Symbolic Specialization of Rewriting Logic Theories with Presto *

MARÍA ALPUENTE, SANTIAGO ESCOBAR, JULIA SAPIÑA
VRAIN (Valencian Research Institute for Artificial Intelligence), Universitat Politècnica de València
(*e-mail:* {alpuente,sescobar,jsapina}@upv.es)

DEMIS BALLIS
DMIF, University of Udine
(*e-mail:* demis.ballis@uniud.it)

submitted 1 June 2020; revised 1 June 2020; accepted 1 June 2020

Abstract

This paper introduces Presto, a symbolic partial evaluator for Maude's rewriting logic theories that can improve system analysis and verification. In Presto, the automated optimization of a conditional rewrite theory $\mathscr{R}$ (whose rules define the concurrent transitions of a system) is achieved by partially evaluating, with respect to the rules of $\mathscr{R}$, an underlying, companion equational logic theory $\mathscr{E}$ that specifies the algebraic structure of the system states of $\mathscr{R}$. This can be particularly useful for specializing an overly general equational theory $\mathscr{E}$ whose operators may obey complex combinations of associativity, commutativity, and/or identity axioms, when being plugged into a host rewrite theory $\mathscr{R}$ as happens, for instance, in protocol analysis, where sophisticated equational theories for cryptography are used. Presto implements different unfolding operators that are based on *folding variant narrowing* (the symbolic engine of Maude's equational theories). When combined with an appropriate abstraction algorithm, they allow the specialization to be adapted to the theory termination behavior and bring significant improvement while ensuring strong correctness and termination of the specialization. We demonstrate the effectiveness of Presto in several examples of protocol analysis where it achieves a significant speed-up. Actually, the transformation provided by Presto may cut down an infinite folding variant narrowing space to a finite one, and moreover, some of the costly algebraic axioms and rule conditions may be eliminated as well. As far as we know, this is the first partial evaluator for Maude that respects the semantics of functional, logic, concurrent, and object-oriented computations.

KEYWORDS: Multi-paradigm Declarative Programming, Partial Evaluation, Rewriting Logic, Narrowing

1 Introduction

Partial evaluation (PE) is a general and powerful optimization technique that automatically specializes a program to a part of its input that is known statically (Jones et al., 1993). PE is motivated by the fact that dedicated programs tend to operate more efficiently than general-purpose ones. It has found applications ranging from compiler optimization to test case generation, among many

* This research was partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215, grant RTI2018-094403-B-C32 funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe", and by Generalitat Valenciana under grant PROMETEO/2019/098. Julia Sapiña has been supported by the Generalitat Valenciana APOSTD/2019/127 grant.

others (Cadar and Sen, 2013; Cook and Lämmel, 2011). In the context of logic programming (LP), partial evaluation is often called partial deduction (PD), while the term partial evaluation is often reserved for the specialization of impure logic programs. PD not only allows input variables to be instantiated with constant values but it also deals with terms that may contain logic variables, thus providing extra capabilities for program specialization (Martens and Gallagher, 1995).

Rewriting Logic (RWL) is a very general *logical* and *semantic framework* that is particularly suitable for modeling and analyzing complex, highly nondeterministic software systems (Martí-Oliet and Meseguer, 2002). Rewriting Logic is efficiently implemented in the high-performance[1] language Maude (Clavel et al., 2007a), which seamlessly integrates functional, logic, concurrent, and object-oriented computations. A Maude *rewrite theory* $\mathscr{R} = (\Sigma, E \uplus B, R)$ combines a term rewriting system R, which specifies the concurrent transitions of a system (for a signature Σ of program operators together with their type definition), with an *equational theory* $\mathscr{E} = (\Sigma, E \uplus B)$ that specifies system states as terms of an algebraic datatype. The equational theory $\mathscr{E}$ contains a set E of equations and a set B of axioms (i.e., distinguished equations that specify algebraic laws such as commutativity, associativity, and unity for some theory operators that are defined on Σ). The equations of E are implicitly oriented from left to right as rewrite rules and operationally used as simplification rules, while the axioms of B are mainly used for B-matching so that rewrite steps in $\mathscr{R}$ are performed *modulo* the equations and axioms of $E \uplus B$.

For instance, in protocol specification, the equations of E are often used to define the cryptoarithmetic functions that are used in the protocol transition rules (e.g., addition, exponentiation, and exclusive-or), while the axioms of B specify properties of the cryptographic primitives (e.g., commutativity of addition and multiplication). For example, consider the (partial) specification of integer numbers defined by the equations $E = \{X + 0 = X,\ X + s(Y) = s(X+Y),\ p(s(X)) = X,\ s(p(X)) = X\}$, where variables X, Y are of *sort* `Int` (types are called *sorts* in RWL), operators p and s respectively stand for the predecessor and successor functions, and B contains the commutativity axiom $X + Y = Y + X$. Also consider that the program signature Σ contains a binary state constructor operator $\langle _,_ \rangle$: `Int x Int` $\to$ `State` for a new sort `State` that models a simple network of processes that consume a token from one place (denoted by the first component of the state) and transmit it to another place (the second component), keeping the total amount of tokens invariant. The system state $t = \langle s(0), s(0) + p(0) \rangle$ can be rewritten to $\langle 0, s(0) \rangle$ (modulo the equations of E and the commutativity of $+$) using the following rule that specifies the system dynamics:

$$\langle A, B \rangle \Rightarrow \langle p(A), s(B) \rangle, \text{ where } A \text{ and } B \text{ are variables of sort } \texttt{Int} \quad (1)$$

Multiparadigm, functional logic languages combine features from functional programming (efficient evaluation strategies, nested expressions, genericity, advanced typing, algebraic data types) and logic programming (logical variables, partial data structures, nondeterministic search for solutions, constraint solving modulo theories) (Meseguer, 1992b; Hanus, 1997; Escobar, 2014). The operational principle that supports functional and logic language integration is called *narrowing* (Fay, 1979; Slagle, 1974), which is a goal-solving mechanism that subsumes the resolution principle of logic languages and the reduction principle of functional languages. Roughly

[1] Maude's rewriting machinery is highly optimized. Recently, an experimental open platform has been developed that allows the performance of functional and algebraic programming languages to be compared, including CafeOBJ, Clean, Haskell, LNT, LOTOS, Maude, mCRL2, OCaml, Opal, Rascal, Scala, SML, Stratego / XT, and Tom (see references in (Garavel et al., 2018)). In the top 5 of the most efficient tools, Maude ranks second after Haskell.

speaking, narrowing can be seen as a generalization of term rewriting that allows free variables in terms (as in logic programming) and that non-deterministically reduces these partially instantiated function calls by using unification (instead of pattern-matching) at each reduction step (Hanus, 1994).

Besides rewriting with rules modulo equations and axioms, Maude provides full *native* support for narrowing computations in rewrite theories since Maude version 3.0 (2020). This endows Maude with symbolic reasoning capabilities (e.g., symbolic reachability analysis in rewrite theories based on narrowing) and unification modulo user-definable equational theories. In our example theory, a narrowing reachability goal from $\langle V+V, 0+V\rangle$ to $\langle p(0), s(0)\rangle$ succeeds (in one step) with computed substitution $\{V \mapsto 0\}$[2], which might signal a possible programming error in rule (1) since the number of available tokens in the first component of the state becomes negative, thereby pinpointing a token shortage. This kind of advanced, LP-like reasoning capability based on narrowing goes beyond standard, rewrite-based equational reasoning that does not support symbolic reachability analysis because input variables are simply handled as constants. Symbolic reasoning methods that are based on narrowing and their applications are generally discussed in (Meseguer, 2020; Meseguer, 2021). For a very brief account of Maude's rewriting and narrowing principles, we refer to (Alpuente et al., 2019b).

In this paper, we develop the narrowing-based partial evaluator Presto and show how it can have a tremendous impact on the symbolic analysis of concurrent systems that are modeled as Maude rewrite theories. Traditional partial evaluation techniques typically remove some computation states by performing as much program computation as possible, hence contracting the search space because some transitions are eliminated. However, in the specialization of concurrent systems that are specified by Maude rewrite theories, we are only interested in compressing the deterministic, functional computations (which normalize the system states and are encoded by means of the theory equations) while preserving all concurrent state transitions (which are defined by means of the rewrite rules). This ensures that all reachable system states are preserved so that any reachability property of the original concurrent system can be correctly analyzed in the specialized one.

Given a rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$ our method proceeds by partially evaluating the underlying equational theory $\mathscr{E} = (\Sigma, E \uplus B)$ with respect to the function calls in the rules of $\mathscr{R}$ in such a way that $\mathscr{E}$ gets rid of any possible overgenerality. By doing this, only the functional computations in $\mathscr{E}$ are compressed by partial evaluation, while keeping every system state in the search space of the concurrent computations of $\mathscr{R}$. Partial evaluation can dramatically help in this process by introducing polyvariance (i.e., specializing a given function into different variants according to distinct invocation contexts (Martens and Gallagher, 1995)). The transformation performed by Presto is non-trivial because, depending on the properties of both theories ($\mathscr{E}$ and $\mathscr{R}$), the right unfolding and abstraction operators are needed to efficiently achieve the largest optimization possible while ensuring termination and total correctness of the transformation.

The partial evaluator Presto provides near-perfect support for the Maude language; specifically, it deals with all of the features that are currently supported by Maude's narrowing infrastructure. This effort is remarkable for at least two reasons: (i) Maude has quite sophisticated

[2] It is calculated by first computing the most general $\mathscr{E}$-unifier σ of the input term $\langle V+V, 0+V\rangle$ and the left-hand side $\langle A, B\rangle$ of rule (1), $\sigma = \{A \mapsto (V+V), B \mapsto V\}$ (σ is a $\mathscr{E}$-unifier of t and s if $s\sigma$ is equal to $t\sigma$ modulo the equations and axioms of $\mathscr{E}$). Second, an $\mathscr{E}$-unifier σ' is computed between the instantiated right-hand side $\langle p(V+V), s(V)\rangle$ and the target state $\langle p(0), s(0)\rangle$, $\sigma' = \{V \mapsto 0\}$. Third, the composition $\sigma\sigma' = \{A \mapsto 0+0, B \mapsto 0, V \mapsto 0\}$ is simplified into $\{A \mapsto 0, B \mapsto 0, V \mapsto 0\}$ and finally restricted to the variable V in the input term, yielding $\{V \mapsto 0\}$.

features (subtype polymorphism, pattern matching and equational unification modulo associativity, commutativity and identity axioms, equations, rules, modules, objects, *etc.*); and (ii) in order to efficiently achieve aggressive specialization that scales to real-world problems, the key components of the Presto system needed to be thoroughly investigated and highly optimized over the years. This is because equational problems such as order-sorted equational homeomorphic embedding and order-sorted equational least general generalization[3] are much more costly than their corresponding "syntactic" counterparts and achieving proper formalizations and efficient implementations has required years (Alpuente et al., 2008; Alpuente et al., 2009b; Alpuente et al., 2014b; Alpuente et al., 2019a; Alpuente et al., 2020b; Alpuente et al., 2020c; Alpuente et al., 2021c).

1.1 Related work

Traditional applications of partial evaluation include the optimization of programs that explore a state space by trying to reduce the amount of search. Relevant prior work includes the use of partial deduction to optimize a theorem prover with respect to a given theory (de Waal and Gallagher, 1994) and to refine infinite state model checking (Leuschel and Gruner, 2001), where both the theorem prover and the model checker are written as a logic program that searches for a proof, and the experiments use a standard partial evaluator that is supplemented with abstract interpretation (Cousot and Cousot, 1977) so that infinite branches in the search space are removed. Also, the specialization of concurrent processes that are modeled by (infinite state) Petri nets or by means of process algebras is performed in (Leuschel and Lehmann, 2000) by a combination of partial deduction and abstract interpretation.

Among the vast literature on program specialization, the partial evaluation of functional logic programs (Alpuente et al., 1998b; Albert et al., 2002; Hanus and Peemöller, 2014) is the closest to our work. The *narrowing-driven Partial Evaluation* (NPE) algorithm of (Alpuente et al., 1998b), implemented in the Indy system (Albert et al., 1998), extends to narrowing the classical PD scheme of (Martens and Gallagher, 1995) and was proved to be strictly more powerful than the PE of both logic programs and functional programs (Alpuente et al., 1998b), with a potential for specialization that is comparable to conjunctive partial deduction (CPD) and positive supercompilation (De Schreye et al., 1999). Early instances of this framework implemented partial evaluation algorithms for different narrowing strategies, including lazy narrowing (Alpuente et al., 1997), innermost narrowing (Alpuente et al., 1998b), and needed narrowing (Albert et al., 1999; Alpuente et al., 2005).

NPE was extended in (Alpuente et al., 2020b) to the specialization of *order-sorted equational theories* and implemented in the partial evaluator for equational theories Victoria. This was essentially achieved by generalizing the key NPE ingredients to work with order-sorted equations and axioms: 1) a tree *unfolding operator* based on folding variant narrowing that ensures strong correctness of the transformation; 2) a novel notion of order-sorted equational *homeomorphic embedding* that achieves local termination (i.e., finiteness of unfolding); 3) a suitable notion of order-sorted equational *closedness* (coveredness of the tree leaves modulo axioms) that ensures strong completeness; and 4) an *abstraction operator* based on order-sorted equational least general generalization that provides global termination of the whole specialization process. However,

[3] Generalization, also known as anti-unification, is the dual of unification: a generalization of two terms t_1 and t_2 is any term t of which t_1 and t_2 are substitution instances (Plotkin, 1970).

the partial evaluator Victoria is only able to specialize deterministic and terminating equational theories, while the Presto system described in this article deals with non-deterministic and non-terminating rewrite theories that allow concurrent systems to be modeled and model-checked. For a detailed discussion of the literature related to narrowing-driven partial evaluation, we refer to (Alpuente et al., 2020b).

The partial evaluator Presto implements our most ambitious generic specialization framework for RWL. Currently, there is hardly any system that can automatically optimize rich rewrite theories that include sorts, subsort overloading, rules, equations, and algebraic axioms. Actually, there are very few transformations in the related literature that can improve the analysis of Maude's rewrite theories and they are only known to the skilled practitioner, and, moreover, they lack automated tool support. For instance, the *total evaluation* of (Meseguer, 2020), which only applies to a restricted class of equational theories called *constructor finite variant theories*. Moreover, unlike the optimization that is achieved by Presto, none of the existing transformations respects the *narrowing semantics* of the theory (both values and computed substitutions), just its *ground value semantics* or its *reduction (normal form) semantics* (Alpuente et al., 2010; Lucas and Meseguer, 2016).

Many kinds of tools are built in Maude that rely on reflection and theory transformations and preserve specific properties such as invariants or termination behavior. Full-Maude (Clavel et al., 2020), Real-time Maude (Ölveczky and Meseguer, 2008), MTT (Durán et al., 2008), and Maude-NPA (Escobar et al., 2009) are prominent examples. Equational abstraction (Meseguer et al., 2008; Bae et al., 2013) reduces an infinite state system to a finite quotient of the original system algebra by introducing some extra equations that preserve certain temporal logic properties. Explicit coherence (a kind of confluence between rules, equations, and axioms) is necessary for executability purposes and also relies on rewrite theory transformations (Viry, 2002; Meseguer, 2020; Durán et al., 2020b). Also the semantic $\mathbb{K}$-framework (Roşu, 2017), the model transformations of (Rodríguez et al., 2019), and the automated program correction technique of (Alpuente et al., 2019c; Alpuente et al., 2020a) are based on sophisticated program transformations that preserve the reduction semantics of the original theory. Nevertheless, none of them aim to achieve program optimization.

