "Unintended Release of Cap" is catastrophic as the cap can hurt someone that is near the PnP. By performing analysis, such as STPA (Systems-Theoretic Process Analysis), one can determine that this event can occur when *The track function block is at state mvL and the vac function block is in state on-noVac or in state off.* This is because when starting to move to the position to the left, the gripper may have succeeded to grab a cap. However, while the arm is moving, the vacuum may have been lost causing the cap to be released, *i.e.*, the vac function block is in state on-noVac or off.

Following our methodology, shown in Figure 1, we feed to our Symbolic Model-Checker the PnP function blocks, its bad state above, and a symbolic intruder that can inject at most one message. One can specify stronger intruders, but this weak intruder is already able to lead the system into a bad state. Indeed, from the model-checker's output, we can compute that there are four different attack messages. One of them is shown in Figure 1, where the intruder impersonates the track and sends to the ctl a message atL.

From the identified attack messages we can see that messages in the flow from the track to the ctl involving the message atL should be protected.

Finally, suppose track and ctl are deployed in dev_1 and dev_2, respectively, then the computed security integrity wrapper on dev_1 will sign atL messages, and the security integrity wrapper on dev_2 will check whether atL messages are signed by dev_1. If track and ctl are deployed on the same device, there is no need to sign atL messages as we trust devices to protect internal communications.

3 Formalization of the I4.0 framework in Maude

We now describe the formal representation of applications, and the deployment and wrapping transformations. We formalize theorems.[5] We describe the main structures, operations, and rules using snippets from the Maude specification. Examples come from the Maude formalization of the PnP application of Section 2.

3.1 Function blocks

An I4.0 application is composed of a set of interconnected interactive finite state machines called function blocks. A function block is characterized by its finite set of states, finite sets of inputs and outputs, a finite set of possible events at each input or output, and a finite set of transitions. We call this a class and we give FBs both an instance and a class identifier to allow for multiple occurrences of a given class.

The Maude representation of a function block (FB) is a term of the form `[fbId : fbCid | fbAttrs]`, where `fbId` is the FB identifier, `fbCid` its class identifier and `fbAttrs` is a set of attribute-value pairs, including `(state : st)`, `(oEvEffs : oeffs)`, and `(ticked : b)`, with `state`, `oEvEffs`, `ticked` being the attribute tags, `st` the current state, `oeffs` a set of signals/events to be transmitted, and `b` a boolean indicating whether the FB has fired a transition in the current cycle.

[5] Proofs can be found in the technical report [14].

A transition is a term of the form `tr(st0,st1,cond,oeffs)` where `st0` is the initial state and `st1` the final state, `cond` is the condition, and `oeffs` is the set of outputs. A condition is a boolean combination of primitive conditions `(in is ev)` specifying a particular event (`ev`) at input `in`. A transition `tr(st0,st1,cond,oeffs)` is enabled by a set of inputs if they satisfy `cond` and the current state of the function block state `st0`. In this case, the transition can fire, changing the function block state to `st1` and emitting `oeffs`.

Example FB. The initial state, `vacInit(id("vac"))`, of an FB with class `vac` and identifier `id("vac")` is

```
[id("vac") : vac | state : st("off") ; ticked : false ;
                   iEvEffs : none ; oEvEffs : none]
```

The function `trsFB(fbCid)` returns the set of transitions for function blocks of class `fbCid`. For example `trsFB(vac, st("off"))` returns three transitions including

```
tr(st("on"), st("off"), inEv("VacOff") is ev("VacOff"),
   outEv("NoVac") :~ ev("NoVac"))
```

We compile a transition condition into a representation as a set of constraint sets which simplifies satisfaction checking, and matching when messages are symbolic. We can think of a constraint set (CSet) as a finite map from function block inputs to finite sets of events. A set of inputs $\texttt{ieffs} = \{(\texttt{in}_i \rhd \texttt{ev}_i) | 1 \leq i \leq k\}$ satisfies a CSet, `css`, just if `css` has size k, the $\texttt{in}_i$ form a set equal to the domain of `css`, and $\texttt{ev}_i$ is in $\texttt{css}(in_i)$ for $1 \leq i \leq k$. The function `condToCSet(cond)` returns the set of CSets such that an input set satisfies some CSet in the result just if it satisfies `cond`. This is lifted to transitions by the function `tr2symtr(tr(st1,st2,cond,oeffs) = symtr(st1,st2,condToCSet(cond),oeffs)`. For the `vac` example the CSet `condToCSet(inEv("VacOn") is ev("VacOn"))` maps `inEv("VacOn")` to the singleton `ev("VacOn")`.

3.2 Application structure and semantics

An application term has the form `[appId | appAttrs]`. Here `appAttrs` is a set of attribute-value pairs including `(fbs : funBs)` and `(iEMsgs : emsgs)`, where `funBs` is a set of function blocks (with unique identifiers), and `emsgs` is the set of incoming messages of the form `{{fbId,in},ev}`.

We use `fbId`, `fbId0` ... for FB identifiers, `in/out` for FB input/output connections, and `ev` for the event transmitted by a message. Terms of the form `{fbId,in/out}` are called Ports. For entities X with attributes, we write X.tag for the value of the attribute of X with name 'tag'.

The initial state of the PickNPlace (PnP) application described in Section 2 is

```
[id("pnp") | fbs : (ctlInit(id("ctl")
                trackInit(id("track")) vacInit(id("vac"))) ;
  iEMsgs : {{id("ctl"),inEv("start")},ev("start")} ;
  oEMsgs : none ; ssbs : none]
```

where the message `{{id("ctl"),inEv("start")},ev("start")}` starts the application controller.

Links of the form `{{fbId0,out},{fbId1,in}}` connect output ports of one FB to inputs of another possibly the same FB. They also connect application level inputs to FB inputs and FB external outputs to application level outputs. In a well formed application, each FB input has exactly one incoming link. In principle the link set is an attribute of the application structure. In practice, since it models fixed 'wires' connecting function block outputs and inputs and does not change, to avoid redundant information in traces, we specify a function `appLinks(appId)` which is defined in application specific scenario modules.

As an example, here are the two links that connect `vac` outputs to controller inputs.

```
{{id("vac"),outEv("NoVac")}, {id("ctl"),inEv("NoVac")}}
{{id("vac"),outEv("HasVac")},{id("ctl"),inEv("HasVac")}}
```

Application Execution Rules. There are two execution rules for application behavior and two rules modeling bounded intruder actions, one for the concrete case and one for the symbolic case. To ensure that an FB fires at most one transition per cycle, each FB is given a boolean `ticked` attribute, initially `false`, which is set to `true` when a transition fires, and reset to `false` when the outputs are collected.

The rule `[app-exe1]` fires an enabled function block transition and sets the ticked attribute to `true`.

```
crl[app-exe1]: [appId | fbs : ([fbId : fbCid | (state : st) ;
     (ticked : false) ; oEvEffs : none ;  fbAttrs] fbs1) ;
     iEMsgs : (emsgs0 iemsgs) ; ssbs : ssbs0 ;  appAttrs ]
=> [appId | fbs : ([fbId : fbCid | (state : st1) ;
     (ticked : true) ; oEvEffs : oeffs ;  fbAttrs] fbs1) ;
     iEMsgs : iemsgs) ; ssbs : (ssbs0 ssbs1) ;  appAttrs ]
 if symtr(st,st1,[css] csss,oeffs) symtrs := symtrsFB(fbCid,st)
 /\ size(emsgs0) = size(css)
 /\({ssbs1} ssbss) := genSol1(fbId,emsgs0,css) .
```