The generic narrowing-driven partial evaluation scheme was originally formulated in (Alpuente et al., 2021b) for (unconditional) rewrite theories. An instance of the framework that mimicks the total evaluation of (Meseguer, 2020) can be found in (Alpuente et al., 2021a). Experiments with a preliminary version of the Presto system together with its basic underpinnings can also be found in (Alpuente et al., 2021b).

1.2 Contributions

The original contributions of this article with respect to the partial evaluation scheme for unconditional rewrite theories presented in (Alpuente et al., 2021b) are as follows.

1. We extend the generic narrowing-driven partial evaluation scheme of (Alpuente et al., 2021b) to the specialization of rewrite theories that may contain conditional rules. Given the rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$, this is done as follows. First, the equational theory is partially evaluated with respect to both the function calls in the right-hand sides and the conditions of the rewrite rules of R. Then, a post-processing refactoring transformation is applied to both equations and rules that gets rid of unnecessary symbols and delivers a more efficient, computationally equivalent rewrite theory where some of the costly

algebraic axioms and rule conditions may simply get removed. The new rule condition simplification algorithm is quite inexpensive and does not modify the program semantics.

2. A novel transformation called *topmost extension* is provided that achieves in one shot the two executability conditions that are required for the completeness of narrowing in Maude rewrite theories: the *explicit coherence*[4] of rules with respect to equations and axioms, and the *topmost* requirement on rules that forces all rewrites to happen on the whole state term —not on its subterms. We demonstrate that the topmost transformation is correct and preserves the solutions of the original theory so narrowing-based symbolic reachability analysis is enabled for all rewrite theories after the topmost transformation.
3. We apply our novel results on the topmost extension to support narrowing-based specialization for *object-oriented*[5] specifications. It is worth noting that Maude's object-oriented rules are typically non-topmost and they were not covered by (Alpuente et al., 2021b).
4. We have expanded the Presto system that implements the partial evaluation scheme of (Alpuente et al., 2021b) both to cope with conditions in rules and to handle Maude object-oriented modules that (in addition to equations, axioms, and rewrite rules) support classes, objects, messages, multiple class inheritance, and object interaction rules.
5. We also endowed Presto with an automated checker for: 1) *strong irreducibility*, a mild condition enforcing that the left-hand sides of the rewrite rules of R cannot be narrowed in E modulo B, which is needed for the strong completeness of our specialization method (Alpuente et al., 2020b); and 2) a novel property called *U-tolerance* ensuring that least general generalization with identity axioms is finitary even if function symbols have more than one unit element (Alpuente et al., 2021c). This scenario can occur when a rewrite theory contains multiple overloaded function symbols associated with distinct identity elements. Although most commonly occurring theories satisfy these properties, these checks are an important addition to the system because checking them manually is painful, since they involve term unification modulo any combination of equational axioms, which is difficult to calculate by hand.
6. We describe the system capabilities and provide an empirical evaluation of Presto on a set of protocol analysis problems where it demonstrates a significant speed-up.

1.3 Plan of the paper

In Section 2, we provide some preliminary notions on RWL, and we introduce a leading example that will be used throughout the paper to describe the specialization capabilities of Presto. Section 3 recalls the narrowing-based symbolic principles of Maude and formalizes an automatic program transformation that generates topmost rewrite theories for which the narrowing machinery for solving symbolic reachability problems can be effectively used. Specifically it enables symbolic reachability for object-oriented rewrite theories. In Section 4, we present our basic specialization scheme for conditional rewrite theories and describe its core functionality. Section 5 provides two instantiations of the specialization scheme that exploit two distinct unfolding operators that are suitable for theories that respectively have either an infinite behavior

[4] Currently, coherence is implicitly and automatically provided by Maude for rewriting computations. However, for narrowing computations, explicit coherence must be specifically ensured (Clavel et al., 2020; Meseguer, 2020).

[5] Actually, object-oriented modules have been totally redesigned in Maude 3.1 (Clavel et al., 2020). They were not handled by the previous version of the Presto system in (Alpuente et al., 2020b).

or a finite behavior with regard to narrowing computations. In Section 6, we present an overview of Presto's main implementation choices and some additional tool features that allow the user to inspect the intermediate results of the specialization as well as to visualize all of the narrowing trees that are deployed during the specialization. In Section 7, we present experimental evidence that Presto achieves significant specialization, with some specialized theories running up to two orders of magnitude faster than the original ones. Section 8 concludes and discusses future work. Proofs of the main technical results are given in Appendix A.

2 Preliminaries

Let Σ be a *signature* that includes typed operators (also called function symbols) of the form $f\colon s_1 \ldots s_m \to s$, where s_i, for $i = 1, \ldots n$, and s are sorts in a poset $(S, <)$ that models subsort relations (e.g., $s < s'$ means that sort s is a subsort of s'). The connected components of $(S, <)$ are the equivalence classes corresponding to the least equivalence relation $\equiv_<$ containing $<$. As usual in rewriting logic (Alpuente et al., 2014b), we assume *kind-complete*[6] signatures such that: each connected component in the poset $(S, <)$ has a top sort (also called *kind*), and, for each $s \in S$, we denote by $[s]$ the top sort in the connected component of s (i.e., if s and s' are sorts in the same connected component, then $[s] = [s']$); and (ii) for each operator declaration $f : s_1 \times \ldots \times s_n \to s$ in Σ, there is also a declaration $f : [s_1] \times \ldots \times [s_n] \to [s]$ in Σ. Binary operators in Σ may have an axiom declaration attached that specifies any combinations of algebraic laws such as associativity (A), commutativity (C), identity (U), left identity (U_l), and right identity (U_r). We use strings over the alphabet $\{A, C, U, U_l, U_r\}$ to indicate the combinations of axioms satisfied by an operator f. By $ax(f)$, we denote the set of algebraic axioms for the operator f.

We consider an S-sorted family $\mathcal{X} = \{\mathcal{X}_s\}_{s \in S}$ of disjoint variable sets. $\mathcal{T}_{\Sigma,s}(\mathcal{X})$ and $\mathcal{T}_{\Sigma,s}$ are the sets of terms and ground terms of sort s, respectively. By $\mathcal{T}_{\Sigma}(\mathcal{X}) = \bigcup_{s \in S} \mathcal{T}_{\Sigma,s}(\mathcal{X})$, we denote the usual non-ground term algebra built over Σ and the variables in $\mathcal{X}$. By $\mathcal{T}_{\Sigma} = \bigcup_{s \in S} \mathcal{T}_{\Sigma,s}$, we denote the ground term algebra over Σ. By notation $x : s$, we denote a variable x with sort s. The set of variables that appear in a term t is denoted by $Var(t)$.

We assume *pre-regularity* of the signature Σ: for each operator declaration $f\colon s_1 \ldots s_m \to s$, and for the set S_f containing all sorts s' that appear in operator declarations of the form : $f\colon s'_1 \ldots s'_m \to s'$ in Σ such that $s_i < s'_i$ for $1 \leq i \leq m$, then the set S_f has a least sort. Given a term $t \in \mathcal{T}_{\Sigma}(\mathcal{X})$, $ls(t)$ denotes the *least sort* of t in the poset $(S, <)$. An expression $\overline{t_n}$ denotes a finite sequence of terms $t_1 \ldots t_n$, $n \geq 0$. A *position* w in a term t is represented by a sequence of natural numbers that addresses a subterm of t (Λ denotes the empty sequence, i.e., the root position). Given a term t, we let $Pos(t)$ denote the set of positions of t. We denote the usual prefix preorder over positions by $\leq$. By $t_{|w}$, we denote the *subterm* of t at position w. By $root(t)$, we denote the operator of t at position Λ.

A *substitution* σ is a sorted mapping from a finite subset of $\mathcal{X}$ to $\mathcal{T}_{\Sigma}(\mathcal{X})$. Substitutions are written as $\sigma = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$. The identity substitution is denoted by id. Substitutions are homomorphically extended to $\mathcal{T}_{\Sigma}(\mathcal{X})$. The application of a substitution σ to a term t is denoted by $t\sigma$. The restriction of σ to a set of variables $V \subset \mathcal{X}$ is denoted $\sigma_{|V}$. Composition of two substitutions is denoted by $\sigma\sigma'$ so that $t(\sigma\sigma') = (t\sigma)\sigma'$.

A Σ-*equation* (or simply equation, where Σ is clear from the context) is an unoriented pair

[6] Actually, Maude automatically completes any input signature with special kinds to ensure uniqueness of top sorts.

$\lambda = \rho$, where $\lambda, \rho \in \mathcal{T}_{\Sigma,s}(\mathcal{X})$ for some sort $s \in S$, where $\mathcal{T}_{\Sigma,s}(\mathcal{X})$ is the set of terms of sort s built over Σ and $\mathcal{X}$. An equational theory $\mathcal{E}$ is a pair $(\Sigma, E \uplus B)$ that consists of a signature Σ, a set E of Σ-equations, and a set B of algebraic axioms (e.g., associativity, commutativity, and/or identity) that are expressed by means of equations for some binary operators in Σ. The equational theory $\mathcal{E}$ induces a congruence relation $=_{\mathcal{E}}$ on $\mathcal{T}_{\Sigma}(\mathcal{X})$.

A term t is more general than (or at least as general as) t' modulo $\mathcal{E}$, denoted by $t \leq_{\mathcal{E}} t'$, if there is a substitution γ such that $t' =_{\mathcal{E}} t\gamma$. We also define $t \simeq_{\mathcal{E}} t'$ iff $t \leq_{\mathcal{E}} t'$ and $t' \leq_{\mathcal{E}} t$. By abuse of notation, we write $\leq_B$ and $\simeq_B$ when B is an axiom set.

A substitution θ is more general than (or at least as general as) σ modulo $\mathcal{E}$, denoted by $\theta \leq_{\mathcal{E}} \sigma$, if there is a substitution γ such that $\sigma =_{\mathcal{E}} \theta\gamma$, i.e., for all $x \in \mathcal{X}, x\sigma =_{\mathcal{E}} x\theta\gamma$. Also, $\theta \leq_{\mathcal{E}} \sigma\ [V]$ iff there is a substitution γ such that, for all $x \in V$, $x\sigma =_{\mathcal{E}} x\theta\gamma$.

An *$\mathcal{E}$-unifier* for a Σ-equation $t = t'$ is a substitution σ such that $t\sigma =_{\mathcal{E}} t'\sigma$.

We consider three different kinds of expressions that may appear in a conditional rewrite theory: 1) an *equational condition*, which is is any (ordinary[7]) equation $t = t'$, with $t, t' \in \mathcal{T}_{\Sigma,s}(\mathcal{X})$ for some sort $s \in S$; 2) a *matching condition*, which is a pair $t := t'$, with $t, t' \in \mathcal{T}_{\Sigma,s}(\mathcal{X})$ for some sort $s \in S$; and 3) a *rewrite expression*, which is a pair $t \Rightarrow t'$, with $t, t' \in \mathcal{T}_{\Sigma,s}(\mathcal{X})$ for some sort $s \in S$. A *conditional* rule is an expression of the form $\lambda \Rightarrow \rho$ *if* C, where $\lambda, \sigma \in \mathcal{T}_{\Sigma,s}(\mathcal{X})$ for some sort $s \in S$, and C is a (possibly empty, with identity symbol *nil*) sequence $c_1 \wedge \ldots \wedge c_n$, where each c_i is an equational condition, a matching condition, or a rewrite expression. Conditions are evaluated[8] from left to right, and therefore the order in which they appear, although mathematically inessential, is operationally important. When the condition C is empty, we simply write $\lambda \Rightarrow \rho$. A *rewrite theory* is a triple $\mathcal{R} = (\Sigma, E \uplus B, R)$, where $(\Sigma, E \uplus B)$ is an equational theory and R is a set of rewrite rules. A rewrite theory $(\Sigma, E \uplus B, R)$ is called *topmost* if there is a sort *State* such that: (i) for each rewrite rule $\lambda \Rightarrow \rho$ *if* C, λ and ρ are of sort *State*; and (ii) for each $f \colon [s_1] \ldots [s_n] \to s \in \Sigma$ and $i \in \{1, \ldots, n\}$, $[s_i] \neq$ *State*. [9]

In a rewrite theory $\mathcal{R} = (\Sigma, E \uplus B, R)$, computations evolve by rewriting states using the *equational rewriting* relation $\to_{R,\mathcal{E}}$, which applies the rewrite rules in R to states *modulo the equational theory* $\mathcal{E} = (\Sigma, E \uplus B)$ (Meseguer, 1992a). The Maude interpreter implements equational rewriting $\to_{R,\mathcal{E}}$ by means of two simple relations, namely $\to_{\vec{E},B}$ and $\to_{R,B}$. These allow rules and (oriented) equations to be intermixed in the rewriting process by simply using both an algorithm of matching modulo B. The relation $\to_{\vec{E},B}$ uses $\vec{E}$ (the explicitly oriented version of the equations in E) for term simplification. Thus, for any term t, by repeatedly applying the equations as simplification rules, we eventually reach a term $t{\downarrow_{\vec{E},B}}$ to which no further equations can be applied. The term $t{\downarrow_{\vec{E},B}}$ is called *canonical form* (also called irreducible or normal form) of t w.r.t. $\vec{E}$ modulo B. On the other hand, the relation $\to_{R,B}$ implements rewriting with the rules of R, which might be nonterminating and nonconfluent, whereas $\mathcal{E}$ is required to be convergent (i,.e., $\to_{\vec{E},B}$ is terminating and confluent modulo B) and B-coherent (a kind of non-interference between E and B) in order to guarantee the existence and unicity (modulo B) of a canonical form w.r.t. $\vec{E}$ for

[7] A boolean equational condition $b = true$, with $b \in \mathcal{T}_{\Sigma}(\mathcal{X})$ of sort Bool, can be abbreviated as b, although it is internally represented as the equation $b = true$.

[8] Given a parameter-passing substitution σ, the interpretation of equational conditions $t' = t$ consists in the joinability of the normal forms $t\sigma$ and $t'\sigma$ (in E modulo B); pattern matching equations $t := t'$ are evaluated by computing the equational pattern matchers, within the instantiated pattern $t\sigma$, of the normal form of $t'\sigma$ (in E modulo B); and rewrite expressions $t \Rightarrow t'$ are interpreted as rewriting-based reachability goals in R (modulo $E \uplus B$).

[9] Note that *State* names an arbitrary sort for which conditions (i) and (ii) are satisfied.

any term. Also, the set R of rules must be coherent w.r.t. $\mathscr{E}$, ensuring that any rewrite step with $\rightarrow_{R,B}$ can always be postponed in favor of deterministically rewriting with $\rightarrow_{\vec{E},B}$.

Formally, for a set of rules P (with P being either R or $\vec{E}$), the rewriting relation $\rightarrow_{P,B}$ in P modulo B is defined as follows. Given a rewrite rule $r = (\lambda \Rightarrow \rho \textit{ if } C) \in P$, a substitution σ, a term t, and a position w of t, $t \overset{r,\sigma,w}{\rightarrow}_{P,B} t'$ iff $\lambda\sigma =_B t_{|w}$, $t' = t[\rho\sigma]_w$, and C holds (the specific evaluation strategy for rule conditions of Maude can be found in (Clavel et al., 2007b)). When no confusion arises, we simply write $t \rightarrow_{P,B} t'$ instead of $t \overset{r,\sigma,w}{\rightarrow}_{P,B} t'$.

Under these conditions, a $(R, E \uplus B)$-rewrite step $\rightarrow_{R,E\uplus B}$ on a term t in the rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$ can be implemented by applying the following rewrite strategy: (i) reduce t w.r.t. $\rightarrow_{\vec{E},B}$ to the canonical form $t\downarrow_{\vec{E},B}$; and (ii) rewrite $t\downarrow_{\vec{E},B}$ w.r.t. $\rightarrow_{R,B}$. This strategy is still complete in the sense that, rewriting of congruence classes induced by $=_{\mathscr{E}}$, with $\mathscr{E} = (\Sigma, E \uplus B)$, can be mimicked by $(R, E \uplus B)$-rewrite steps.

A rewrite sequence $t \rightarrow^*_{R,\mathscr{E}} t'$ in the rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$ is then deployed as the (possibly infinite) rewrite sequence (with $t_0 = t$ and $t_n\downarrow_{\vec{E},B} = t'$)

$$t_0 \rightarrow^*_{\vec{E},B} t_0\downarrow_{\vec{E},B} \rightarrow_{R,B} t_1 \rightarrow^*_{\vec{E},B} t_1\downarrow_{\vec{E},B} \rightarrow_{R,B} \dots \rightarrow_{R,B} t_n\downarrow_{\vec{E},B}$$

that interleaves $\rightarrow_{\vec{E},B}$ rewrite steps and $\rightarrow_{R,B}$ rewrite steps following the strategy mentioned above. Note that, after each rewrite step using $\rightarrow_{R,B}$, generally the resulting term t_i, $i = 1, \dots, n$, is not in canonical form and is thus normalized before the subsequent rewrite step using $\rightarrow_{R,B}$ is performed. Also, in the precise strategy adopted by Maude, the last term of a finite computation is finally normalized before the result is delivered.

2.1 Concurrent Object-Oriented Specifications: a Communication Protocol Model with Caesar *ciphering*

Rewrite theories provide a natural computation model for concurrent object-oriented systems (Meseguer, 1992a). Indeed, Maude fully supports the object-oriented programming paradigm by means of pre-defined data structures that identify the main building blocks of this paradigm such as classes, objects, and messages (Clavel et al., 2020). The essential facts about concurrent object-oriented configurations are all formalized in the `CONFIGURATION` module, which provides special notation and avoids boilerplate `class` declarations. More specifically, the sorts `Oid` and `Cid` respectively define object and class identifiers, while the sort `Msg` represents messages (which are typically used to model object communication). It is worth noting that class names are just sorts. Therefore, class inheritance is directly supported by rewriting logic order-sorted type structure. A subclass declaration is thus an expression of the form `subsort C < C′` where `C` and `C′` are two sorts representing object classes. Multiple inheritance is also supported, allowing a class `C` to be defined as a subsort of many several sorts (each of which represent a different class).

The partial evaluation framework that we define in this work is able to deal with objects that are represented as terms of the following form:

$$< O \,:\, C \mid a_1 \,:\, v_1,\, \dots\, ,\, a_n \,:\, v_n >$$

where O is a term of sort *Oid*, C is a term of sort *Cid*, and $a_1 : v_1, \dots, a_n : v_n$ is a list of attributes of sort *Attribute*, each consisting of an identifier a_i followed by its respective value v_i.

The concurrent state of an object-oriented system is a multiset (of sort *Configuration*) of objects and messages that are built using the empty syntax (juxtaposition) ACU operator __ *:*

Configuration Configuration $\rightarrow$ *Configuration* whose identity is *none*. Thus, an object-oriented configuration will be either *none* (empty configuration) or $Ob_1\ \ldots\ Ob_k\ Mes_1\ \ldots\ Mes_n$, where $Ob_1,\ldots,Ob_k$ are objects, and $Mes_1,\ldots,Mes_n$ are messages.

Transitions between object-oriented configurations are specified by means of rewrite rules of the form

$$Ob_1\ \ldots\ Ob_k\ Mes_1\ \ldots\ Mes_n \Rightarrow Ob'_1\ \ldots\ Ob'_j\ Ob_{k+1}\ \ldots\ Ob_m\ Mes'_1\ \ldots\ Mes'_p\ if\ Cond.$$

where $Ob'_1\ \ldots\ Ob'_j$ are updated versions of $Ob_1\ \ldots\ Ob_j$ for $j \leq k$, $Ob_{k+1}\ \ldots\ Ob_m$ are newly created objects, $Mes'_1\ \ldots\ Mes'_p$ are new messages, and *Cond* is a rule condition.

An important special case are rules with a single object and at most one message on the left-hand side. These rules directly model asynchronous distributed interactions. Rules involving multiple objects are called synchronous and are used to model higher-level communication abstractions.

Example 1

Let us consider a rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$ that encodes an object-oriented[10] specification for a client-server communication protocol in an asynchronous medium where messages can arrive out-of-order. The theory signature Σ includes several operators and sorts that model the protocol entities. The constant operators `Cli` and `Serv` (of respective sorts `Client` and `Server`) are used to specify two classes that respectively identify clients and servers. Both sorts inherit the class behavior of the built-in abstract class `Cid` via the subsort relations `Client < Cid` and `Server < Cid`.

Data exchanged between clients and servers is encoded as non-empty, associative sequences $\mathtt{s_1 \ldots s_n}$, where, for the sake of simplicity, each $\mathtt{s_i}$ is a term of sort `Symbol` in the alphabet `{a,b,c}`. We assume that `Symbol` is a subsort of `Data`; hence, any symbol is also a (one-symbol) data sequence.

Clients are represented as objects of the form

`< C : Cli | server : S , data : M , key : K , status : V >`

where `C` is an object identifier for a client of the class `Cli`, `S` is the server object that `C` wants to communicate with, `M` is a piece of data representing a client request, `K` is a natural number (specified in Peano's notation) that determines an encryption/decryption key for messages, and `V` is a constant value that models the client status. Initially, the status is set to the empty value `mt`, and it changes to `success` whenever a server acknowledges message reception.

Servers are simple objects of the form

`< S : Serv | key : K >`

where `S` is an object identifier for a server, and `K` is an encryption/decryption key.

Network packets are naturally modeled by object messages that can be exchanged between servers and clients. More specifically a network packet is a pair of the form `Host <- CNT` of sort `Msg`, where `Host` is a client or server recipient, and `CNT` specifies the packet content. Specifically, `CNT` is a term `{H,M}`, with `H` being the sender's name and `M` being a data item that represents either a client request or a server response, expressed as a list of symbols in the alphabet `{a,b,c}`.

[10] A non-object-oriented, topmost version of this protocol can be found in (Alpuente et al., 2021b).

```
crl [req] : < C : Cli | server : S ,
                        data : M ,
                        key : s(K) ,
                        status : mt >
           =>

           < C : Cli | server : S ,
                       data : M ,
                       key : s(K) ,
                       status : mt >
           ( S <- { C , enc(M,s(K)) } ) if s(K) < len /\
                                       enc(M,s(K)) = enc(enc(M,K),s(0)) .

rl [reply] :  < S : Serv | key : K >
              (S <- {C,M})

              =>

               < S : Serv | key : K >
               (C <- {S, dec(M,K)}) [narrowing] .

rl [rec] : < C : Cli | server : S ,
                       data : M ,
                       key : K ,
                       status : mt >
           (C <- {S,M})

           =>

           < C : Cli | server : S ,
                       data : M ,
                       key : K ,
                       status: success > [narrowing] .
```

Fig. 1. Rewrite rules for the client-server communication protocol.

System states are represented by object-oriented configurations which may include clients, servers, and network packets.

The protocol dynamics is specified by the term rewriting system R in $\mathscr{R}$ of Figure 1. It consists of three rules, where clients and servers agree on a shared key `K`.

More specifically, the rule `req` allows a client `C` to initiate a transmission request with a server `S` by sending the data item `M` that is encrypted by function `enc(M,s(K))` using a positive client's key `s(K)`. Note that this rule is conditional and is enabled if and only if the key `s(K)` is less than the constant `len`, which represents the cardinality of the chosen alphabet of symbols. Furthermore, to initiate a client request, we require the satisfaction of the property `enc(M,s(K)) = enc(enc(M,K),s(0))`, which enforces an additive property on the `enc` function.

The rule `reply` lets the server `S` consume a client request packet `S <- {C,M}` by first decrypting the incoming data item `M` with the server key and then sending a response packet back to `C` that includes the decrypted request data. The rule `rec` successfully completes the data transmission between `C` and `S` whenever the server response packet `C <- {S,M}` includes a data item `M` that is equal to the initial client request message. In this case, the status of the client is changed from `mt` to `success`. Note that the transmission succeeds when the client and server use the same key `K`, as this guarantees that the client plain message `M` is equal to `dec(enc(M,K),K)` as required by the `rec` rule.

Encryption and decryption functionality is implemented by two functions (namely, `enc(M,K)`

```
eq len = s(s(s(0))) [variant] .  --- alphabet cardinality is hardcoded
                                 --- via the constant len = 3

--- Function toNat(s) takes an alphabet symbol s as input and returns the
--- corresponding position in the alphabet.
eq toNat(a) = 0 [variant] .
eq toNat(b) = toNat(a) + s(0) [variant] .
eq toNat(c) = toNat(b) + s(0) [variant] .

--- Function toSym(n) takes a natural number n as input and returns the
--- corresponding alphabet symbol.
eq toSym(0)= a [variant] .
eq toSym(s(0)) = b [variant] .
eq toSym(s(s(0))) = c [variant] .

var M : Data .
vars K X Y  : Nat .

--- Function shift(k) increments (modulo the alphabet cardinality)
--- the natural number k.
eq shift(X) = [ s(X) < len,s(X), 0 ] [variant] .

--- Function unshift(k) decrements (modulo the alphabet cardinality)
--- the natural number k.
eq unshift(0) = s(s(0)) [variant] .
eq unshift(s(X)) = X [variant] .

--- Function e(n,k) increments the natural number n by k units
--- (modulo the alphabet cardinality).
eq e(X,0) = X [variant] .
eq e(X,s(Y)) = e(shift(X),Y) [variant] .

--- Function d(n,k) decrements the natural number n by k units
--- (modulo the alphabet cardinality).
eq d(X,0) = X [variant] .
eq d(X,s(Y)) = d(unshift(X),Y) [variant] .


--- Function enc(m,k)  (resp. dec(m,k)) takes a data item m and a
--- natural number k as input and returns the corresponding encrypted (resp. decrypted)
--- data item using the Caesar cipher with key K
eq enc(S:Symbol,K) = toSym(e(toNat(S:Symbol),K)) [variant] .
eq enc(S:Symbol M,K) = toSym(e(toNat(S:Symbol),K)) enc(M,K) [variant] .
eq dec(S:Symbol,K) = toSym(d(toNat(S:Symbol),K))[variant] .
eq dec(S:Symbol M,K) = toSym(d(toNat(S:Symbol),K)) dec(M,K) [variant] .
```

Fig. 2. Equations of the equational theory encoding the *Caesar* cipher.

and `dec(M,K)`) that are specified by the equational theory $\mathscr{E}$ in $\mathscr{R}$. The equational theory $\mathscr{E}$ implements a *Caesar* cipher with key `K`, which is a simple substitution ciphering where each symbol in the plaintext data item `M` is replaced by the symbol that appears `K` positions later in the alphabet (handled as the list `a,b,c`). The cipher is circular, i.e., it works modulo the cardinality of the alphabet. For instance, `enc(a b,s(0))` delivers `(b c)`, and `dec(a b,s(0))` yields `(c a)`. The equational theory $\mathscr{E}$ includes the equations[11] in Figure 2.

[11] For the sake of simplicity, we omitted the definition of the operators `[_,_,_]`, `_<_`, and `_+_` that respectively implement the usual *if-then-else* construct, the *less-than* relation, and the associative and commutative addition over natural numbers.

3 A Glimpse into Narrowing-based Symbolic Computation in Maude

Similarly to *rewriting modulo an equational theory* $\mathscr{E}$, where syntactic pattern-matching is replaced with matching modulo $\mathscr{E}$ (or $\mathscr{E}$-matching), in *narrowing modulo an equational theory*, syntactic unification is replaced by *equational* unification (or $\mathscr{E}$-unification).

Given a rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$, with $\mathscr{E} = (\Sigma, E \uplus B)$, the *conditional narrowing* [12] relation $\leadsto_{R,\mathscr{E}}$ on $\mathscr{T}_\Sigma(\mathscr{X})$ is defined by $t \leadsto_{\sigma,p,R,\mathscr{E}} t'$ (or simply $t \leadsto_{R,\mathscr{E}} t'$) if and only if there is a non-variable position $p \in Pos_\Sigma(t)$, a (renamed apart) rule $\lambda \Rightarrow \rho$ *if* C in R, and an $\mathscr{E}$-unifier of $t_{|p}$ and λ such that $t' = t[\rho]_p\sigma$ and condition $C\sigma$ holds (Aguirre et al., 2014). Furthermore, we write $t \leadsto_{\sigma,p,r,\mathscr{E}} t'$, when we want to highlight the rewrite rule r that has been used in the narrowing step. A term t is called $(R,\mathscr{E})$-*strongly irreducible* (also called a rigid normal form (Alpuente et al., 2009a)) iff there is no term u such that $t \leadsto_{\sigma,p,R,\mathscr{E}} u$ for any position p, which amounts to say that no subterm of t unifies modulo B with the left-hand side of any equation of E.

Narrowing derivations correspond to sequences $t_0 \leadsto_{\sigma_0,p_0,r_0,B} t_1 \leadsto_{\sigma_1,p_1,r_1,B} \cdots \leadsto_{\sigma_n,p_{n-1},r_{n-1},B} t_n$. The composition $\sigma_0\sigma_1 \ldots \sigma_{n-1}$ of all the unifiers along a narrowing sequence leading to t_n (restricted to the variables of t_0) is called *computed substitution*.

The search space of all narrowing derivations for a term t in $\mathscr{R} = (\Sigma, E \uplus B, R)$ is represented as a tree-like structure called $(R, E \uplus B)$-*narrowing tree* (or simply narrowing tree).

3.1 Two-level narrowing in Maude

A rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$ can be symbolically executed in Maude by using narrowing at *two levels*: (i) narrowing with oriented equations $\vec{E}$ (i.e., rewrite rules obtained by explicitly orienting the equations in E) modulo the axioms B; and (ii) narrowing with the (typically non-confluent and non-terminating) rules of R modulo $\mathscr{E} = (\Sigma, E \uplus B)$.

Narrowing with $\vec{E}$ modulo B. This form of narrowing is typically used for $\mathscr{E}$-unification and it is implemented in Maude via the *folding variant narrowing* (or simply fV-narrowing) strategy of (Escobar et al., 2012). fV-narrowing is centered around the notion of equational variant of a term. Given $\mathscr{E} = (\Sigma, E \uplus B)$, the set of all pairs (t', σ), where t' is $(t\sigma)\downarrow_{\vec{E},B}$, i.e., the canonical form of $t\sigma$ for a substitution σ, is called the set of *equational variants* (or simply variants) of a term t (Escobar et al., 2012). Intuitively, the variants of t are the "irreducible patterns" to which t can be symbolically evaluated by applying $\vec{E}$ modulo B. A variant (t, σ) is *more general* than a variant (t', σ') w.r.t. an equational theory $\mathscr{E}$ (in symbols, $(t, \sigma) \leq_{\mathscr{E}} (t', \sigma')$) iff there exists a substitution γ such that $t\gamma =_{\mathscr{E}} t'$ and $\sigma\gamma =_{\mathscr{E}} \sigma'$.