The function `genSol1(fbId,emsgs0,css)` returns a set of substitutions, consisting of all and only substitutions that match `emsgs0` to a solution of the CSet, `css`. In the case of concrete messages, *i.e.*, not containing symbols, the function `genSol1` just returns an empty substitution if `emsgs0` satisfies `css`. When `[app-exe1]` is no longer applicable (`hasSol(fbs,iemsgs)` is `false`), `[app-exe2]` collects and routes generated output and prepares for the next cycle.

```
crl[app-exe2]: [appId | fbs : fbs ; iEMsgs : iemsgs ;
                        oEMsgs : oemsgs ; ssbs : ssbs ; attrs]
=> [appId | fbs : fbs2 ; iEMsgs : emsgs0 ;
            oEMsgs : (oemsgs emsgs1) ; ssbs : ssbs  ; attrs1]
 if not hasSol(fbs,iemsgs)
 /\ tick := notApp(attrs)
 /\ not getTicked(attrs)  --- avoid extracting when no trans
 /\ attrs1 := setTicked(attrs, true)
 /\ {fbs2,emsgs0,emsgs1} :=
    extractOutMsgs(tick,fbs,none, none,none,appLinks(appId)) .
```

The function extractOutMsgs removes outputs from the function blocks that fired and routes them using appLinks(appId) to the linked FB input or application output. Application level inputs are accumulated in emsgs0 and outputs are accumulated in emsgs1. The ticked attribute of each FB is set to the value of tick. In the case of a basic application, this will be false indicating the FB is ready for the next cycle. When the application level execution rules are used in a larger context, (notApp(attrs) is true), extractOutMsgs ensures that each FBs ticked attribute is true, allowing further message processing before repeating the execution cycle. If the application has a ticked attribute, it is set to true, to indicate it has completed the current cycle. fbs2 collects the updated function blocks.

3.3 Intruders

An application A in the context of an intruder is represented in the concrete case by a term of the form [A, emsgs, n] where emsgs is a set of specific messages (typically all the messages that could be delivered) and n is the number of injection actions remaining. The rule [app-intruder-c] (omitted) selects one of the candidate messages, injects it, and decrements the counter.
An application A in the context of a symbolic intruder is represented by a structure of the form [A, smsgs] where smsgs are symbolic intruder messages of the form {{idSym,inSym},evSym} (idSym, inSym, evSym are symbols standing for function block identifiers, inputs, and events respectively). We require different messages to have distinct symbols. The rule [app-intruder] selects one of the intruder messages, and moves it from the intruder message set to the incoming messages iEMsgs.

```
rl[app-intruder]:
[[appId | fbs : fbs ; iEMsgs : emsgs0 ;  attrs], emsg emsgs]
=>
[[appId | fbs : fbs ; iEMsgs : (emsgs0 emsg) ; attrs], emsgs] .
```

We note that this rule works equally well with concrete or symbolic messages, allowing one to explore consequences of injecting specific messages. Using genSol1, a symbolic message can be instantiated to any deliverable message. Also, if a message is injected after all function blocks have been ticked and before [app-exe2] is applied, it will be dropped by [app-exe2], since function block inputs are cleared before collecting the next round of inputs.

3.4 The Intruder Theorem

We define the following correspondence between symbolic and concrete intruder states: [A,smsgs] $\sim$ [A,cmsgs,n] holds only if size(smsgs) $= n$, As.fbs $=$ Ac.fbs, and (As.iEMsgs)[ssbs] $=$ Ac.iEMsgs for some symbol substitution ssbs.[6] Two rule instances correspond if they are instances of the same rule. Also, in the [app-exe1] case the instances are the same transition of FBs with the same identifier, and in the [app-exe2] case the instances collect the same outputs.

An execution trace is an alternating sequence of (application) states and rule instances connecting adjacent states as usual. A symbolic trace TrS from [A,smsgs]

[6] Note that the attributes ssbs and oEMsgs do not affect rule application.

and a concrete trace TrC from `[A,emsgs,n]` correspond just if they have the same length and the i^{th} elements correspond as defined above.

Theorem 1. Let `[A,smsgs]` $\sim$ `[A,cmsgs,n]` be corresponding initial application states in symbolic and concrete intruder environments respectively, where no intruder messages have been injected.

If TrS is an execution trace from `[A,smsgs]` then there is a corresponding execution trace TrC starting with `[A,cmsgs, n]` and conversely.

Corollary 1. Search using the symbolic intruder model for paths reaching a `badState` finds all successful (bounded intruder) attacks.

We define the function `getBadEMsgs([A,smsgs])` that returns the set of injected message sets that lead to `badState`. This function uses reflection to enumerate search paths reflecting the command

```
search [A,smsgs] =>+ appInt:AppIntruder
        such that badState(appInt:AppIntruder) .
```

Injected symbolic messages are determined by looking for adjacent states where the symbolic message set decreases. The symbols of injected messages that were actually delivered are in the domain of the value of the `sbss` attribute of the final state.

In the PnP application for an intruder with a single message, `getBadEMsgs` returns four attack message sets, including the message set
`{{{id("ctl"),inEv("HasVac")},ev("HasVac")}}`.

Recall from Section 2 that the PnP application state satisfies `badState` if the `track` FB is in state `st("mvL")`, presumably carrying something from right to left, and the `vac` FB is in an *off* state (`st("on-novac")` or `st("off")`).

3.5 Deploying an Application

Once an application is designed, the next step is determining how to deploy FBs on devices. We model deployment as a theory transformation, introducing a data structure to represent deployed applications, called *System*s, extending the application module with rules to model system level communication elements, and defining a function mapping applications to their deployment given an assignment of FBs to host devices.

A deployed application is represented in Maude by terms of the form: `[sysId | appId | sysAttrs]` where `sysAttrs` is a set of attribute-value pairs including `(devs : devs)` and `(iMsgs : msgs)`. `devs` is a set of devices, and `msgs` is a set of system level messages of the form `{srcPort,tgtPort,ev}` where `srcPort`/`tgtPort` are terms of the form `{devId, {fbId, out/in}}`.

A device is represented as an application term with additional attributes including `(ticked : b)` which indicates whether all FBs have had a chance to execute. The function blocks of the application named by `appId` are distributed amongst the devices. The function `sysMap(sysId)` maps each FB identifier to the identifier of the device where the FB is hosted. Each device has incoming/outgoing ports corresponding to links between its function blocks and function blocks on other devices.

The function `deployApp(sysId,A,sysMap(sysId))` produces the deployment of application `A` as a system with identifier `sysId` and FBs distributed to devices according to `sysMap(sysId)`.

```
ceq deployApp(sysId,app,idmap) =
       mkSys(sysId,getId(app),devs,msgs)
if emsgs := getIEMsgs(app)
/\ devs := deployFBs(getFBs(app),none,idmap)
/\ msgs := emsgs2imsgs(sysId,emsgs,idmap,none) .
```

The real work is done by the function `deployFBs(fbs,none,idmap)` which creates an empty device for each device identifier in the range of `idmap` (setting `iMsgs` to `none` and `ticked` to `true`). Then each FB (identifier `fbId`) of `app` is added to the `fbs` attribute of the device identified by `idmap[fbId]`.

Note that the `deployApp` function can be applied to any state $\mathtt{A}_k$ in an execution trace from `A`. A system $\mathtt{S}_k$ can be abstracted to an application by collecting all the device FBs in the application `fbs` attribute, collecting the `iEMsgs` attributes of devices into the `iEMsgs` attribute of the application and adding system level input messages to the `iEMsgs` attribute of the application (after conversion to application level).

The execution rules for applications apply to devices in a system. There are two additional rules for system execution: `[sys-deliver]` and `[sys-collect]`. The rule `[sys-deliver]` delivers messages associated to the `iMsgs` attribute. The rule requires `isDone` to hold of the system devices, which means all the devices have their `ticked` attribute set to `true`. The target port of a system level message identifies the device and function block for delivery.

The rule `[sys-collect]` collects and distributes messages produced by the application level execution rules. It collects application level output messages from each device and converts them to system level output messages. Messages from device `iEMsgs` attributes are split into local and external. The local messages are left on the device, the external messages are converted to system level input messages.