An equational theory $\mathscr{E}$ has the *finite variant property* (FVP) (or $\mathscr{E}$ is called a *finite variant theory*) iff there is a finite, complete, and minimal set of most general equational variants for each term. It is generally undecidable whether an equational theory has the FVP (Bouchard et al., 2013); a semi-decision procedure is given in (Meseguer, 2015) that works well in practice. The FVP can be semi-decided by checking whether there is a finite number of most general variants for all *flat* terms $f(X_1, \ldots, X_n)$ for any n-ary operator f in the theory and pairwise-distinct variables $X_1, \ldots, X_n$ (of the corresponding sort). For instance, the specification of the exclusive-or operator on natural numbers given by $E = \{N \oplus N = 0, N \oplus 0 = N\}$, with $\oplus$ being

[12] Conditional narrowing is not currently supported by Maude but can be easily mimicked through an unraveling transformation that removes the rule conditions (Meseguer, 2020).

commutative, is a finite variant theory, since the (flat) term $N \oplus M$ has four most general variants: $(N \oplus M, \{\})$, $(M, \{N \mapsto 0\})$, $(N, \{M \mapsto 0\})$, and $(0, \{N \mapsto M\})$. Instead, the specification of the auxiliary function `d` in Figure 2 does not satisfy the FVP, since the term `d(X,Y)` has an infinite number of most general variants (`X`, {`Y` ↦`0`}), (`unshift(X)`, {`Y` ↦`s(0)`}), ..., ($\texttt{unshift}^k\texttt{(X)}$, {`Y` ↦ $\texttt{s}^k\texttt{(0)}$}). Therefore, the equational theory of Example 1 does not have the FVP either.

The main idea of folding variant narrowing is to "fold"[13] the $(\vec{E}, B)$-narrowing tree by subsumption modulo B. That is, folding variant narrowing shrinks the search space and avoids computing any variant that is a substitution instance modulo B of a more general variant.

A fV-narrowing derivation is a sequence $t_0 \leadsto_{\sigma_0,p_0,\vec{e}_0,B} t_1 \leadsto_{\sigma_1,p_1,\vec{e}_1,B} \cdots \leadsto_{\sigma_n,p_{n-1},\vec{e}_{n-1},B} t_n$, where $t \leadsto_{\sigma,p,\vec{e},B} t'$ (or simply $t \leadsto_{\sigma} t'$ when no confusion can arise) denotes a transition (modulo the axioms B) from term t to t' via the *variant equation e* (i.e., an oriented equation $\vec{e}$ that is enabled to be used for fV-narrowing thanks to the attribute `variant`) using the B-unifier σ. Assuming that the initial term t is normalized, each step $t \leadsto_{\sigma,p,\vec{e},B} t'$ (or folding variant narrowing step) is followed by the simplification of the term into its normal form by using all equations in the theory, which may include not only the variant equations in the theory but also (non-variant) equations (e.g., built-in equations in Maude). In order for finite variant narrowing to be effectively applicable the equational theory $\mathscr{E} = (\Sigma, E \uplus B)$ must be convergent, B-coherent, finite variant, and B-unification must be decidable.

Narrowing with R modulo $\mathscr{E}$. It is mainly used for solving *reachability goals* (Meseguer and Thati, 2007) and *logical model checking* (Escobar and Meseguer, 2007).

Given a non-ground term t, t represents an abstract characterization of the (possibly) infinite set of all of the concurrent states $[\![t]\!]$ (i.e., all the ground substitution instances of t, or, more precisely, the $\mathscr{E}$-equivalence classes associated to such ground instances) within $\mathscr{R}$. In this scenario, each narrowing derivation $\mathscr{D}$ subsumes all of the rewrite computations that are "instances" of $\mathscr{D}$ modulo $\mathscr{E}$ (Clavel et al., 2020). Therefore, an exhaustive exploration of the $(R, E \uplus B)$-narrowing tree of t allows one to prove existential reachability goals of the form

$$(\exists X)\, t \longrightarrow^* t' \tag{2}$$

with t and t' being two (possibly) non-ground terms —called the *input* term and *target* term, respectively— and X being the set of variables appearing in both t and t'.

A *solution* for the reachability goal (2) in a rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$, with $\mathscr{E} = (\Sigma, E \uplus B)$, is a substitution σ such that $t\sigma \rightarrow^*_{R,\mathscr{E}} t'\sigma$; thus, any computed substitution θ of a narrowing derivation $t \leadsto^*_{R,\mathscr{E}} t'$ is a *solution* for the reachability goal (2) since $t\theta \rightarrow^*_{R,\mathscr{E}} t'\theta$.

In practice, solving a reachability goal $(\exists X)\, t \longrightarrow^* t'$ means searching for a symbolic solution within the $(R, E \uplus B)$-narrowing tree that originates from t in a hopefully *complete* way (so that, for any existing solution, a more general answer modulo $\mathscr{E}$ will be found). Completeness of narrowing with R modulo $\mathscr{E}$ holds for topmost rewrite theories, provided that a finitary and complete $\mathscr{E}$-unification algorithm exists[14].

We note that only rewrite rules with the `narrowing` attribute are used by $\leadsto_{R,\mathscr{E}}$, while any rule without this attribute is only considered for rewriting modulo $\mathscr{E}$. By $att(r)$ we denote the

[13] This notion is quite different from the classical folding operation of Burstall and Darlington's fold/unfold transformation scheme (Burstall and Darlington, 1977). The idea of *folding* in fV-narrowing is essentially a *subsumption* check that is applied to the tree leaves so that less general leaves are subsumed (folded into) most general ones.

[14] $\mathscr{E}$-unification can be effectively and completely computed by folding variant narrowing under the executability conditions presented in this section.

attributes[15] associated with the rewrite rule r. For instance, in Example 1, `rec` and `reply` rules can be used by $\leadsto_{R,\mathscr{E}}$, while `req` only supports rewriting modulo $\mathscr{E}$, since `narrowing` $\notin att(\texttt{req})$.

Unfortunately, not all the rewrite theories are topmost or admit finitary and complete $\mathscr{E}$-unification. For instance, object-oriented specifications of Section 2.1 are typically non-topmost, since their rewrite rules might apply to inner state positions, thereby implementing local state changes. This fact may compromise the effectiveness of narrowing-based symbolic reachability, because existing solutions of a given reachability goal may be missed due to incompleteness of $\leadsto_{R,\mathscr{E}}$. Let us see an example.

Example 2

Consider a simpler version of the client-server protocol given in Example 1, where the shared key `K` is set to a fixed value (for simplicity, `s(0)`) and messages consist of just one symbol. In this setting, we can greatly simplify the equational theory of Figure 2 into an equational theory $\mathscr{E}$ that only contains the encryption and decryption equations:

```
eq enc(a,s(0)) = b [variant] .
eq enc(b,s(0)) = c [variant] .
eq enc(c,s(0)) = a [variant] .
eq dec(a,s(0)) = c [variant] .
eq dec(b,s(0)) = a [variant] .
eq dec(c,s(0)) = b [variant] .
```

Note that $\mathscr{E}$-unification can be performed by fV-narrowing, since $\mathscr{E}$ has the FVP (in contrast to the equational theory of Example 1, which did not have the FVP) and fV-narrowing executability conditions are met (in particular, $\mathscr{E}$ is convergent and B-unification is trivially decidable since $B = \emptyset$).

The protocol dynamics is implemented by the rewrite rules `reply`, `rec`, and `req` that are specified in Example 1. Such rules are clearly not topmost since they may be applied to (strict) fragments of an object-oriented configuration.

Now, consider the reachability goal G

```
∃S : Symbol
        < Cli-A : Cli | server : Srv-A , data : S , key : s(0) , status : mt >
        < Srv-A : Serv | key : s(0) > (Srv-A <- { Cli-A , c } )
        →*
        < Cli-A : Cli | server : Srv-A , data : S , key : s(0) , status : success >
        < Srv-A : Serv | key : s(0) >
```

The goal admits the solution $\{\texttt{S} \mapsto \texttt{b}\}$ and proves that the handshake between client `Cli-A` and server `Srv-A` succeeds when `Cli-A` sends the encrypted message `c` to `Srv-A` if the original unencrpyted data `Q` is equal to `b`.

Nonetheless, since completeness of the narrowing relation $\leadsto_{R,\mathscr{E}}$ is not ensured for rewrite theories that are not topmost, the solution $\{\texttt{S} \mapsto \texttt{b}\}$ for G cannot be found. Indeed, executing G by means of the `vu-narrow` built-in Maude command, which implements narrowing-based reachability search in Maude, produces the following output:

[15] Other available statement attributes in Maude include `label`, `metadata`, `nonexec` and `print` (Clavel et al., 2020).

```
vu-narrow [1, 100] in CLI-SERV-PROTOCOL-OBJECT-ORIENTED-ONESYMBOL :
    < Cli-A : Cli | server : Srv-A , data : S:Symbol , key : s(0) , status : mt >
    < Srv-A : Serv | key : s(0) > (Srv-A <- {Cli-A,c})
    =>*
    < Cli-A : Cli | server : Srv-A , data : S:Symbol , key : s(0) , status : success >
    < Srv-A : Serv | key : s(0) > .

No solution.
rewrites: 3361 in 409ms cpu (413ms real) (8212 rewrites/second)
```

3.2 Relaxing the topmost requirement for object-oriented specifications

Object-oriented specifications, albeit not topmost, fall into the more general category of topmost *modulo Ax* rewrite theories, for which the topmost property can be easily recovered by means of an automatic program transformation. Topmost *modulo Ax* rewrite theories are rewrite theories where *Ax* consists of any of the combinations of axioms ACU, ACU_l, ACU_r, AC, AU, AU_l, AU_r, or A for a given binary symbol $\otimes : \mathit{Config}\,\mathit{Config} \to \mathit{Config}$ of the signature. The operator $\otimes$ is used to build *system configurations* that obey the *structural axioms* of $\otimes$ given by *Ax*.

Definition 1 (*topmost* modulo *Ax rewrite theory*)
Let $\mathscr{R} = (\Sigma, E \uplus B, R)$ be a rewrite theory with a finite poset of sorts $(S, <)$. Let *Config* be a top sort in *S*, and let $\otimes : \mathit{Config}\,\mathit{Config} \to \mathit{Config} \in \Sigma$ be a binary operator that obeys a combination of associativity, commutativity, and/or identity axioms $Ax \in \{ACU, ACU_l, ACU_r, AC, AU, AU_l, AU_r, A\}$.
The theory $\mathscr{R}$ is said to be topmost *modulo Ax* if

1. for each rule $(\lambda \Rightarrow \rho \text{ if } C)$ in R, λ and ρ are terms of sort *Config* and C does not contain terms of sort s such that its top sort $[s] = \mathit{Config}$;
2. $\otimes$ is the only operator in Σ whose arguments include a sort s such that its top sort $[s] = \mathit{Config}$.

The rewrite rules in R are also said to be topmost modulo *Ax*.

Example 3
The rewrite theory in Example 1 is topmost modulo ACU. To show this, it suffices to interpret the juxtaposition ACU operator _ _ *: Configuration Configuration* $\to$ *Configuration*, which is used to build object-oriented configurations, as the operator $\otimes : \mathit{Config}\,\mathit{Config} \to \mathit{Config}$ of Definition 1.

We are able to convert topmost modulo *Ax* rewrite theories into topmost theories automatically by means of the following solution-preserving theory transformation.

Definition 2 (*topmost extension of* $\mathscr{R}$)
Let $\mathscr{R} = (\Sigma, E \uplus B, R)$ be a topmost modulo *Ax* rewrite theory and $Ax \in B$. Let X, X_1, and X_2 be variables of sort *Config* not occurring in either R or E. We define the topmost rewrite theory $\hat{\mathscr{R}} = (\hat{\Sigma}, E \uplus B, \hat{R})$ where $\hat{\Sigma}$ extends Σ by adding a new top sort *State*, and a new operator $\{_\} : \mathit{Config} \to \mathit{State}$; and $\hat{R}$ is obtained by transforming R according to *Ax* as follows. For each $(\lambda \Rightarrow \rho \text{ } \mathit{if}\, C) \in R$

Case $Ax = ACU$.	$(\{X \otimes \lambda\} \Rightarrow \{X \otimes \rho\}\, if\ C) \in \hat{R}$;
Case $Ax = ACU_l$.	$(\{X \otimes \lambda\} \Rightarrow \{X \otimes \rho\}\, if\ C) \in \hat{R}$;
Case $Ax = ACU_r$.	$(\{X \otimes \lambda\} \Rightarrow \{X \otimes \rho\}\, if\ C) \in \hat{R}$;
Case $Ax = AC$.	$(\{X \otimes \lambda\} \Rightarrow \{X \otimes \rho\}\, if\ C) \in \hat{R}$, $(\{\lambda\} \Rightarrow \{\rho\}\, if\ C) \in \hat{R}$;
Case $Ax = AU$.	$(\{X_1 \otimes \lambda \otimes X_2\} \Rightarrow \{X_1 \otimes \rho \otimes X_2\}\, if\ C) \in \hat{R}$;
Case $Ax = AU_l$.	$(\{X_1 \otimes \lambda \otimes X_2\} \Rightarrow \{X_1 \otimes \rho \otimes X_2\}\, if\ C) \in \hat{R}$; $(\{X_1 \otimes \lambda\} \Rightarrow \{X_1 \otimes \rho\}\, if\ C) \in \hat{R}$;
Case $Ax = AU_r$.	$(\{X_1 \otimes \lambda \otimes X_2\} \Rightarrow \{X_1 \otimes \rho \otimes X_2\}\, if\ C) \in \hat{R}$; $(\{\lambda \otimes X_2\} \Rightarrow \{\rho \otimes X_2\}\, if\ C) \in \hat{R}$;
Case $Ax = A$.	$(\{X_1 \otimes \lambda \otimes X_2\} \Rightarrow \{X_1 \otimes \rho \otimes X_2\}\, if\ C) \in \hat{R}$, $(\{X_1 \otimes \lambda\} \Rightarrow \{X_1 \otimes \rho\}\, if\ C) \in \hat{R}$, $(\{\lambda \otimes X_1\} \Rightarrow \{\rho \otimes X_1\}\, if\ C) \in \hat{R}$, $(\{\lambda\} \Rightarrow \{\rho\}\, if\ C) \in \hat{R}$.

We call $\hat{\mathscr{R}}$ the *topmost extension* of $\mathscr{R}$.

The topmost extension of a rewrite theory $\mathscr{R}$ preserves the rewriting semantics of $\mathscr{R}$, as shown in the following proposition.

Proposition 3.1 (*correctness and completeness of the topmost extension*)
Let $\mathscr{R} = (\Sigma, E \uplus B, R)$, with $\mathscr{E} = (\Sigma, E \uplus B)$, be a topmost modulo Ax theory and let $\hat{\mathscr{R}}$ be the topmost extension of $\mathscr{R}$. For any terms t_i and t_f of sort *Config*, $t_i \rightarrow^*_{R,\mathscr{E}} t_f$ iff $\{t_i\} \rightarrow^*_{\hat{R},\mathscr{E}} \{t_f\}$.

The above proposition implies that the set of solutions of a reachability goal $G = (\exists X)\, t \longrightarrow^* t'$ in $\mathscr{R}$ is the same as the set of solutions of $\hat{G} = (\exists X)\ \{t\} \longrightarrow^* \{t'\}$ in the topmost extension $\hat{\mathscr{R}}$ of $\mathscr{R}$. Thus, to find a complete set of solutions in $\mathscr{R}$ for G, we can just find a complete set of solutions for $\hat{G}$ in $\hat{\mathscr{R}}$. Furthermore, as $\hat{\mathscr{R}}$ is topmost, narrowing with $\hat{R}$ modulo $\mathscr{E}$ is complete and can thus be effectively used to search for all solutions of G even if they could not be computed in Maude by narrowing in $\mathscr{R}$.