We define a correspondence between execution traces from an application `A`, and a deployment `S = deployApp(sysId,A,idmap)` of that application. An application state `A1` corresponds to a system state `S1` just if they have the same function blocks and the same undelivered messages. (Note that the deployment and abstraction operations are subsets of this correspondence relation.) An instance of the `[app-exe1]` rule in an application trace corresponds to the same instance of that rule in a system trace (fires the same transition for the same function block). An instance of `[app-exe2]` in an application trace corresponds to a sequence `app-exe2+;sys-collect;sys-deliver` in a system trace collecting and delivering corresponding messages.

Theorem 2. Let `A` be an application and `S = deployApp(sysid,A,idmap)` be a deployment of `A`. Then `A` and `S` have corresponding executions.

Corollary 2. `A` and `S` as above satisfy the same properties that are based only on FB states and transitions. This is because corresponding traces have the same underlying function block transitions.

Intruders at the system level Deployed applications are embedded in an intruder environment analogously to applications. We consider a simple case where the intruder has a finite set of concrete messages to inject, using it to show that any attack at the system level can already be found at the application level. A system in a bounded intruder

environment is a term of the form `[sys,msgs]` where `sys` is a system as above, and `msgs` is a finite set of system level messages. The deployment function is lifted by

```
deployAppI(sysId,[A,emsgs],idmap) =
[deployApp[sysId,A,idmap],deployMsgs(emsgs,appLinks(A),idmap)]
```

where `deployMsgs` transforms application level messages `{fbport,ev}` to system level, `{srcdevport,tgtdevport,ev}` using the link and deployment maps.

The intruder injection rule, `[app-intruder]`, is lifted to `[sys-intruder]` and the correspondence relation of the deployment theorem is lifted in the natural way to the intruder case.

Theorem 3. Assume `Ai = [A,emsgs]` where `A` is an application in its initial state (no intruder messages injected) and `Si = deployAppI(sysId,Ai,idmap)`.

1. If TrS is a trace from `Si` then there is a corresponding trace from `Ai`.
2. If `TrA` is a trace from `Ai` that delivers no intruder messages that flow on links internal to a device, then there is a corresponding trace from `Si`.

Corollary 3. If a `badState` is reachable from `Si` then `sys2app(msgs)` is an element of `getBadEMsgs([A,smsgs])` where `size(smsgs) = size(msgs)`.

3.6 Wrapping

Towards the goal of signing only when necessary (Section 2) we define the transformation `wrapApp(A,smsgs,idmap)` of deployed applications as:

```
wrapSys(deployApp(sysId,A,idmap),flatten(getBadEMsgs([A,smsgs])))
```

where flatten unions the sets in a set of sets. `wrapSys(S,emsgs)` wraps the devices in `S` with policies for signing and checking signatures of messages on flows defined by `emsgs` as described below.

A wrapped device has input/output policy attributes `iPol`/`oPol` used to control the flow of messages in and out of the device. An input/output policy is an `iFact`/`oFact` set where an `iFact` has the form `[i : fbId ; in, devId]` and an `oFact` has the form `[o : fbId ; out]`. If `[i : fbId ; in, devId]` is in the input policy of a device then a message `{{fbId,in}, ev}` is accepted by that device only if `ev` is signed by `devId`, otherwise the message is dropped. Dually, if `[o : fbId ; out]` is in the output policy of a device, then when a message `{{fbId,out}, ev}` is transmitted `ev` is signed by the device. Following the usual logical representation of crypto functions, we represent a signed event by a term `sg(ev,devId)`, assuming that only the device with identifier `devId` can produce such a signature, and any device that knows the device identifier can check the signature.

The function `wrapSys(S,emsgs)` invokes the function `wrap-dev` to wrap each of its devices, `S.devs`. In addition to the device, the arguments of this function include the set of messages, `emsgs`, to protect, the application links and the deployment map. The links determine the sending FB, and the deployment determines the sending/receiving devices. If these are the same, no policy facts are added. Otherwise, policy facts are added so the sending device signs the message event and the receiving device checks for a signature according to the rules above.

Theorem 4. Assume `A` is an application, `allEMsgs` is the set of all messages deliverable in some execution of `A`, and `smsgs` is a set of symbolic messages of size n. Assume `badState` is not reachable in an execution of `A`, and `emsgs` contains `flatten(getBadEMsgs([A,smsgs]))`.

1. Let `wA = [wrapSys(deployApp(sysId,A,idmap),emsgs]`. Every execution from `wA` has a corresponding execution from `A` and conversely. In particular `badState` is not reachable from `wA`.
2. `badState` is not reachable from `wAC = [wrap(deploy(A,idmap),emsgs),allEMsgs,n]`

4 Related Work

There are a number of recent reports concerning the importance of cybersecurity for Industry 4.0. Two examples are the German Federal Office for Information Security (BSI) commissioned report on OPC UA security [7], and the ENISA study on good practices for IoT security [6]. OPC Unified Architecture (OPC UA) is a standard for networking for Industry 4.0 and includes functionality to secure communication. The BSI commissioned report describes a comprehensive analysis of security objectives and threats, and a detailed analysis of the OPC UA Specification. The analyses are informal but systematic, following established methods. A number of ambiguities and issues were found in this process. The ENISA report provides guidelines and security measures especially aimed at secure integration of IoT devices into systems. It includes a comprehensive review of resources on Industry 4.0 and IoT security, defines concepts, threat taxonomies and attack scenarios. Again, systematic but informal.

Although there is much work on modeling cyber physical systems and cyber-physical security (see [12] for recent review), much of it is based on simulation and testing. The formal modeling work focuses on general CPS and IoT not on the issues specific to I4.0 type situations. Lanotte *et al.* [10] propose a hybrid model of cyber and physical systems and associated models of cyber-physical attacks. Attacks are classified according to target device(s) and timing characteristics. Vulnerability to a given class is assessed based on the trace semantics. A measure of attack impact is proposed along with a means to quantify the chances of success. The proposed model is much more detailed than our model, considering device dynamics, and is focussed on traditional control systems rather than IoT in an Industry 4.0 setting. The attacks on devices modeled include our injection attacks. The Lanotte et. al. work is complementary to ours, while being more detailed we suspect our more abstract model combined with symbolic analysis is more scalable. The work in [17] relates to our work in proposing a method using model-checking to find all attacks on a system given possible attacker actions. The authors do not propose mitigations. SOTERIA [2] is a tool for evaluating safety and security of individual or collections of IoT applications. It uses model-checking to verify properties of abstract models of applications derived automatically from code (of suitable form). It requires access to the application source code.

The idea of using theory transformations to relate the application, system level specifications and reduce many reasoning problems to reasoning at the application level is

based on the notion of formal patterns reviewed in [13]. An early example of wrapping to achieve security guarantees is presented in [3] to mitigate DoS attacks.

5 Conclusions and Future Work

This paper presents a formal framework in rewriting logic for exploring I4.0 (smart factory) application designs and a bounded intruder model for security analysis. The framework provides functions for enumerating message injection attacks, and generating policies mitigating such attacks. It provides theory transformations from application specifications to specifications of systems with application components executing on devices, and for wrapping devices to protect against attacks using the generated policies. Theorems relating different specifications and showing preservation of key properties are given. We believe that formal executable models can be valuable to system designers to find corner cases and to explore tradeoffs in design options concerning the cost and benefits of security elements.

Future work includes theory transformations to refine the system level model to a network model with multiple subnets and switches, adding timing and modeling constraints induced by use of the TSN network protocol. As in our previous work [8], we are investigating the complexity of security properties given intruder models weaker than the traditional Dolev-Yao intruder [5]. We are also considering increasing the expressiveness of function block specifications to include time constraints as in [9] to automate the verification of properties based on time trace equivalence [16], such as privacy attacks. Finally, since these devices have limited resources, they may be subject to DDoS attacks. Symbolic verification can be used to check for such vulnerabilities [19].