Theorem 3.1 (*strong completeness of the topmost extension*)
Let $\mathscr{R} = (\Sigma, E \uplus B, R)$ be a topmost modulo Ax rewrite theory, with $Ax \in B$, and let $\mathscr{E} = (\Sigma, E \uplus B)$ be a convergent, finite variant equational theory where B-unification is decidable. Let $\hat{\mathscr{R}} = (\hat{\Sigma}, E \uplus B, \hat{R})$ be the topmost extension of $\mathscr{R}$ and let $G = (\exists X)\ t \longrightarrow^* t'$ be a reachability goal in $\mathscr{R}$. If σ is a solution for G in $\mathscr{R}$, then there exists a term t'' such that $\{t\} \rightsquigarrow^*_{\hat{R},\mathscr{E}} \{t''\}$ with computed substitution θ and $\sigma =_{\mathscr{E}} \theta\eta$, where η is an $\mathscr{E}$-unifier of t'' and t'.

Example 4
Consider the rewrite theory $\mathscr{R}$ for the simplified client-server protocol and the reachability goal G of Example 2. Let `EXT` be a variable that we use as a placeholder for arbitrary object-oriented configurations to generate an ACU topmost extension $\hat{\mathscr{R}}$ of $\mathscr{R}$. The topmost extension $\hat{\mathscr{R}}$ includes the three rewrite rules of Figure 3.

In contrast to Example 2, here the solution $\{\mathtt{S} \mapsto \mathtt{b}\}$ for G can be automatically obtained in Maude by searching the solution for $\hat{G}$ in $\hat{\mathscr{R}}$ via narrowing. This is because $\hat{\mathscr{R}}$ is topmost and hence $\mathscr{E}$-unification is decidable. In fact, executing $\hat{G}$ in $\hat{\mathscr{R}}$ via the `vu-narrow` command

```
vu-narrow [1, 100] in CLI-SERV-PROTOCOL-OBJECT-ORIENTED-ONESYMBOL-EXTENDED :
    { < Cli-A : Cli | server : Srv-A , data : S:Symbol , key : s(0) , status : mt >
      < Srv-A : Serv | key : s(0) > (Srv-A <- {Cli-A,c}) }
```

```
crl [req'] : { < C : Cli | server : S ,
                        data : M ,
                        key : s(K) ,
                        status : mt > EXT }
            =>

            { < C : Cli | server : S ,
                        data : M ,
                        key : s(K) ,
                        status : mt > ( S <- { C , enc(M,K) } )
                        EXT } if s(K) < len /\ enc(M,s(K)) = enc(enc(M,K),s(0)) .

rl [reply'] : { < S : Serv | key : K >
                (S <- {C,M}) EXT }

               =>

               { < S : Serv | key : K >
                 (C <- {S, dec(M,K)}) EXT } [ narrowing ] .

rl [rec'] : { < C : Cli | server : S ,
                         data : M ,
                         key : K ,
                         status : mt >
             (C <- {S,M}) EXT }

            =>

            { < C : Cli | server : S ,
                         data : M ,
                         key : K,
                         status : success > EXT }[ narrowing ] .
```

Fig. 3. Topmost extension of the simplified client-server protocol of Example 2

```
    =>*
    { < Cli-A : Cli | server : Srv-A , data : S:Symbol , key : s(0) , status : success >
      < Srv-A : Serv | key : s(0) > } .
```

produces

```
Solution 1
rewrites: 55 in 10ms cpu (11ms real) (5257 rewrites/second)
state: { < Srv-A : Serv | key : s(0) > < Cli-A : Cli | server : Srv-A,data : b,
    key : s(0),status : success > }
accumulated substitution:
S:Symbol --> b
```

It is worth noting that the program transformation in Definition 2 is not only able to transform a topmost modulo Ax rewrite theory $\mathcal{R}$ into a semantically equivalent topmost rewrite theory, but it also ensures coherence of the rewrite rules of $\hat{\mathcal{R}}$ w.r.t. Ax for free. Indeed, the transformation algorithm of Definition 2 extends to object-oriented specifications the Maude (explicit) coherence completion algorithm that automatically produces coherent rewrite theories w.r.t. any given combination of ACU axioms (Clavel et al., 2020).

In the following section, we show that our partial evaluation algorithm can deal with finite variant theories as well as theories that do not satisfy the FVP, and it is powerful enough to transform them into specialized theories that, in many cases, do satisfy the FVP, thus enabling the use of both levels of narrowing on the theory. In Section 5, we show how the communication protocol of Example 1 can be automatically transformed into a specialized theory that meets the

FVP and in which reachability goals of the form $(\exists X)\ t \longrightarrow^* t'$ can be symbolically solved by means of narrowing after having applied the topmost extension described above.

4 Specializing Rewrite Theories with Presto

In this section, we present the specialization procedure NPER$^{\mathscr{U}}$, which allows a rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$ to be optimized by specializing the underlying equational theory $\mathscr{E} = (\Sigma, E \uplus B)$ with respect to the calls in the rewrite rules of R. The NPER$^{\mathscr{U}}$ procedure extends the equational, narrowing-driven partial evaluation algorithm EQNPE$^{\mathscr{U}}$ of (Alpuente et al., 2020b), which applies to equational theories and is parametric w.r.t. an unfolding operator $\mathscr{U}$ that is used to construct finite narrowing trees for a given expression.

4.1 Partial Evaluation of Equational Theories

Given $\mathscr{E} = (\Sigma, E \uplus B)$, we consider a natural partition of the signature as $\Sigma = \mathscr{D} \uplus \mathscr{C}$, where $\mathscr{C}$ are the *constructor* symbols, which are used to define the (irreducible) values of the theory, and $\mathscr{D}$ are the *defined* symbols, which are evaluated away by equational rewriting. Given a set Q of calls (henceforth called *specialized calls*), the main goal of EQNPE$^{\mathscr{U}}$ is to derive a new equational theory $\mathscr{E}'$ that computes the same substitutions (and values) as $\mathscr{E}$ for any input term t that is a recursive instance (modulo B) of the terms in Q (i.e., t is an instance of some term in Q and, recursively, every non-variable subterm of t that is rooted by a defined symbol is an instance of some term in Q). The algorithm follows the classic control strategy of logic specializers (Martens and Gallagher, 1995), with two separate components: 1) local control (managed by the unfolding operator), which avoids infinite evaluations and is responsible for the construction of the residual rules for each specialized call; 2) global control or control of polyvariance (managed by an abstraction operator), which avoids infinite iterations of the partial evaluation algorithm and decides which specialized functions appear in the transformed theory. Abstraction guarantees that only finitely many expressions are evaluated, thus ensuring global termination.

The transformation of (Alpuente et al., 2020b) is done by iterating two steps:

1. Symbolic execution (*Unfolding*). A finite, possibly partial folding variant narrowing tree for each input term t of Q is generated. This is done by using the unfolding operator $\mathscr{U}(Q, \vec{\mathscr{E}})$ that determines when and how to stop the derivations in the narrowing tree.

2. Search for regularities (*Abstraction*). In order to ensure that all calls that may occur at runtime are *covered* by the specialization, it must be guaranteed that every (sub-)term in any leaf of the tree is *equationally closed* w.r.t. Q. Equational closedness [16] extends the classical PD closedness (Lloyd and Shepherdson, 1991) by: 1) considering B-equivalence of terms; and 2) recursing over the term structure (in order to handle nested function calls). To properly add the non-closed (sub-)terms to the set of already partially evaluated calls, an abstraction operator $\mathscr{A}$ is applied that yields a new set of terms which may need further evaluation.

[16] A term u is closed modulo B w.r.t. Q iff either: (i) it does not contain defined function symbols of $\mathscr{D}$, or (ii) there exists a substitution θ and a (possibly renamed) $q \in Q$ such that $u =_B q\theta$, and the terms in θ are recursively Q-closed. For instance, given a defined binary symbol $\bullet$ that does not obey any structural axioms, the term $t = a \bullet (Z \bullet a)$ is closed w.r.t. $Q = \{a \bullet X, Y \bullet a\}$ or $\{X \bullet Y\}$, but it is not closed with Q being $\{a \bullet X\}$; however, it would be closed if $\bullet$ were commutative.

Steps (1) and (2) are iterated as long as new terms are generated until a fixpoint is reached, and the augmented, final set Q' of specialized calls is yielded. Note that the algorithm does not explicitly compute a partially evaluated equational theory. It does so implicitly; the desired partially evaluated equations E are generated as the set of *resultants* $t\sigma = t'$ associated with the derivations in the narrowing tree from the root $t \in Q'$ to the leaf t' with computed substitution σ, such that the closedness condition modulo B w.r.t. Q' is satisfied for all function calls that appear in the right-hand sides of the equations in E'. We assume the existence of a function $\textsc{GenTheory}(Q', (\Sigma, E \uplus B))$ that delivers the partially evaluated equational theory $\mathscr{E}' = (\Sigma, E' \uplus B)$ that is uniquely determined by Q' and the original equational theory $\mathscr{E} = (\Sigma, E \uplus B)$. Formally,

$$\textsc{GenTheory}(Q', \mathscr{E}) = (\Sigma, \{t\sigma = t' \mid t \in Q', t' \in \mathscr{U}(Q', \vec{\mathscr{E}}), t \text{ fV-narrows to } t' \text{ with computed substitution } \sigma\} \uplus B).$$

4.2 *The* NPER$^{\mathscr{U}}$ *Scheme for the Specialization of Rewrite Theories*

The specialization of the rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$ is achieved by partially evaluating the hosted equational theory $\mathscr{E} = (\Sigma, E \uplus B)$ w.r.t. the rules of R, which is done by using the partial evaluation procedure EQNPE$^{\mathscr{U}}$ of Section 4.1. By providing suitable unfolding (and abstraction) operators, different instances of the specialization scheme can be defined.

Before going into the details, let us summarize the advantages of the final, specialized rewrite theory $\mathscr{R}_{\mathsf{sp}} = (\Sigma_{\mathsf{sp}}, E_{\mathsf{sp}} \uplus B_{\mathsf{sp}}, R_{\mathsf{sp}})$ over the original $\mathscr{R} = (\Sigma, E \uplus B, R)$:

1. The transformed rewrite theory $\mathscr{R}_{\mathsf{sp}}$ generally performs more efficiently than $\mathscr{R}$ for two reasons. On the one hand, the partially evaluated equational theory $\mathscr{E}_{\mathsf{sp}} = (\Sigma_{\mathsf{sp}}, E_{\mathsf{sp}} \uplus B_{\mathsf{sp}})$ has been optimized to fulfill the needs of the host rewrite rules of R (that have been correspondingly transformed into R_{sp}). On the other hand, the specialization process may remove unnecessary equational axioms and rule conditions, thereby significantly speeding up the rewriting and narrowing computations in the specialized theory at the level of both equations and rules.
2. The specialization may cut down an infinite folding variant narrowing space to a finite one. Actually, the transformed equational theory $\mathscr{E}_{\mathsf{sp}}$ may have the FVP whilst the original one may not, so that symbolic analysis based on narrowing with R_{sp} modulo $\mathscr{E}_{\mathsf{sp}}$ is enabled, whereas narrowing with R modulo $\mathscr{E}$ is not.

The NPER$^{\mathscr{U}}$ procedure is outlined in Algorithm 1 and it consists of two phases.

Phase 1 (Partial Evaluation). It applies the EQNPE$^{\mathscr{U}}$ algorithm to specialize the equational theory $\mathscr{E} = (\Sigma, E \uplus B)$ w.r.t. a set Q of specialized calls that consists of all of the *maximal function calls* that appear in the $(\vec{E}, B)$-normalized version R' of the conditional rewrite rules of R, where $C{\downarrow_{\vec{E},B}}$ denotes $c_1{\downarrow_{\vec{E},B}} \wedge \ldots \wedge c_n{\downarrow_{\vec{E},B}}$ for any rule condition $C = c_1 \wedge \ldots \wedge c_n$. Given $\Sigma = (\mathscr{D} \uplus \mathscr{C})$, a maximal function call in a term t is any outermost subterm of t that is rooted by a defined function symbol of E. By $mcalls(R)$, we denote the set of all maximal calls in both the right-hand sides and the conditions of the rules of R. This phase produces the new set of specialized calls Q' from which the partial evaluation $\mathscr{E}' = (\Sigma', E' \uplus B')$ of $\mathscr{E}$ w.r.t. Q is unambigously derived by $\textsc{GenTheory}(Q', (\Sigma, E \uplus B))$.

Phase 2 (Compression). It consists of a refactoring transformation that computes a new, simplified rewrite theory $\mathscr{R}_{\mathsf{sp}} = (\Sigma_{\mathsf{sp}}, E_{\mathsf{sp}} \uplus B_{\mathsf{sp}}, R_{\mathsf{sp}})$ by taking as input the $(\vec{E}, B)$-normalized rewrite

Algorithm 1 Symbolic Specialization of Rewrite Theories NPER$^{\mathscr{U}}(\mathscr{R})$

Require:

A rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$, an unfolding operator $\mathscr{U}$

1: **function** NPER$^{\mathscr{U}}$ $(\mathscr{R})$

Phase 1. Partial Evaluation

2: $R' \leftarrow \{(\lambda \downarrow_{\vec{E},B}) \Rightarrow (\rho \downarrow_{\vec{E},B})$ *if* $C \downarrow_{\vec{E},B} \mid (\lambda \Rightarrow \rho$ *if* $C) \in R\}$

3: $Q \leftarrow mcalls(R')$

4: $Q' \leftarrow$ EQNPE$^{\mathscr{U}}((\Sigma, E \uplus B), Q)$

5: $(\Sigma', E' \uplus B') \leftarrow$ GENTHEORY$(Q', (\Sigma, E \uplus B))$

Phase 2. Compression

6: $\mathscr{R}_{ren} \leftarrow$ RENAME$((\Sigma, E \uplus B, R'), (\Sigma', E' \uplus B'), Q')$

7: $(\Sigma_{\mathsf{sp}}, E_{\mathsf{sp}} \uplus B_{\mathsf{sp}}, R_{\mathsf{sp}}) \leftarrow$ SIMPLIFY$(\mathscr{R}_{ren})$

8: **return** $(\Sigma_{\mathsf{sp}}, E_{\mathsf{sp}} \uplus B_{\mathsf{sp}}, R_{\mathsf{sp}})$

theory $\mathscr{R}' = (\Sigma, E \uplus B, R')$, the computed partially evaluated theory $\mathscr{E}' = (\Sigma', E' \uplus B')$, and the final set of specialized calls Q' from which $\mathscr{E}'$ derives.