Another important direction is developing theory transformations for correct-by-construction distributed execution [11]. This means accounting for real timing considerations and network protocols, and identifying conditions under which application and system level properties are preserved. An important use of the framework that we intend to investigate is relating safety and security analyses and connecting formal analyses to the engineering notations used for safety and security.

We are also currently extending our implementation to support the automated exploration of mappings of function blocks to devices. In particular, we are investigating the extension of [18] to take into account security objectives in addition to device performance limitations, device capabilities, and deadlines.

Acknowledgements This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830892. Talcott is partly supported by ONR grant N00014-15-1-2202 and NRL grant N0017317-1-G002. Nigam is partially supported by NRL grant N0017317-1-G002, and CNPq grant 303909/2018-8.

References

1. Cyberattack on a German steel-mill, 2016. Available at https://www.sentryo.net/cyberattack-on-a-german-steel-mill/.
2. Z. B. Celik, P. McDaniel, and G. Tan. SOTERIA: Automated IoT safety and security analysis. https://arxiv.org/pdf/1805.08876, 2018.

3. Rohit Chadha, Carl A. Gunter, José Meseguer, Ravinder Shankesi, and Mahesh Viswanathan. Modular preservation of safety properties by cookie-based DoS-protection wrappers. In *Proc. FMOODS 2008*, volume 5051 of *LNCS*, pages 39–58. Springer, 2008.
4. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.
5. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
6. ENSIA. Good practices for security of internet of things in the context of smart manufacturing, 2018.
7. Mr. Fiat and et.al. OPC UA security analysis, 2017.
8. Max I. Kanovich, Tajana Ban Kirigin, Vivek Nigam, and Andre Scedrov. Bounded memory Dolev-Yao adversaries in collaborative systems. *Inf. Comput.*, 238:233–261, 2014.
9. Max I. Kanovich, Tajana Ban Kirigin, Vivek Nigam, Andre Scedrov, and Carolyn L. Talcott. Time, computational complexity, and probability in the analysis of distance-bounding protocols. *Journal of Computer Security*, 25(6):585–630, 2017.
10. Ruggero Lanotte, Massimo Merro, Riccardo Muradore, and Luca Vigano. A formal approach to cyber-physical attacks. In *30th IEEE Computer Security Foundations Symposium*, pages 436–450. IEEE Computer Society, 2017.
11. Si Liu, Atul Sandur, José Meseguer, Peter Csaba Ölveczky, and Qi Wang. Automatic transformation of formal maude designs into correct-by-construction distributed implementations. Technical report, 2019.
12. Yuriy Zacchia Lun, Alessandro D'Innocenzo, Ivano Malavolta, and Maria Domenica Di Benedetto. Cyber-physical systems security: a systematic mapping study. *CoRR*, abs/1605.09641, 2016.
13. José Meseguer. Taming distributed system complexity through formal patterns. *Sci. Comput. Program.*, 83:3–34, 2014.
14. Vivek Nigam and Carolyn Talcott. Automated construction of security wrappers for industry 4.0 applications (long version). Available at https://github.com/SRI-CSL/WrapPat.git.
15. Vivek Nigam and Carolyn Talcott. Formal security verification of industry 4.0 applications. In *ETFA, Special Track on Cybersecurity in Industrial Control Systems*, 2019.
16. Vivek Nigam, Carolyn Talcott, and Abraão Aires Urquiza. Symbolic timed trace equivalence. In *Catherine Meadow's Festschirft*, 2019.
17. Farid Molazem Tabrizi and Karthik Pattabiraman. IOT: Formal security analysis of smart embedded systems. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 1–15. ACM, NY, 2016.
18. Tarik Terzimehic, Sebastian Voss, and Monika Wenger. Using design space exploration to calculate deployment configurations of IEC 61499-based systems. In *14th IEEE International Conference on Automation Science and Engineering*, pages 881–886, 2018.
19. Abraão Aires Urquiza, Musab A. AlTurki, Max I. Kanovich, Tajana Ban Kirigin, Vivek Nigam, Andre Scedrov, and Carolyn L. Talcott. Resource-bounded intruders in denial of service attacks. In *CSF*, pages 382–396, 2019.
20. L. H. Yoong, P. S. Roop, Z E Bhatti, and M. M. Y Kupz. *Model-Driven Design Using IEC 61499: A Synchronous Approach for Embedded Automation Systems*. Springer, 2015.

Maude-SE: a Tight Integration of Maude and SMT Solvers

Geunyeol Yu and Kyungmin Bae

Department of Computer Science and Engineering,
Pohang University of Science and Technology (POSTECH), Korea
{rgyen,kmbae}@postech.ac.kr

Abstract. This paper presents *Maude-SE*, an SMT extension of Maude that tightly integrates Maude and SMT solvers with extra functionality. In addition to the existing SMT solving capability of Maude, the tool provides three additional features that are not currently supported by Maude but that are very useful for rewriting modulo SMT: (i) building satisfying assignments by SMT solving, (ii) simplifying formulas using the underlying SMT solver, and (iii) dealing with non-linear arithmetic formulas. Hence, Maude-SE can analyze nontrivial systems that cannot be dealt with by the previous Maude-SMT implementation.

1 Introduction

Rewriting techniques are combined with satisfiability modulo theories (SMT) to model and analyze infinite-state concurrent systems. In *rewriting modulo SMT* [1, 10], system states are symbolically represented as terms constrained by SMT formulas, and system transitions are specified using conditional rewrite rules between constrained terms. SMT solving and term rewriting are then combined to perform symbolic reachability analysis of infinite-state systems. The Maude tool [5] provides the basic functionality of SMT solving for linear integer and real arithmetic. It is implemented using a *Maude-SMT* wrapper interface to connect Maude to SMT solvers at the C++ level.

The current Maude-SMT implementation, however, has several limitations. First, Maude-SMT provides no capability to obtain a satisfying assignment to the variables in a satisfiable formula. This makes it difficult to construct concrete witnesses for symbolic reachability analysis within Maude. Second, Maude-SMT does not simplify SMT formulas, resulting in unnecessarily large constraints when analyzing complex systems, whereas SMT solvers can internally simplify SMT formulas for efficiency reasons. Finally, many state-of-the-art SMT solvers, including Yices2 [8] and Z3 [7], can handle formulas in non-linear arithmetic, but Maude-SMT can only deal with formulas in linear arithmetic.

To cope with these difficulties, we present a new SMT extension of Maude, called *Maude-SE*, that tightly integrates Maude and SMT solvers. Maude-SE extends the previous Maude-SMT interface to support all the existing SMT solving features of Maude. Moreover, Maude-SE provides extra functionality to

build satisfying assignments and to simplify SMT formulas by means of two Core Maude functions `smtCheck` and `simplifyFormula`, implemented at the C++ level. Maude-SE provides the `smt-search2` command for symbolic reachability analysis with concrete witnesses, implemented by extending Full Maude using the Core Maude functions.

In Maude-SE, we have integrated three widely used SMT solvers with Maude. Besides the existing connections to CVC4 [2] and Yices2, Maude-SE provides a new connection to Z3. Our implementation exploits the power of the SMT solver to check the satisfiability of non-linear arithmetic formulas. Even though the satisfiability of non-linear formulas is undecidable for integers and very costly for reals, all three SMT solvers implement non-linear arithmetic algorithms [9, 4]. Maude-SE allows us to symbolically analyze complex systems involving non-linear arithmetic, which was previously impossible using Maude. The tool, together with several examples, is available at `https://maude-se.github.io`.

The rest of the paper is organized as follows. Section 2 gives a background on rewriting modulo SMT. Section 3 presents the main functionality of Maude-SE for symbolic reachability analysis. Section 4 presents the Core Maude interface of Maude-SE, including two functions `smtCheck` and `simplifyFormula`. Section 5 explains the implementation of Maude-SE. Section 6 presents a simple example. Finally, Section 7 gives some concluding remarks.