Algorithm 2 Renaming algorithm

Require:

A rewrite theory $\mathscr{R}' = (\Sigma, E \uplus B, R')$, a partial evaluation $\mathscr{E}' = (\Sigma', E' \uplus B')$ of $(\Sigma, E \uplus B)$ w.r.t. a set of specialized calls Q

1: **function** RENAME$(\mathscr{R}', \mathscr{E}', Q)$

2: Let δ be an independent renaming for Q in

3: $E_{ren} \leftarrow \bigcup_{t \in Q} \{\delta(t)\theta = RN_{\delta}(t') \mid t\theta = t' \in E'\}$

4: $R_{ren} \leftarrow \{RN_{\delta}(\lambda) \Rightarrow RN_{\delta}(\rho)$ *if* $RN_{\delta}(C) \mid \lambda \Rightarrow \rho$ *if* $C \in R'\}$

5: $\Sigma_{ren} \leftarrow (\Sigma' \setminus \{f \mid f \text{ occurs in } ((E \uplus B) \setminus (E' \uplus B'))\}) \cup \{root(\delta(t)) \mid t \in Q\}$

6: $B_{ren} \leftarrow \{ax(f) \in B' \mid f \in \Sigma' \cap \Sigma_{ren}\}$

7: $\mathscr{R}_{ren} \leftarrow (\Sigma_{ren}, E_{ren} \uplus B_{ren}, R_{ren})$

8: **return** $\mathscr{R}_{ren}$

where

$$RN_{\delta}(t) = \begin{cases} c(\overline{RN_{\delta}(t_n)}) & \text{if } t = c(\overline{t_n}) \text{ with } c : \overline{s_n} \to s \in \Sigma \; s.t. \; c \in \mathscr{C}, \; ls(t) = s, \; n \geq 0 \\ \delta(u)\theta' & \text{if } \exists\theta, \exists u \in Q \text{ s.t. } t =_B u\theta \text{ and } \theta' = \{x \mapsto RN_{\delta}(x\theta) \mid x \in Dom(\theta)\} \\ t & \text{otherwise} \end{cases}$$

The compression phase consists of two subphases: *renaming* and *rule condition simplification*. The renaming (sub-)phase aims at yielding a much more compact equational theory $\mathscr{E}_{ren} = (\Sigma_{ren}, E_{ren} \uplus B_{ren})$ where unused symbols and unnecessary repetitions of variables are removed and equations of E_{ren} are simplified by recursively renaming all expressions that are Q'-closed modulo B by using an independent renaming function that is derived from the set of specialized calls Q'. Formally, an *independent renaming* δ for a set of terms T and a signature Σ is a mapping from terms to terms, which is defined as follows. For each t of sort s in T with $root(t) = f$, $\delta(t) = f_t(\overline{x_n : s_n})$, where $\overline{x_n}$ are the distinct variables in t in the order of their first occurrence and $f_t : \overline{s_n} \to s$ is a new function symbol that does not occur in Σ or T and is different from the root symbol of any other $\delta(t')$, with $t' \in T$ and $t' \neq t$. Abusing notation, we let $\delta(T)$ denote the set $\{\delta(t) \mid t \in T\}$ for a given set of terms T.

Example 5

Consider the rewrite theory in Example 1 together with the set of specialized calls

$$Q = \{\texttt{dec(enc(M:Data,s(s(0))))},\texttt{enc(enc(M,K1:Nat),K2:Nat)}\}$$

An independent renaming δ for Q is given by

$$\delta = \{\texttt{dec(enc(M:Data,s(s(0))))} \mapsto \texttt{f0(M:Data)}, \\ \texttt{enc(enc(M:Data,K1:Nat),K2:Nat)} \mapsto \texttt{f1(M:Data,K1:Nat,K2:Nat)}\}$$

where `f0: Data -> Data` and `f1: Data Nat Nat -> Data` are new function symbols.

Renaming is performed by the RENAME function given in Algorithm 2.

Essentially, the RENAME function recursively computes (by means of the function RN_δ) a new equation set E_{ren} by replacing each call in E' by a call to the corresponding renamed function according to δ. Furthermore, a new rewrite rule set R_{ren} is also produced by consistently applying RN_δ to the conditional rewrite rules of R'. Formally, each conditional rewrite rule $\lambda \Rightarrow \rho$ *if* C in R' is transformed into the rewrite rule $RN_\delta(\lambda) \Rightarrow RN_\delta(\rho)$ *if* $RN_\delta(C)$, in which every maximal function call t in the rule is recursively renamed according to the independent renaming δ. The renaming algorithm also computes the specialized signature Σ_{ren} and restricts the set B' to those axioms obeyed by the function symbols in $\Sigma' \cap \Sigma_{ren}$, delivering the rewrite theory $(\Sigma_{ren}, E_{ren} \uplus B_{ren}, R_{ren})$.

Note that, while the independent renaming suffices to rename the left-hand sides of the equations in E' (since they are mere instances of the specialized calls), the right-hand sides are renamed by means of the auxiliary function RN_δ, which recursively replaces each call in the given expression by a call to the corresponding renamed function (according to δ).

Algorithm 3 Simplification algorithm

Require:

A rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$, with equational theory $\mathscr{E} = (\Sigma, E \uplus B)$

1: **function** SIMPLIFY($\mathscr{R}$)
2: $R' \leftarrow SimpConds(\mathscr{R})$
3: $E' \leftarrow E \setminus \{l = r \mid root(l)$ does not occur in either R' or $(E \setminus \{l = r\})\}$
4: $\Sigma' \leftarrow \{f \in \Sigma \mid f$ occurs in R' or $E'\}$
5: $B' = \{ax(f) \in B \mid f \in \Sigma'\}$
6: **return** $(\Sigma', E' \uplus B', R')$

The simplification subphase is undertaken by the SIMPLIFY procedure of Algorithm 3 that further refines the rewrite theory $\mathscr{R}_{ren}$. Indeed, the partial evaluation process may produce specialized rewrite rules that include conditions that can be safely removed without changing the original program semantics. This transformation is implemented by function $SimpConds(\mathscr{R})$ and may have a huge impact on program efficiency since Maude executes unconditional rules much faster than conditional rules because the evaluation of rule conditions is notoriously speculative in Maude. This is because, besides the fact that many equational matchers for the rule conditions may exist, solving rewriting conditions requires non-deterministic search (Serbanuta and Rosu, 2006; Clavel et al., 2020). The rule simplification function $SimpConds(\mathscr{R})$ is left generic in Algorithm 3 and can be specialized for different kinds of rewrite theories.

After simplifying rule conditions, there might exist spurious equations $l = r$ in E_{ren} such that

$root(l)$ does not occur in either the simplified rewrite rules of R' or in $(E_{ren} \setminus \{l = r\})$. Hence, SIMPLIFY removes such equations, and the theory signature is cleaned up by deleting any operators (and their corresponding axioms) that do not occur in the transformed equations and rules.

A concrete implementation $SimpConds^*(\mathscr{R})$ of the generic rule simplication function $SimpConds(\mathscr{R})$ of Algorithm 3 that works for the theories that are considered in this article is given in the following definition:

Definition 3 (*Rule Condition Simplification*)
Let $\mathscr{R} = (\Sigma, E \uplus B, R)$ be a rewrite theory with an equational theory $\mathscr{E} = (\Sigma, E \uplus B)$. Then, $SimpConds^*(\mathscr{R})$ is given by the set

$$\bigcup_{\lambda \Rightarrow \rho \ if \ C \in R} \{\lambda \Rightarrow \rho \ if \ sCond_{\mathscr{E}}(\lambda, \rho, C, nil) \mid \text{no condition in } C \text{ is } false\}$$

where the definition of the auxiliary function $sCond_{\mathscr{E}}(\lambda, \rho, C, \sharp C)$ is given in Figure 4.

$$sCond_{\mathscr{E}}(\lambda,\rho,C,\sharp C) = \begin{cases} \sharp C & \text{if (1) } C = nil \\ sCond_{\mathscr{E}}(\lambda,\rho,C'',\sharp C) & \text{if (2) } C = (C' \wedge C''), \text{ with } C' \text{ being a maximal,} \\ & \quad \text{non-empty subsequence of C s.t. either} \\ & \quad \text{(a) } C' \text{ consists of expressions of the form} \\ & \qquad t = t' \text{ (or } t := t' \text{ or } t \Rightarrow t'\text{) s.t. } t =_B t'\text{; or} \\ & \quad \text{(b) } C' \text{ consists of } n \text{ equations } t_i = t'_i, \\ & \qquad \text{with } t_i \neq_B t'_i \text{ for } i = 1,\ldots,n, \\ & \qquad \mathscr{E} \text{ has the FVP,} \\ & \qquad Var(C') \cap Var(\lambda \Rightarrow \rho \ if \ \sharp C \wedge C'') = \emptyset, \\ & \qquad \text{and exists } \theta \text{ s.t. } t_i\theta =_{\mathscr{E}} t'_i\theta \text{ for } i = 1,\ldots,n \\ sCond_{\mathscr{E}}(\lambda,\rho,C',\sharp C) & \text{if (3) } C = (c \wedge C'), \text{ with } c \text{ being either} \\ & \quad \text{(a) } t_1 = t_2 \text{(resp. } t_1 := t_2 \text{ or } t_2 \Rightarrow t_1\text{) s.t.} \\ & \qquad t_1 \ B\text{-unifies with } t_2 \text{ (resp. } t_2 \text{ is a } B\text{-instance of } t_1\text{),} \\ & \qquad \text{and } (Var(c) \cap Var(\lambda \Rightarrow \rho \ if \ \sharp C \wedge C') = \emptyset\text{; or} \\ & \quad \text{(b) an equation } t_1 = t_2, \text{with } t_1 \neq t_2, \\ & \qquad \mathscr{E} \text{ has the CFVP}, CV_{\mathscr{E}}(t_1) = CV_{\mathscr{E}}(t_2), \text{and} \\ & \qquad \texttt{narrowing} \notin att(\lambda \Rightarrow \rho \ if \ C) \\ sCond_{\mathscr{E}}(\lambda,\rho,C',\sharp C \wedge c) & \text{if (4) } C = (c \wedge C'), \text{ and Cases 2) and 3) do not apply} \end{cases}$$

Fig. 4. Definition of the *sCond* function.

Roughly speaking, for any conditional rule $\lambda \Rightarrow \rho \ if \ C$ in R such that none of the conditions in C is false[17], the auxiliary function $sCond_{\mathscr{E}}$ gets rid of

- (Case 2) those maximal subsequences[18] C' of C such that

  (2.a) all of its elements are tautological expressions of the form $t = t$, $t' := t'$, or $t'' \Rightarrow t''$ (so that t and t' unify with the empty substitution by using fV-narrowing). Note that this optimization cannot be done by standard Maude Boolean simplification, which does not apply to non-ground, rule condition expressions; or

[17] This is because a rule condition that is generally satisfiable (e.g., $f(Y) = g(Y)$) can specialize into a contradiction w.r.t. the equational theory that is considered (e.g., $\mathscr{E} = \{f(X) = a,\ g(X) = b)\}$.

[18] We consider ordered subsequences of not necessarily consecutive elements of C.

(2.b) C' consists of n equations of the form $t_i = t'_i$, with $t_i \neq t'_i$, such that there exists an $\mathscr{E}$-unifier θ for all of the equations in the sequence. To safely perform this optimization two extra conditions must hold: (i) $\mathscr{E}$ must have the FVP to make $\mathscr{E}$-unification decidable; (ii) the variables occurring in C' must not occur in the rest of the rule to be simplified. The constraint on the variables in C' is required to preserve the narrowing semantics since we must ensure that removing the equationally satisfiable sequence C' does not affect the overall computation.

- (Case 3)

(3.a) those equations (resp. matching equations $t_1 := t_2$ and rewrite expressions $t_2 \Rightarrow t_1$) such that t_1 and t_2 are B-unifiable (resp. t_2 is an instance of t_1 modulo B), and the variables in t_1 and t_2 do not occur in the rest of the rule.

(3.b) those equations $t_1 = t_2$, with $t_1 \neq t_2$, such that the set of (most general) *constructor variants* (i.e., equational variants that only include constructor symbols and variables (Meseguer, 2020)) of t_1 and t_2 are the same (in symbols, $CV_{\mathscr{E}}(t_1) = CV_{\mathscr{E}}(t_2)$), which implies that t_1 and t_2 compute the very same set of constructor values and answers. However, since variables in $t_1 = t_2$ are not required to be apart from the variables in the rest of the rule, this simplification only applies when the rule to be simplified is not enabled for narrowing but only for rewriting. Otherwise, removing the condition $t_1 = t_2$ might not preserve the overall computed answers. Furthermore, to ensure that constructor variant sets are computable, two extra conditions on $\mathscr{E}$ are required (Meseguer, 2020): (i) $\mathscr{E}$ must have the FVP, and (ii) $\mathscr{E}$ must be *sufficiently complete modulo axioms* (i.e., it specifies total functions). We say that equational theories satisfying (i) and (ii) have the *constructor finite variant property*[19] (CFVP).

Note that Cases 1 and 4 of the $sCond_{\mathscr{E}}$ function are simply used to incrementally process the rule condition to be simplified, yet they do not remove any expression.

The renaming subphase of the compression transformation can be seen as a qualified extension to the Rewriting Logic setting of the classical renaming[20] that is applied in the partial evaluation of logic programs and implemented in well-known partial deduction tools such as Logen and ECCE (Leuschel et al., 2006), which greatly lessens the program structure. The rule condition simplification transformation is novel and further contributes to lighten the specialized rewrite theory. Although more aggressive rule simplification is possible, our transformation is relatively straightforward and inexpensive. Actually, neither it propagates answers from the removed expressions to the rest of the rule, nor does it require replicating rules due to several equational unifiers.

Let us illustrate the rule condition simplification given by $SimpConds^*(\mathscr{R})$ by means of two examples.

Example 6
Let $\mathscr{E} = (\Sigma, E \uplus B)$ be the equational theory of Figure 2 modeling the *Caesar* cypher, and let $\mathscr{R}$ be a rewrite theory $(\Sigma, E \uplus B, \{r\})$, where r is a conditional rewrite rule whose condition C is

[19] The (unconditional) NPER$^{\mathscr{U}}$ scheme of (Alpuente et al., 2021b) is instantiated in (Alpuente et al., 2021a) for theories that satisfy the CFVP.

[20] Renaming can also be presented as a new definition in Burstall-Darlington style unfold-fold-new definition transformation frameworks (Burstall and Darlington, 1977). The removal of unnecessary structure by renaming is discussed in (Alpuente et al., 1998a; Gallagher and Bruynooghe, 1990; Gallagher, 1993).

$\mathtt{enc(a,s(K:Nat))} = \mathtt{enc(b,K:Nat)}$. By applying the partial evaluation algorithm NPER$^{\mathscr{U}}$ to $\mathscr{R}$, first the rule condition C is normalized in $\vec{E}$ (modulo B) into

$$\mathtt{toSym(e(s(0),K))} = \mathtt{toSym(e(s(0),K))}.$$

After partially evaluating the equational theory, the rule condition is transformed into a trivial equation `f(K:Nat) = f(K:Nat)`, with independent renaming

$$\mathtt{toSym(e(s(0),K:Nat))} \mapsto \mathtt{f(K:Nat)}$$

for the call `toSym(e(s(0), K:Nat))`. Then, the trivial equation `f(K:Nat) = f(K:Nat)` is removed (via Case 2.a of the *sCond* function), thereby delivering an unconditional version of r as final outcome.

Example 7

Let $\mathscr{E} = (\Sigma, E \uplus B)$ be the equational theory of Figure 2 modeling the *Caesar* cypher, and let $\mathscr{R}$ be a rewrite theory $(\Sigma, E \uplus B, \{r\})$, where r is a conditional rewrite rule whose condition C is $\mathtt{enc(S:Symbol,s(0))} = \mathtt{enc(enc(S:Symbol,0),s(0))}$. We note that this condition does not generally hold for every symbol S in an arbitrary encryption function, but it certainly does when considering the implementation of the enc function defined by the equational theory of Example 1. We also assume that the variable S only appears in the condition C of the rule r. By specializing in Presto the two (already normalized) maximal function calls in rule r, `enc(S:Symbol,s(0))` and `enc(enc(S:Symbol,0),s(0))`, we get an equational theory $\mathscr{E}_{ren}$ containing the following equations

```
eq f0(a) = b [ variant ] .   eq f1(a) = b [ variant ] .
eq f0(b) = c [ variant ] .   eq f1(b) = c [ variant ] .
eq f0(c) = a [ variant ] .   eq f1(c) = a [ variant ] .
```

where `f0` and `f1` are two new operators that are introduced to rename the initial calls `enc(S:Symbol,s(0))` and `enc(enc(S:Symbol,0),s(0))`, and actually denote the very same encryption function.

Furthermore, the condition C is renamed as

$$\mathtt{f1(S:Symbol)} = \mathtt{f0(S:Symbol)}.$$

Since $\mathscr{E}_{ren}$ has the finite variant property, equational unification of $\mathtt{f1(S:Symbol)}$ and $\mathtt{f0(S:Symbol)}$ is decidable and delivers three unifiers, namely, $\{\mathtt{S} \mapsto \mathtt{a}\}$, $\{\mathtt{S} \mapsto \mathtt{b}\}$ and $\{\mathtt{S} \mapsto \mathtt{c}\}$. Hence, the condition `f1(S:Symbol) = f0(S:Symbol)` can be safely removed from the specialized version of r (via Case 2.b of the $sCond_{\mathscr{E}}$ function), since variable S does not appear elsewhere in the rule.

An example of condition simplification that exploits Case 3.b of the $sCond_{\mathscr{E}}$ function is shown in Example 10.

Given the rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$ and its specialization $\mathscr{R}_{\mathsf{sp}}$ =NPER$^{\mathscr{U}}(\mathscr{R})$, all of the executability conditions that are satisfied by $\mathscr{R}$ are also satisfied by $\mathscr{R}_{\mathsf{sp}} = (\Sigma_{\mathsf{sp}}, E_{\mathsf{sp}} \uplus B_{\mathsf{sp}}, R_{\mathsf{sp}})$, including the fact that R_{sp} is coherent w.r.t. E_{sp} modulo B_{sp}. This is proved in Theorem 1 of (Alpuente et al., 2021b) for the case of unconditional rewrite theories such that the left-hand sides of the rules in R are $(\vec{E},B)$-*strongly irreducible*, and straightforwardly extends to conditional rewrite theories since 1) the conditional version of NPER$^{\mathscr{U}}$ only augments the initial set Q of

specialized calls with the maximal calls in the rule conditions, while the core equational partial evaluation algorithm is kept untouched; and 2) only conditions that do not modify any semantic property are removed by the rule condition simplification that is performed by the compression phase.

Similarly, the strong semantic correspondence between both narrowing and rewriting computations in $\mathscr{R}$ and $\mathscr{R}_{\mathsf{sp}}$ was proved in Theorem 2 of (Alpuente et al., 2021b) for unconditional rewrite theories and immediately extends to the conditional case for the same reasons and under the above-mentioned strong irreducibility requirement.

5 Instantiating the Specialization Scheme for Rewrite Theories

In the following, we outline two instances of the NPER$^{\mathscr{U}}$ scheme that are obtained by choosing two distinct implementations of the unfolding operator $\mathscr{U}$. More precisely, we present the unfolding operators $\mathscr{U}_{FVP}$ and $\mathscr{U}_{\overline{FVP}}$ that allow $\mathscr{R}$ to be specialized when $\mathscr{E}$ is (respectively, is not) a finite variant theory. Both operators have been implemented in Presto.

With regard to global control, Presto implements the same abstraction operator for both scheme instantiations. This abstraction operator is described in (Alpuente et al., 2020c) and relies on the equational least general generalization algorithm of (Alpuente et al., 2021c) that improves our original calculus in (Alpuente et al., 2014b) by providing a finitary, minimal and complete set of order-sorted generalizers modulo any combinations of associativity and/or commutativity and/or identity axioms for any unification problem. This property holds even when the theory contains function symbols with an arbitrary number n of distinct identity elements, $e_1,\ldots,e_n$, provided their respective *least sorts* are incomparable at the kind level. In symbols, $[ls(e_i)] \neq [ls(e_j)]$ for any $i \neq j$, with $i, j \in 1,\ldots,n$. Theories that satisfy this property are called *U-tolerant* theories. Note that the traditional one-unit condition that was implicitly assumed in (Alpuente et al., 2014b) implies U-tolerance.

Example 8
Consider the following Maude functional module that defines lists and multisets of natural numbers:

```
fmod LIST+MSET-NO-U-tolerant is
 sorts Nat List MSet Top .
 subsorts Nat < List < Top .
 subsorts Nat < MSet < Top .
 op nil : -> List [ctor] .
 op _;_ : List List -> List [ctor assoc id: nil] .
 op null : -> MSet [ctor] .
 op _,_ : MSet MSet -> MSet [ctor assoc comm id: null] .
 op 0 : -> Nat [ctor] .
 op s : Nat -> Nat [ctor] .
endfm
```

where the `ctor` declarations specify that all of these operators are *data constructors*. Note that, since `Nat < List` and `Nat < MSet`, 0 is both a list of length one and a singleton multiset, but the list `0 ; s(0) ; s(s(0))` and the multiset `0 , s(0) , s(s(0))` have incomparable least sorts `List` and `MSet`. Moreover these are also the least sorts of `nil` and `null`, respectively. Therefore, this theory is not *U-tolerant*, since $[ls(\texttt{nil})] = [ls(\texttt{null})] = \texttt{Top}$.

Automated ways to achieve U-tolerance in a *semantics-preserving manner* are discussed in (Alpuente et al., 2021c).

Case 1: $\mathscr{E}$ is not a finite variant theory. When $\mathscr{E}$ is not a finite variant theory, the fV-narrowing strategy may lead to the creation of an infinite fV-narrowing tree for some specialized calls in Q. In this case, an *equational order-sorted extension* $\breve{\trianglelefteq}_B$ (Alpuente et al., 2020b) of the classical homeomorphic embedding relation[21] $\trianglelefteq$ is used to detect the risk of non-termination. Roughly speaking, a homeomorphic embedding relation is a structural preorder under which a term t is greater than or equal to another term t' (i.e., t embeds t'), written as $t \trianglerighteq t'$, if t' can be obtained from t by deleting some parts, e.g., $s(s(X+Y)*(s(X)+Y))$ embeds $s(Y*(X+Y)))$. Embedding gets much less intuitive when dealing with axioms. For example, let us write natural numbers in classical decimal notation and the addition operator + for natural numbers in prefix notation. Due to associativity and commutativity of symbol +, there can be many equivalent terms such as $+(+(1,2),+(3,0))$ and $+(+(1,+(3,2)),0)$, all of which are internally represented in Maude by the flattened term $+(0,1,2,3)$. Actually, flattened terms like $+(1,0,2,3)$ can be further simplified into a single *canonical representative* $+(0,1,2,3)$, hence $+(1,2)\breve{\trianglelefteq}_B+(0,1,2,3)$ and also $+(2,1)\breve{\trianglelefteq}_B+(0,1,2,3)$. In contrast, if $+$ was associative (and not commutative), then $+(2,1)\breve{\not\trianglelefteq}_B+(1,0,3,2)$. A more detailed description that takes into account the extra complexity given by sorts and subsorts can be found in (Alpuente et al., 2020c).

When iteratively computing a sequence $t_1,t_2,\ldots,t_n$, finiteness of the sequence can be guaranteed by using the well-quasi order relation $\breve{\trianglelefteq}_B$ (Alpuente et al., 2020c) as a whistle (Leuschel, 1998): whenever a new expression t_{n+1} is to be added to the sequence, we first check whether t_{n+1} embeds any of the expressions already in the sequence. If that is the case, we say that the homeomorphic embedding test $\breve{\trianglelefteq}_B$ whistles, i.e., it has detected (potential) non-termination and the computation has to be stopped. Otherwise, t_{n+1} can be safely added to the sequence and the computation can proceed.

The operator $\mathscr{U}_{\overline{FVP}}$ implements such an embedding check to guarantee the termination of the unfolding phase. Specifically, the generation of a fV-narrowing tree fragment is stopped when each fV-narrowing derivation (i.e., each branch of the fV-narrowing tree) is stopped because a term is reached that is either unnarrowable or it embeds (w.r.t. $\breve{\trianglelefteq}_B$) a previously narrowed term in the same derivation.

Example 9
Consider the specific instance of the rewrite theory of Example 1 where servers and clients use a pre-shared fixed key `K=s(0)` but messages can have any length, in contrast to Example 2.

Let $\mathscr{R}=(\Sigma,E\uplus B,R)$ be such a rewrite theory, where $\mathscr{E}=(\Sigma,E\uplus B)$ is the equational theory of $\mathscr{R}$. In $\mathscr{E}$, the fV-narrowing trees associated with encryption and decryption functionality may be infinite since $\mathscr{E}$ does not have the FVP, as shown in Section 3. For instance, terms of the form $(\mathtt{t}_1 \ldots \mathtt{t}_n\ \mathtt{enc}(\mathtt{M}',\mathtt{s}(0)))$ derive from $\mathtt{enc}(\mathtt{M},\mathtt{k1})$ by fV-narrowing, where $\mathtt{enc}(\mathtt{M}',\mathtt{s}(0))$ can be further narrowed to unravel an unlimited sequence of identical terms modulo renaming. Nonetheless, equational homeomorphic embedding detects this non-terminating behavior since $\mathtt{enc}(\mathtt{M}',\mathtt{s}(0))$ embeds $\mathtt{enc}(\mathtt{M},\mathtt{s}(0))$.

By using the unfolding operator $\mathscr{U}_{\overline{FVP}}$, the first phase of the NPER$^{\mathscr{U}_{\overline{FVP}}}(\mathscr{R})$ algorithm first normalizes the rewrite rules in R. The normalization process produces a new set of rules R' where the `reply` and `rec` rules are left unchanged, while the `req` rule is simplified into

```
crl [req] : < C : Cli | server : S ,
                        data : M ,
```

[21] A notion of homeomorphic embedding for a typed language was presented in (Albert et al., 2009).

```
                key : s(0) ,
                status : mt >
        =>

        < C : Cli | server : S ,
                    data : M ,
                    key : s(0) ,
                    status : mt >
        ( S <- { C , enc(M,s(0)) } ) if true = true /\ enc(M,s(0)) = enc(enc(M,0),s(0)).
```

because of the normalization of the equational condition `(s(0) < len) = true` and the instantiation of the condition `enc(M,s(0)) = enc(enc(M,0),s(0))` to the specific case where `K` is equal to `s(0)`.

Subsequently, the specialization algorithm computes the initial set

$$\mathtt{Q} = \{\mathtt{enc}(\mathtt{M},\mathtt{s}(0)), \mathtt{dec}(\mathtt{M},\mathtt{s}(0)), \mathtt{enc}(\mathtt{enc}(\mathtt{M},0),\mathtt{s}(0))\}$$

of the (normalized) maximal function calls in R', where `M` is a variable of sort `Data`, and the equational theory $\mathscr{E}$ is partially evaluated by EQNPE$^{\mathscr{U}_{\overline{FVP}}}$ w.r.t. `Q`. During the partial evaluation process, $\mathscr{U}_{\overline{FVP}}$ only unravels finite fragments of the fV-narrowing trees that are rooted by the specialized calls, and the final set Q' of specialized calls is

$$\{\mathtt{dec}(\mathtt{X}:\mathtt{Data},\mathtt{s}(0)), \mathtt{enc}(\mathtt{X}:\mathtt{Data},\mathtt{s}(0)), \mathtt{enc}(\mathtt{enc}(\mathtt{X}:\mathtt{Data},0),\mathtt{s}(0)),$$
$$\mathtt{toSym}([\mathtt{toNat}(\mathtt{Y}:\mathtt{Symbol}) < \mathtt{s}(\mathtt{s}(0)), \mathtt{s}(\mathtt{toNat}(\mathtt{Y}:\mathtt{Symbol})), 0]), \mathtt{toSym}(\mathtt{toNat}(\mathtt{Y}:\mathtt{Symbol})),$$
$$\mathtt{toSym}(\mathtt{unshift}(\mathtt{toNat}(\mathtt{Y}:\mathtt{Symbol})))\}.$$

```
eq dec(a, s(0)) = c [ variant ] .
eq dec(b, s(0)) = a [ variant ] .
eq dec(c, s(0)) = b [ variant ] .
eq dec(S:Symbol M:Data, s(0)) = toSym(unshift(toNat(S:Symbol)))
                                dec(M:Data, s(0)) [ variant ] .
eq enc(a, s(0)) = b [ variant ] .
eq enc(b, s(0)) = c [ variant ] .
eq enc(c, s(0)) = a [ variant ] .
eq enc(S:Symbol M:Data, s(0)) = toSym([toNat(S:Symbol) < s(s(0)),s(toNat(S:Symbol)),0])
                                enc(M:Data, s(0)) [ variant ] .
eq enc(enc(a, 0), s(0)) = b [ variant ] .
eq enc(enc(b, 0), s(0)) = c [ variant ] .
eq enc(enc(c, 0), s(0)) = a [ variant ] .
eq enc(enc(S:Symbol M:Data, 0), s(0)) = toSym([toNat(toSym(toNat(S:Symbol))) < s(s(0)),
                                        s(toNat(toSym(toNat(S:Symbol)))),0])
                                        enc(enc(M:Data, 0), s(0)) [ variant ] .
eq toSym([toNat(a) < s(s(0)),s(toNat(a)),0]) = b [ variant ] .
eq toSym([toNat(b) < s(s(0)),s(toNat(b)),0]) = c [ variant ] .
eq toSym([toNat(c) < s(s(0)),s(toNat(c)),0]) = a [ variant ] .
eq toSym(toNat(a)) = a [ variant ] .
eq toSym(toNat(b)) = b [ variant ] .
eq toSym(toNat(c)) = c [ variant ] .
eq toSym(unshift(toNat(a))) = c [ variant ] .
eq toSym(unshift(toNat(b))) = a [ variant ] .
eq toSym(unshift(toNat(c))) = b [ variant ] .
```

Fig. 5. NPER$^{\mathscr{U}_{\overline{FVP}}}$ Phase 1: Partial evaluation of $\mathscr{E}$ w.r.t. `Q` output by Presto

The resulting partial evaluation $\mathscr{E}'$ of $\mathscr{E}$, which is generated from Q', is given in Figure 5. The

```
eq f0(a) = c [variant] .    eq f0(b) = a [variant] .    eq f0(c) = b [variant] .
eq f2(a) = b [variant] .    eq f2(b) = c [variant] .    eq f2(c) = a [variant] .
eq f1(a) = b [variant] .    eq f1(b) = c [variant] .    eq f1(c) = a [variant] .
eq f3(a) = b [variant] .    eq f3(b) = c [variant] .    eq f3(c) = a [variant] .
eq f4(a) = a [variant] .    eq f4(b) = b [variant] .    eq f4(c) = c [variant] .
eq f5(a) = c [variant] .    eq f5(b) = a [variant] .    eq f5(c) = b [variant] .

eq f0(S:Symbol M:Data) = f5(S:Symbol) f0(M:Data) [variant] .
eq f1(S:Symbol M:Data) = f3(S:Symbol) f1(M:Data) [variant] .
eq f2(S:Symbol M:Data) = f3(f4(S:Symbol)) f2(M:Data) [variant] .
```

Fig. 6. NPER$^{\mathcal{U}_{FVP}}$ Phase 2: Renamed theory $\mathcal{E}_{ren}$ output by Presto

second phase (compression) renames $\mathcal{E}'$ into the equational theory $\mathcal{E}_{ren}$ of Figure 6 by computing the following renaming for the specialized calls:

$$\begin{array}{l}
\texttt{dec}(\texttt{X:Data},\texttt{s}(0)) \mapsto \texttt{f0}(\texttt{X:Data}) \\
\texttt{enc}(\texttt{X:Data},\texttt{s}(0)) \mapsto \texttt{f1}(\texttt{X:Data}) \\
\texttt{enc}(\texttt{enc}(\texttt{X:Data},0),\texttt{s}(0)) \mapsto \texttt{f2}(\texttt{X:Data}) \\
\texttt{toSym}([\texttt{toNat}(\texttt{Y:Symbol}) < \texttt{s}(\texttt{s}(0)),\texttt{s}(\texttt{toNat}(\texttt{Y:Symbol})),0]) \mapsto \texttt{f3}(\texttt{Y:Symbol}) \\
\texttt{toSym}(\texttt{toNat}(\texttt{Y:Symbol})) \mapsto \texttt{f4}(\texttt{Y:Symbol}) \\
\texttt{toSym}(\texttt{unshift}(\texttt{toNat}(\texttt{Y:Symbol}))) \mapsto \texttt{f5}(\texttt{Y:Symbol})
\end{array}$$

which eliminates complex nested calls and redundant arguments in $\mathcal{E}_{ren}$ computations.