2 Background: Rewriting Modulo SMT and Maude

An *order-sorted signature* is a triple $\Sigma = (S, \leq, F)$ with a finite poset of sorts $(S, \leq)$ and a set of function symbols F typed with sorts in $(S, \leq)$. The set $T_\Sigma(X)_s$ denotes the set of Σ-terms of sort s over a set of sorted variables X, and $T_{\Sigma,s}$ denotes the set of ground Σ-terms of sort s. A term t is called *linear* if and only if each variable $x \in var(t)$ occurs only once in the term t, where $var(t)$ denotes the set of variables in a term t.

A *built-in subtheory* $\mathcal{E}_0$ is a first-order theory that is handled by SMT solving. The built-in subsignature $\Sigma_0 = (S_0, F_0)$ of $\mathcal{E}_0$ is a a subsignature of Σ. The set $QF_{\Sigma_0}(X_0)$ denotes the set of quantifier-free Σ_0-formulas over $X_0 \subseteq X$ a set of built-in variables. The satisfiability of a formula $\phi \in QF_{\Sigma_0}(X_0)$ in $\mathcal{E}_0$ can be decided using the SMT solver. We require that any terms of built-in sorts are syntactically built-in terms (i.e., $T_\Sigma(X_0)_{s_0} = T_{\Sigma_0}(X_0)_{s_0}$ for each $s_0 \in \Sigma_0$).

A *rewrite theory modulo a built-in subtheory* $\mathcal{E}_0$ [10] is a triple $\mathcal{R} = (\Sigma, E, R)$, where: (i) Σ is an order-sorted signature with the built-in subsignature $\Sigma_0 \subseteq \Sigma$; (ii) (Σ, E) is an equational theory with E a set of Σ-equations; and (iii) R is a set of topmost rewrite rules of the form $l \longrightarrow r$ **if** ϕ, with $l, r \in T_\Sigma(X)_{State}$ for some sort *State*, and $\phi \in QF_{\Sigma_0}(X_0)$. To ensure that all rewrite rules take place at the top of the term, we assume that sort *State* is at the top of one of the connected component of poset of sorts, and no operator in Σ has *State* or any of its subsorts as an argument sort. We require that an equational theory (Σ, E) *protects* $\mathcal{E}_0$ (i.e., for built-in terms $t_1, t_2 \in \mathcal{T}_{\Sigma_0}$, $t_1 =_E t_2$ implies $\mathcal{E}_0 \models t_1 = t_2$, where $=_E$ is a congruence relation on Σ-terms induced by (Σ, E)) [1, 10].

In rewriting modulo SMT, *constrained terms* symbolically represent sets of system states. A constrained term is a pair $\{\phi, t\}$ of a constraint $\phi \in QF_{\Sigma_0}(X_0)$ and a state term $t \in T_\Sigma(X)_{State}$. Intuitively, $\{\phi, t\}$ represents the set of all ground instances ρt of the term t by a substitution ρ such that $\mathcal{E}_0 \models \rho\phi$ holds. A *one-step symbolic rewrite* $\{\phi_t, t\} \rightsquigarrow_{\mathcal{R}} \{\phi_u, u\}$ between constrained terms holds if and only if there exists a rule $l \longrightarrow r$ **if** ϕ and a substitution θ such that (i) $\theta l =_E t$, (ii) $\theta r =_E u$, (iii) $\mathcal{E}_0 \models \phi_u$, where $\mathcal{E}_0 \models (\phi_t \wedge \theta\phi) \leftrightarrow \phi_u$. The correctness of rewriting modulo SMT [10] guarantees that symbolic rewrites have corresponding concrete rewrites, and vice versa.

Maude [5] provides an interface, called *Maude-SMT* in this paper, to perform SMT solving and symbolic reachability analysis since version 3, using connections to Yices2 [8] and CVC4 [2]. Maude-SMT declares built-in signatures with three sorts `Boolean`, `Integer`, and `Real` for the theories of Booleans, integers, and reals, respectively, in the SMT-LIB standard [3]. The `check` command invokes the SMT solver to check the satisfiability of a given formula. The `smt-search` command performs symbolic reachability analysis using rewriting modulo SMT. Maude-SMT can only deal with linear arithmetic formulas, and the commands do not give satisfiability assignments by SMT solving.

3 Symbolic Reachability Analysis in Maude-SE

This section presents the main functionality of Maude-SE. Section 3.1 explains the admissibility requirements imposed on system modules. Section 3.2 presents the Full Maude interface of Maude-SE for symbolic reachability analysis with concrete witness. Section 3.3 describes two meta-level functions `metaSmtSearch2` and `metaSmtSearch2Path` for symbolic reachability analysis.

3.1 System specification

A system module M specifies a rewrite theory modulo a built-in subtheory in Maude-SE. There are several admissibility requirements imposed on M. First, M must specify a topmost rewrite theory with some top sort *State*. Second, M must *protect* the built-in subtheory. Third, each rewrite rule $l \longrightarrow r$ **if** ϕ in M must satisfy the following conditions: (i) $l, r \in T_\Sigma(X)_{State}$; (ii) each variable in $(var(r) \cup var(\phi)) \setminus var(l)$ has a built-in sort; and (iii) ϕ is a conjunction of SMT conditions of the form $u = v$ with $u, r \in T_{\Sigma_0}(X_0)$.

These admissibility requirements are relaxed than those for Maude-SMT. Originally, each rule is required to be left-linear with respect to SMT variables [10]. This is not a strict restriction, because any rewrite rule can be transformed into a conditional rule that is left-linear with respect to SMT variables [10]. Unlike Maude-SMT, this transformation is automated in Maude-SE; a system module can freely contains rules with duplicate SMT variables in the left-hand sides, and Maude-SE will automatically transform these rules using the theory transformation specified in [10] (see Section 5.1 for the details).

Example 1. The following system module specifies the Euclidean algorithm for the greatest common divisor. This module is slightly adapted from [6] to contain duplicate SMT variables in the left-hand side of the third rule.

```
mod GCD is
  protecting INTEGER .
  sort GcdResult .
  op gcd : Integer Integer -> GcdResult [ctor] .
  op return : Integer -> GcdResult [ctor] .

  vars X Y : Integer .

  crl [r1] : gcd(X, Y) => gcd(X - Y, Y) if (X > Y) = true .
  crl [r2] : gcd(X, Y) => gcd(X, Y - X) if (X < Y) = true .
  rl  [r3] : gcd(X, X) => return(X) .
endm
```

Example 2. There are two rewrite rules in the following module. For example, given real numbers r_1 and r_2, the rule `c1` rewrites `f(`r_1`,` r_2`)` into `f(`r_1`,` r_m`)` for some real number $r_1 < r_m < r_2$. Notice that both rules contain SMT variable `Z` that does not appear in the left-hand side of the rules.

```
mod RMID is
  protecting REAL .
  sort Calc .
  op f : Real Real -> Calc .

  vars X Y Z : Real .

  crl [c1] : f(X, Y) => f(X, Z)
   if (X < Z) = true /\ (Z < Y) = true [nonexec] .
  crl [c2] : f(X, Y) => f(Z, X)
   if (Y < Z) = true /\ (Z < X) = true [nonexec] .
endm
```

3.2 Full Maude Interface of Maude-SE

The Full Maude interface of Maude-SE provides the `smt-search2` command for symbolic reachability analysis. Given an initial state $t \in T_\Sigma(X_0)$, a goal pattern $u \in T_\Sigma(X)$, and a goal condition $\varphi \in QF_{\Sigma_0}(X_0)$, the following command searches for n states that are reachable within m rewrite steps from the initial state t, match the goal pattern u, and satisfy the goal condition φ:

`(smt-search2 [`n`,`m`] in` M `:` t `=>*` u `such that` φ `.)`

It is worth noting that t and φ can contain SMT variables of built-in sorts. Unlike the `smt-search` command available in Maude-SMT, the `smt-search2` command returns concrete states and the corresponding satisfying assignments, in addition to symbolic states and constraints.

Example 3. Consider the module `GCD` in Example 1. The following Full Maude command can find two integers X and Y with a sum of 27, where the greatest common divisor of X and Y is 3.

```
Maude> (smt-search2 [1] gcd(X, Y) =>* return(3)
        such that (X + Y === 27) = true .)

Solution 1
symbolic state: return(#V#10:Integer)
constraint:
    #V#0:Integer === #V#2:Integer and
    (#V#1:Integer === #V#3:Integer and
    (not #V#3:Integer <= #V#2:Integer and
    (#V#2:Integer === #V#4:Integer and
    (#V#3:Integer + -1 * #V#2:Integer === #V#5:Integer and
    (not #V#5:Integer <= #V#4:Integer and
    (#V#4:Integer === #V#6:Integer and
    (#V#5:Integer + -1 * #V#4:Integer === #V#7:Integer and
    (not #V#7:Integer <= #V#6:Integer and
    (#V#6:Integer === #V#8:Integer and
    (#V#7:Integer + -1 * #V#6:Integer === #V#9:Integer and
    (not #V#8:Integer <= #V#9:Integer and
    (#V#8:Integer + -1 * #V#9:Integer === #V#10:Integer and
    #V#9:Integer === #V#10:Integer))))))))))))
substitution:
    V0:Integer --> #V#10:Integer ;
    X:Integer --> #V#0:Integer ;
    Y:Integer --> #V#1:Integer
concrete state:
    return(3)
assignment:
    #V#0:Integer |--> 6
    #V#1:Integer |--> 21
    #V#2:Integer |--> 6
    #V#3:Integer |--> 21
    #V#4:Integer |--> 6
    #V#5:Integer |--> 15
    #V#6:Integer |--> 6
    #V#7:Integer |--> 9
    #V#8:Integer |--> 6
    #V#9:Integer |--> 3
    #V#10:Integer |--> 3
```

In the above result, the symbolic state and constraint give the constrained term specifying the set of solution states. The concrete state shows one concrete state in the (possibly infinite) set of solution states, and the assignment shows the satisfying assignment for the constraint to build the concrete state.

3.3 Metalevel Functions for Symbolic Reachability Analysis

There are two meta-level functions `metaSmtSearch2` and `metaSmtSearch2Path` in Maude-SE. Consider a module M, an initial state t, a goal pattern u, and a goal condition φ. When $\overline{M}$, $\overline{t}$, $\overline{u}$, and $\overline{\varphi}$ denote the metarepresentations of M, t, u, and φ, respectively, the function `metaSmtSearch2(`$\overline{M}$`,`$\overline{t}$`,`$\overline{u}$`,`$\overline{\varphi}$`,'*,`m`,`k`)` returns the $(k-1)$-th solution of the command

`(smt-search2 [`n`,`m`] in` M `:` t `=>*` u `such that` φ `.)`

where k is less than n. The function `metaSmtSearch2Path(`$\overline{M}$`,`$\overline{t}$`,`$\overline{u}$`,`$\overline{\varphi}$`,'*,`m`,`k`)` returns a symbolic path and its concrete witness for the solution of the same command, if it exists.

Example 4. Consider the `smt-search2` command for `GCD` in Example 3. We can run the same symbolic reachability analysis at the metalevel as follows.

```
Maude> red metaSmtSearch2(['GCD],
    upTerm(gcd(X, Y)),
    upTerm(return(3)),
    upTerm(X + Y === 27) = upTerm(true),
    '*, unbounded, 0) .

result SmtResult2:
{'return['#V#10:Integer],...
 'X:Integer <- '#V#0:Integer ;
 'Y:Integer <- '#V#1:Integer,
 ...,
 'return['3.Integer],
 '_`,_[ '_|->_['#V#0:Integer,'6.Integer],
        '_|->_['#V#1:Integer,'21.Integer],...]}
```

Using `metaSmtSearch2Path`, we can obtain a symbolic path—that represents the (possibly infinite) set of solution paths—and one concrete witness path.[1]

```
Maude> red metaSmtSearch2Path(['GCD],
    upTerm(gcd(X,Y)),
    upterm(return(3)),
    upTerm(X + Y === 27) = upTerm(true),
    '*, unbounded, 0) .

result Trace2:
{'gcd['#V#0:Integer,'#V#1:Integer],...,
 'gcd['6.Integer,'21.Integer], ...}
...
{'gcd['_-_['#V#8:Integer,'#V#9:Integer],'#V#9:Integer],...,
 'gcd['3.Integer,'3.Integer], ...}
```

[1] This feature can also be implemented in the Full Maude interface. However, the current version of Full Maude does not yet provide the ability to display search paths, and neither does Maude-SE.

4 Core Maude Interface of Maude-SE

Maude-SE provides Core Maude functions to connect Maude and SMT solvers. These functions can be used to directly ask SMT queries within Maude. As a matter of fact, the meta-level functions and the Full Maude interface explained in Section 3 are implemented using these functions. In Maude-SE, the predefined functional module `SMT-CHECK` declares the Core Maude functions, along with data structures for SMT expressions.

Unlike the Full Maude interface, SMT expressions are specified as ground terms in the Core Maude interface. In particular, SMT variables are declared using the following operators, where `SMTVarId` denotes variable identifiers.

```
op b : SMTVarId -> BooleanVar [ctor] .
op i : SMTVarId -> IntegerVar [ctor] .
op r : SMTVarId -> RealVar [ctor] .
```

In the Core Maude interface, SMT expressions with SMT variables have sorts `BooleanExpr`, `IntegerExpr`, or `RealExpr`, whereas SMT expressions *without* SMT variables have built-in sorts `Boolean`, `Integer`, or `Real`. This can be declared using the following subsort relations:

```
subsorts Boolean BooleanVar < BooleanExpr .
subsorts Integer IntegerVar < IntegerExpr .
subsorts Real RealVar < RealExpr .
```

Example 5. Let `x` be a constant of sort `SMTVarId`. The integer constant 1 has sort `Integer`, the variable term `i(x)` has sort `IntegerVar`, $1 + 2 * 3$ has sort `Integer`, and $1 + i(x) * 3$ has sort `IntegerExpr`.

The `smtCheck` function checks the satisfiability of a given Boolean expression. This function takes two arguments: an SMT formula φ and a flag b. The result of `smtCheck`(φ, b) is as follows. When φ is satisfiable, it returns *true* for $b = \mathit{false}$ and a satisfying assignment for $b = \mathit{true}$. When φ is unsatisfiable, `smtCheck` returns *false*. When the SMT solver cannot determine the satisfiability (e.g., in case of non-linear integer arithmetic), `smtCheck` returns *unknown*.

```
op smtCheck : BooleanExpr Bool -> SmtCheckResult .
op smtCheck : BooleanExpr -> SmtCheckResult .
eq smtCheck(BE:BooleanExpr) = smtCheck(BE:BooleanExpr, false) .

sorts SmtCheckResult .
subsort Bool < SmtCheckResult .
op unknown : -> SmtCheckResult [ctor] .
op {_} : SatAssignmentSet -> SmtCheckResult [ctor] .
```

Example 6. Given integer variables `i(x)` and `i(y)`, the following show the results for `smtCheck` of two formulas in nonlinear integer arithmetic.

```
Maude> red smtCheck (i(x) * i(y) > 2, true) .
result SmtCheckResult: {(i(x) |-> 1),i(y) |-> 3}

Maude> red smtCheck (i(x) * i(y) > 2 and i(x) === 0, true) .
result Bool: (false).Bool
```

The `simplifyFormula` function simplifies SMT expressions using the SMT solver. Given an SMT expression e, `simplifyFormula(`e`)` returns an equivalent expression obtained by simplifying e using the SMT solver. In particular, if e contains no SMT variable, `simplifyFormula` returns the value of e.

```
op simplifyFormula : BooleanExpr -> BooleanExpr .
op simplifyFormula : IntegerExpr -> IntegerExpr
op simplifyFormula : RealExpr -> RealExpr
```

Example 7. The following shows `simplifyFormula` for two expressions:

```
Maude> red simplifyFormula (1/1 * (1/1 * 1/1) + 1/1 * (9/1 * 1/1)) .
result Real: 10/1

Maude> red simplifyFormula (i(x) + 1 * 22 div 2) .
result IntegerExpr: 11 + i(x)
```

5 Implementation

5.1 Theory Transformation

We consider two theory transformations, introduced in [10], to implement the main functionality of Maude-SE in Section 3. Given a rewrite theory $\mathcal{R}$, we first obtain an equivalent rewrite theory $\mathcal{R}^\circ$ where each rule is left-linear with respect to SMT variables. We then build a rewrite theory $\widehat{\mathcal{R}^\circ}$ that explicitly specifies symbolic rewrites between constrained terms. These transformations are implemented using the Core Maude functions `smtCheck` and `simplifyFormula`.

By the transformation $\mathcal{R} \mapsto \mathcal{R}^\circ$, each rule $l \longrightarrow r$ **if** ϕ in $\mathcal{R}$ is transformed into the following rule that is left-linear with respect to SMT variables:

$$l^\circ \longrightarrow r \ \textbf{ if } \phi \wedge \textstyle\bigwedge_{x_0 \in \mathrm{dom}(\theta^\circ)} x_0 = \theta^\circ x_0$$

where $l = \theta^\circ l^\circ$ for $l^\circ \in T_{\Sigma \setminus \Sigma_0}(X)$ and substitution $\theta^\circ : X_0 \to T_{\Sigma_0}(X_0)$, and l° is linear with respect to SMT variables. The pair (l°, θ°) is called an *abstraction of built-ins* for l [10]. Each subterm u of built-in sort in l is replaced by a distinct variable x_u in l° in such a way that $u = \theta^\circ x_u$.

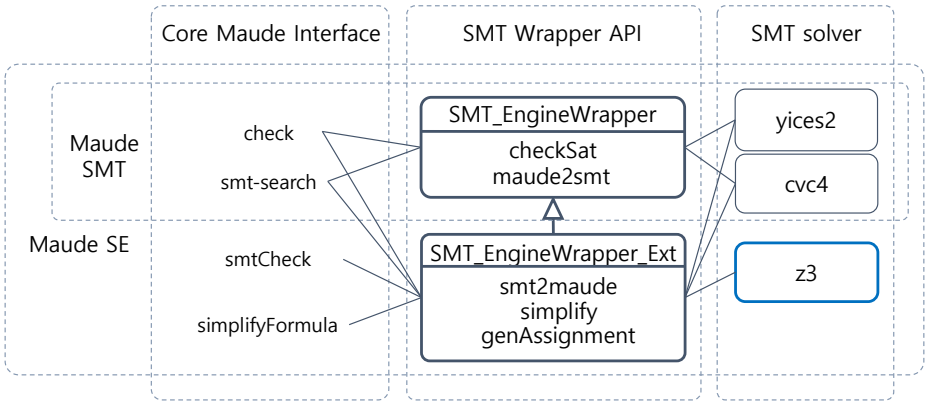


Fig. 1. The design of the Maude-SE implementation.

By the transformation $\mathcal{R}^\circ \mapsto \widehat{\mathcal{R}^\circ}$, each rewrite rule $l^\circ \longrightarrow r$ **if** ϕ in $\mathcal{R}^\circ$ is transformed into the following rule that expresses one-step symbolic rewrites:

$$\{B, l^\circ, N\} \longrightarrow \{B', r, N+m\} \text{ if } \bigwedge_{i=1,\ldots,m} Y_i := \texttt{newVar}(N+i) \\ \wedge\ B' := \texttt{simplifyFormula}(B \wedge \phi) \\ \wedge\ \texttt{smtCheck}(B', \mathit{false})$$

such that: (i) B and B' are Maude variables of sort `BooleanExpr` to denote accumulated constraints; (ii) N is a Maude variable that represents a counter for creating a new SMT variable; (iii) $\{Y_1, \ldots, Y_m\} = (var(r) \cup var(\phi)) \setminus var(l)$; and (iv) `newVar(`$n$`)` returns a fresh SMT variable having one of the forms `b(`n`)`, `i(`n`)`, and `r(`n`)`. By the matching condition $Y_i :=$ `newVar(`$N+i$`)`, each variable Y_i appearing in r and ϕ but not appearing in l is instantiated by a fresh SMT variable term. To declare such transformed rules, $\widehat{\mathcal{R}^\circ}$ declares new operators such as `{_,_,_}` and `newVar`, and imports the predefined module `SMT-CHECK`.

5.2 C++ Implementation

Maude-SE's Core Maude interface has been implemented at the C++ level. Figure 1 shows the design of our Maude-SE implementation. We define an interface `SMT_EngineWrapper_Ext` for Maude-SE, which extends the existing interface for Maude-SMT. There are two methods in `SMT_EngineWrapper` for Maude-SMT: `checkSat` invokes the SMT solver to check the satisfiability of a formula, and `maude2smt` converts Maude terms into the data structures for the SMT solver. The interface `SMT_EngineWrapper_Ext` has three additional methods: `simplify` calls the SMT solver to simplify a formula, `genAssignment` invokes the SMT solver to obtain satisfying assignments, and `smt2maude` builds Maude terms from the solver's formula data structures.

The Core Maude functions explained in Section 4 use these API methods to interact with the underlying SMT solver. For example, Maude-SMT's

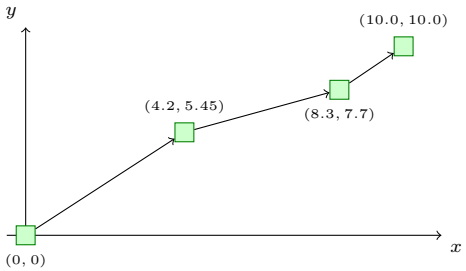


Fig. 2. A simple robot

`check` command calls `checkSat` to check the satisfiability of a formula, where the input to the SMT solver is obtained from the Maude representation using `maude2smt`. Similarly, Maude-SE's `smtCheck` function calls the same methods `checkSat` and `maude2smt`. When the second argument is *true*, `smtCheck` also calls `genAssignment` to obtain a satisfying assignment and uses `smt2maude` to build a Maude term for the satisfying assignment. The method `maude2smt` in `SMT_EngineWrapper_Ext` *overrides* the same method in `SMT_EngineWrapper` to deal with SMT variable terms, such as `b(`n`)`, `i(`n`)`, and `r(`n`)`.

We have implemented connections to three widely used SMT solvers, namely, CVC4, Yices2, and Z3. Because the wrapper interface of Maude-SE extends one of Maude-SMT, all the features of Maude-SMT, such as the Core Maude commands `check` and `smt-search`, are also available in Maude-SE. For CVC4 and Yices2, our implementation is adapted from the existing implementation of Maude-SMT to support non-linear arithmetic. Specifically, Maude-SMT's Yices2 connector does not support non-linear arithmetic, although Yices2 implements the MCSAT algorithm for non-linear arithmetic in [8]. A connection to Z3 is not previously available in Maude-SMT, and so we have implemented a new Z3 connection for Maude-SE that also supports non-linear arithmetic.

6 An Example

This section presents a simple robot system and its symbolic reachability analysis using Maude-SE. As shown in Fig. 2, a robot moves in a two-dimensional space. The position $\boldsymbol{x}$ of the robot changes according to its velocity $\boldsymbol{v}$ and acceleration $\boldsymbol{a}$, given by the following ordinary differential equation (ODE):

$$\dot{\boldsymbol{x}} = \boldsymbol{v}, \qquad \dot{\boldsymbol{v}} = \boldsymbol{a}$$

The solution of the above ODEs is given by the functions $\boldsymbol{v}(\tau) = \boldsymbol{v}_0 + \boldsymbol{a}\tau$ and $\boldsymbol{x}(\tau) = \frac{1}{2}\boldsymbol{a}\tau^2 + \boldsymbol{v}_0\tau + \boldsymbol{x}_0$ for time τ. We assume that the robot can only have non-zero acceleration in either the horizontal or vertical direction at a time.

In our Maude specification, a robot is modeled as an object with the following attributes: position (`pos`), velocity (`vel`), acceleration (`acc`), trace, and clock.

The clock attribute denotes the time elapsed from the beginning. The trace attribute denotes the history of the robot's behavior, given by a sequence of velocities and positions.

```
class Robot | pos : Vector, acc : Vector,
              vel : Vector, clock : Real, trace : RobotTrace .
```

For example, the following object oriented term represents a robot with postion $(0, 1)$, velocity $(1, 2)$, acceleration $(2, -1)$, and clock 2.

```
< O : Robot | pos : [0/1, 1/1], vel : [1/1, 2/1],
              acc : [2/1, -1/1], clock : 2/1,
            trace : [0/0, 2/1] ==([2/7, 5/2] : 2/1)==> [10/3, 10/7] >
```

There are three rewrite rules to specify the robot's behavior. The *move* rule represents the change of the robot's state according to its physical dynamics for time τ.

```
crl [move]:
    < O : Robot | pos : [PX, PY], vel : [VX, VY],
                  acc : [AX, AY], clock : T, trace : TRACE >
 => < O : Robot | pos : [PX', PY'], vel : [VX', VY'],
                clock : T + TAU,
                trace : TRACE ==([VX', VY'] : T + TAU)==> [PX', PY'] >
 if VX' = VX + AX * TAU
 /\ VY' = VY + AY * TAU
 /\ PX' = 1/2 * AX * TAU * TAU + VX * TAU + PX
 /\ PY' = 1/2 * AY * TAU * TAU + VY * TAU + PY
 /\ TAU >= toReal(0) = true [nonexec] .
```

The `accX` and `accY` rules change the accelerations of the robot in the horizontal or vertical direction. For example, the `accX` rule changes the horizontal acceleration `AX` to some arbitrary acceleration `AX'`, provided that `AY` is zero.

```
crl [accX]:
    { < O : C:Robot | acc : [AX, AY] > }
 => { < O : C:Robot | acc : [AX', AY] > }
 if AY = toReal(0) .

crl [accY]:
    { < O : C:Robot | acc : [AX, AY] > }
 => { < O : C:Robot | acc : [AX, AY'] > }
 if AX = toReal(0) .
```

The following `smt-search2` command checks whether the robot can reach the goal position $(10, 10)$ from the initial position $(0, 0)$, where the initial velocity and acceleration are both $(0, 0)$.

```
Maude> (smt-search2 [1]
    < r : Robot | pos : [0/1, 0/1], vel : [0/1, 0/1],
                  acc : [0/1, 0/1], clock : 0/1, trace : [0/1, 0/1]> =>*
    < r : Robot | pos : [10/1, 10/1] > .)

Solution 1
symbolic state:
<   r : Robot |
  pos : [#V#8:Real, #V#9:Real], vel : [#V#11:Real, #V#12:Real],
  acc : [#V#6:Real, #V#7:Real], clock : (0/1 + #V#3:Real + #V#10:Real),
  trace :
       [0/1,0/1]
    ==([#V#4:Real, #V#5:Real] : 0/1 + #V#3:Real)==>
       [#V#1:Real, #V#2:Real]
    ==([#V#11:Real,#V#12:Real] : 0/1 + #V#3:Real + #V#10:Real)==>
       [#V#8:Real, #V#9:Real]>
constraint:
    #V#4:Real === #V#0:Real * #V#3:Real and
    (#V#5:Real === 0/1 and (#V#1:Real === 1/2 *
    (#V#0:Real * (#V#3:Real * #V#3:Real)) and
    (#V#2:Real === 0/1 and (#V#3:Real >= 0/1 and (#V#6:Real === 0/1 and
    (#V#11:Real === #V#4:Real + #V#6:Real * #V#10:Real and
    (#V#12:Real === #V#5:Real + #V#7:Real * #V#10:Real and
    (#V#8:Real === 1/2 * (#V#6:Real *(#V#10:Real * #V#10:Real)) +
    (#V#4:Real * #V#10:Real + #V#1:Real) and
    (#V#9:Real === 1/2 * (#V#7:Real * (#V#10:Real * #V#10:Real)) +
    (#V#5:Real * #V#10:Real + #V#2:Real) and #V#10:Real >= 0/1))))))))
substitution:
    V0:Real --> #V#8:Real ;
    V1:Real --> #V#9:Real
    ...
concrete state:
    < r : Robot | pos : [10/1, 10/1], vel : [20/1, 20/1],
    acc : [0/1, 0/1], clock : 1/1,
  trace : ([0/1, 0/1] ==([20/1,20/1] : 1/1)==> [10/1,10/1]) >
assignment:
    #V#0:Real |--> 20/3
    #V#1:Real |--> 10/3
    #V#2:Real |--> 0/1
    #V#3:Real |--> 1/1
    #V#4:Real |--> 20/3
    #V#5:Real |--> 0/1
    #V#6:Real |--> 0/1
    #V#7:Real |--> 20/1
    #V#8:Real |--> 10/1
    #V#9:Real |--> 10/1
    #V#10:Real |--> 1/1
    #V#11:Real |--> 20/3
    #V#12:Real |--> 20/1
```

The concrete state of Solution 1 shows that the robot arrives at $(10, 10)$ by the strategy in the trace. This concrete state represents one of the (possibly infinite) set of the solution states, given by the symbolic state and constraint.

7 Conclusion

We have developed Maude-SE, an extension of Maude for symbolic reachability analysis using rewriting modulo SMT. Maude-SE provides a new Full Maude command `smt-search2` for symbolic reachability analysis, which can also show concrete witnesses, besides all the functionalities of the existing Maude-SMT interface. For this purpose, we have implemented two Core Maude functions at the C++ level: `smtCheck` for obtaining satisfying assignments from the SMT solver, and `simplifyFormula` for simplifying SMT expressions using the SMT solver. We have implemented a new Z3 connection for Maude-SE, in addition to the existing connections to Yices2 and CVC4. Our implementation of these connectors supports nonlinear arithmetic formulas, which cannot be handled by the previous Maude-SMT implementation. We have demonstrated the power of Maude-SE using symbolic reachability analysis of a robot with polynomial dynamics, which is beyond the capability of the previous Maude-SMT.

References

1. Bae, K., Rocha, C.: Symbolic state space reduction with guarded terms for rewriting modulo SMT. Science of Computer Programming **178**, 20–42 (2019)
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. pp. 171–177. Springer (2011)
3. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB standard: Version 2.0. In: SMT. vol. 13, p. 14 (2010)
4. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Invariant checking of NRA transition systems via incremental reduction to LRA with EUF. In: TACAS. pp. 58–75. Springer (2017)
5. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.: Maude manual (version 3.0). Tech. rep., SRI International, Menlo Park (2020)
6. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Talcott, C.: Two decades of Maude. In: Martí-Oliet, N., Ölveczky, P.C., Talcott, C. (eds.) Logic, Rewriting, and Concurrency, pp. 232–254. Springer (2015)
7. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. pp. 337–340. Springer (2008)
8. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV. LNCS, vol. 8559, pp. 737–744. Springer (July 2014)
9. Jovanović, D., De Moura, L.: Solving non-linear arithmetic. In: International Joint Conference on Automated Reasoning. pp. 339–354. Springer (2012)
10. Rocha, C., Meseguer, J., Muñoz, C.: Rewriting modulo SMT and open system analysis. J. Log. Algebr. Methods Program **86**(1), 269–297 (2017)

Author Index