It is worth noting that the resulting specialization $\mathcal{E}_{ren}$ provides a highly optimized version of $\mathcal{E}$ for the arbitrarily fixed key `s(0)`, where both functional and structural compression are achieved. Specifically, data structures in $\mathcal{E}$ for natural numbers and their associated operations for message encryption and decryption are totally removed from $\mathcal{E}_{ren}$. Also, code reuse is automatically achieved in $\mathcal{E}_{ren}$. Indeed, the obtained specialization reuses the renamed specialized call `f3(X:Data)` in the specialization of both `enc(X:Data,s(0))` (given by `eq f1(S:Symbol M:Data) = f3(S:Symbol) f1(M:Data) [variant]`) and `enc(enc(X:Data,0),s(0))` (given by `eq f2(S:Symbol M:Data) = f3(f4(S:Symbol)) f2(M:Data) [variant]`).

Note that the `_+_` operator, together with its associative and commutative axioms, disappears from $\mathcal{E}_{ren}$, thereby avoiding expensive matching operations modulo axioms. This transformation power cannot be achieved by existing functional, logic, or functional-logic partial evaluators for a language that implements native matching modulo axioms. Decryption (resp., encryption) in $\mathcal{E}_{ren}$ is now the direct mapping `f0` (resp., `f1`) that associates messages to their corresponding decrypted (resp. encrypted) counterparts, avoiding a huge amount of computation in the profuse domain of natural numbers. The computed renaming is also applied to $\mathcal{R}$ by respectively replacing, into the rewrite rules of $\mathcal{R}$, the maximal function calls `dec(M,s(s(0))`, `enc(M,s(s(0))`, and `enc(enc(M,0),s(0))` with `f0(M)`, `f1(M)`, and `f2(M)`, respectively. This allows the renamed rewrite rules to be able to access the new specialized encryption and decryption functionality provided by $\mathcal{E}_{ren}$. Finally, condition simplification is run, thereby removing the trivial condition `true = true` from the renamed version of the `req` rule and delivering the set of specialized rules of Figure 7 as outcome. Note that condition simplification is not able to remove condition `f1(M) = f2(M)` from the specialized rewrite rule `req-sp` since the partially evaluated equational theory $\mathcal{E}_{ren}$ does not have the FVP and thus the SIMPLIFY Algorithm can only remove trivial rule conditions.

```
crl [req-sp] : < C : Cli | server : S ,
                          data : M ,
                          key : s(0) ,
                          status : mt >

               =>

               < C : Cli | server : S ,
                              data : M ,
                              key : s(0) ,
                              status : mt >
               ( S <- { C , f1(M) } ) if  f1(M) = f2(M) .

rl [reply-sp] : < S : Serv | key : s(0) >
                (S <- {C,M})

                =>

                < S : Serv | key : s(0) >
                (C <- {S, f0(M)}) .

rl [rec-sp] : < C : Cli | server : S ,
                            data : M ,
                            key : s(0) ,
                            status : mt >
              (C <- {S,M})

              =>

              < C : Cli | server : S ,
                            data : M ,
                            key : s(0) ,
                            status: success >
```

Fig. 7. NPER$^{\mathcal{U}_{\overline{FVP}}}$ Phase 2: Specialized rewrite rules output by Presto

Example 9 shows that a great degree of simplification can be achieved by the specialization technique of Presto for theories that do not have the FVP. Furthermore, in many cases, the algorithm NPER$^{\mathcal{U}_{\overline{FVP}}}$ is also able to transform an equational theory that does not meet the FVP into a specialized one that does. This typically happens when the function calls to be specialized can only be unfolded a finite number of times. Let us see an example.

Example 10
Consider a slight variant of the protocol theory of Example 9 in which messages consist of one single symbol instead of arbitrarily long sequences of symbols. This variant can be obtained by simply modifying the sort of the messages from Data to Symbol in the protocol rewrite rules. In this scenario, the set of (normalized) maximal function calls becomes

$$\mathtt{Q} = \{\mathtt{enc}(\mathtt{M},\mathtt{s}(0))\downarrow_{\tilde{E},B}, \mathtt{dec}(\mathtt{M},\mathtt{s}(0))\downarrow_{\tilde{E},B}, \mathtt{enc}(\mathtt{enc}(\mathtt{M},0),\mathtt{s}(0))\downarrow_{\tilde{E},B}\}$$

where M is a variable of sort Symbol. The rewrite theory can be automatically specialized by Presto for this use case by using NPER$^{\mathcal{U}_{\overline{FVP}}}$. The intermediate, partially evaluated equational theory $\mathcal{E}_{ren}$ is shown in Figure 8.

The obtained specialization gets rid of the associative data structure that is needed to build messages of arbitrary size since only one-symbol messages are allowed in the specialized program. Also, note that $\mathcal{E}_{ren}$ clearly meets the FVP because it specifies three non-recursive functions (namely, f0, f1, f2), which all work over the finite domain {a,b,c}. Additionally, $\mathcal{E}_{ren}$

```
eq f0(a) = c [variant] .     eq f0(b) = a [variant] .     eq f0(c) = b [variant] .
eq f1(a) = b [variant] .     eq f1(b) = c [variant] .     eq f1(c) = a [variant] .
eq f2(a) = b [variant] .     eq f2(b) = c [variant] .     eq f2(c) = a [variant] .
```

Fig. 8. Equations of the specialized equational theory for one-symbol messages and key `s(0)`.

is sufficiently complete (indeed, `f1`, `f2`, and `f3` are total functions that respectively implement the behaviour of the calls `dec(M,s(0))`, `enc(M,s(0))` and `enc(enc(M,0),s(0))` for each $\mathtt{M} \in \{\mathtt{a},\mathtt{b},\mathtt{c}\}$). Therefore $\mathscr{E}_{\mathsf{ren}}$ also has the CFVP, which means that the SIMPLIFY algorithm can fully apply its simplification strategy to the (renamed) specialized conditional rule for protocol request

```
crl [req-ren] : < C : Cli | server : S ,
                            data : M ,
                            key : s(0) ,
                            status : mt >
                =>
                < C : Cli | server : S ,
                            data : M ,
                            key : s(0) ,
                            status : mt >
                ( S <- { C , f1(M) } ) if  true = true /\ f1(M) = f2(M) .
```

This rule is completely deconditionalized by removing both the trivial condition `true = true` (as in Example 9) and the condition `f1(M) = f2(M)`. The latter condition is removed by applying Case 3.b of the $sCond_{\mathscr{E}}$ function since the `req-ren` rule has not the `narrowing` label and `f1(M)` and `f2(M)` have the same (most general) constructor variants in $\mathscr{E}_{ren}$:

$$CV_{\mathscr{E}_{ren}}(\mathtt{f1(M)}) = \{(\{\mathtt{M} \mapsto \mathtt{a}\},\mathtt{b}),(\{\mathtt{M} \mapsto \mathtt{b}\},\mathtt{c}),(\{\mathtt{M} \mapsto \mathtt{c}\},\mathtt{a})\} = CV_{\mathscr{E}_{ren}}(\mathtt{f2(M)}).$$

Finally, note that the satisfaction of the FVP allows narrowing-based reachability problems in the specialized rewrite theory $\mathscr{R}_{\mathsf{sp}}$ to be effectively solved in the topmost extension of $\mathscr{R}_{\mathsf{sp}}$, which can be automatically computed by the program transformation of Section 3.2. Therefore, reachability goals such as the one of Example 4 can be solved in the topmost extension of the specialized rewrite theory obtained in Example 10.

Case 2: $\mathscr{E}$ is a finite variant theory. When $\mathscr{E}$ is a finite variant theory, fV-narrowing trees are always finite objects that can be effectively constructed in finite time. Therefore, it is possible to construct the complete fV-narrowing tree for any possible specialized call. The unfolding operator $\mathscr{U}_{FVP}$ develops such a complete fV-narrowing tree for a given input term in $\mathscr{E}$.

The advantage of using $\mathscr{U}_{FVP}$ instead of $\mathscr{U}_{\overline{FVP}}$ is twofold. First, $\mathscr{U}_{FVP}$ disregards any embedding check, which can be extremely time-consuming when $\mathscr{E}$ includes several operators that obey complex modular combinations of algebraic axioms[22]. Second, $\mathscr{U}_{FVP}$ exhaustively explores the whole fV-narrowing tree of a term, while $\mathscr{U}_{\overline{FVP}}$ does not. This leads to a lower degree of specialization when $\mathscr{U}_{\overline{FVP}}$ is applied to a finite variant theory, as shown in the following (pathological) example.

[22] Actually, given an AC operator $\circ$, if we want to check whether a term $t = t_1 \circ t_2 \ldots \circ t_i$ is embedded into another term with a similar form, all possible permutations of the elements of both terms must be tried.

Example 11
Consider the equational theory that is encoded by the following Maude functional module:

```
fmod PATHOLOGICAL is sort Nat .
  ops 0 1 : -> Nat [ctor] .  op mkEven : Nat Nat -> Nat .
  op_+_ : Nat Nat -> Nat [ctor assoc comm id: 0] .
  vars X Y : Nat .
  eq mkEven(X + X + 1,1 + Y) = mkEven(X + X + 1 + 1,Y) [variant] .
  eq mkEven(X + X,0) = X + X [variant] .
endfm
```

The equational theory specifies the encoding for natural numbers in Presburger's style[23] and uses this encoding to define the function `mkEven(X,Y)` that makes `X` even (if `X` is odd) by "moving" one unit from `Y` to `X`. Otherwise, if `X` is even, `X` is left unchanged. The partial evaluation of the given theory w.r.t. the call `mkEven(X,Y)` yields different outcomes that depend on the chosen unfolding operator. On the one hand, by using $\mathcal{U}_{\overline{FVP}}$, the output is the very same input theory, thus no "real" specialization is achieved. On the other hand, the unfolding operator $\mathcal{U}_{FVP}$ produces the specialized definition of `mkEven` given by

```
eq mkEven(X + X,0) = X + X [variant] .
eq mkEven(1 + X + X, 1) = 1 + 1 + X + X [variant] .
```

where the second equation is generated by fully exploring the fV-narrowing derivation

$$\mathsf{mkEven(Z,W)} \leadsto_{\{\mathsf{Z}\mapsto\mathsf{X+X+1},\mathsf{W}\mapsto\mathsf{1+Y}\}} \mathsf{mkEven(1 + 1 + X + X,Y)} \leadsto_{\{\mathsf{Y}\mapsto\mathsf{0}\}} \mathsf{1 + 1 + X + X}$$

where $t \leadsto_\sigma t$ denotes a fV-narrowing step from t to t' with substitution σ. In contrast, $\mathcal{U}_{\overline{FVP}}$ stops the sequence at `mkEven(1 + 1 + X + X, Y)` since `mkEven(1 + 1 + X + X, Y)` embeds `mkEven(Z,W)` and hence yields a specialized equation that is equal (modulo associativity and commutativity of +) to the original equation `mkEven(X + X + 1,1 + Y) = mkEven(X + X + 1 + 1,Y)`.

6 The Presto System

Presto is a program specializer for Maude programs whose core engine consists of approximately 1800 lines of Maude. The system is available, together with a quick start guide and a number of examples at `http://safe-tools.dsic.upv.es/presto`. The Presto distribution package also contains Presto source code for local installation, which is updated to the latest Maude release.

Similarly to Victoria, the earlier partial evaluation tool for equational theories (Alpuente et al., 2020b), Presto follows the on-line approach to partial evaluation that makes control decisions about specialization on the fly. This offers better opportunities for powerful automated strategies than off-line partial evaluation (Jones et al., 1993), where decisions are made before specialization by using abstract data descriptions that are represented as program annotations. Compared to Victoria, the whole Presto implementation is totally new code that is cleaner and much simpler, including the equational least general generalization component of the system that is based on the improved least general generalization algorithm of (Alpuente et al., 2021c). Also, Presto can deal with Maude system modules (i.e., rewrite theories) and multi-module Maude specifications that cannot be managed by Victoria, which only handles equational theories encoded by single Maude functional modules.

[23] Using this encoding, a natural number can be the constant 0 or a sequence of the form `1 + 1 ... + 1`.

Presto can be accessed via a user-friendly web interface that has been developed by using CSS, HTML5, and JavaScript technologies in addition to a command-line interface. This enables users to specialize Maude rewrite theories using a web browser, without the need for a local installation. The Maude core component is connected to the client web user interface through 10 RESTful Web Services that are implemented by using the JAX-RS API for developing Java RESTful Web Services (around 500 lines of Java source code). Since Presto is a fully automatic online specializer, its usage is simple and accessible for inexperienced users. With regard to the security of the system, in order to prevent the execution of unsafe code submitted by users, every call to a Maude function is first checked against a blocklist of unsafe functions so that the process is aborted if there is a call that belongs to the blocklist and a warning message is triggered. We consider unsafe functions to be all of those Maude system-level commands that may remotely control the system-level activities in the server hosting Presto. Also, spawned processes are automatically killed when a time limit is exceeded.

Presto comes with a set of preloaded Maude specifications that includes all of the examples shown in this paper. Preloaded examples can be selected from a pull-down menu, or new specifications can be written from scratch using a dedicated text area. Another pull-down menu allows the user to select the unfolding operator (either $\mathcal{U}_{\overline{FVP}}$ or $\mathcal{U}_{FVP}$) to automatically optimize the input rewrite theory by specializing its underlying guest equational theory and hardcoding the obtained specialized functions into the rewrite rules. Presto also supports the partial evaluation of equational theories w.r.t. a given set of user-defined calls by using either $\mathcal{U}_{\overline{FVP}}$ or $\mathcal{U}_{FVP}$.

Figure 9 shows a screenshot of the Presto interface where the object-oriented client-server communication protocol of Example 9 has been selected among the preloaded examples and the $\mathcal{U}_{\overline{FVP}}$ operator has been chosen for the specialization of the rewrite theory under consideration (the Partial evaluation of rewrite theories (no-FVP/embedding) option in the pull-down menu).

Finally, Presto includes the following useful additional features.

Specialization views. Two possible views of the resulting specialization are delivered, with or without the compression phase enabled. The user can switch back and forth between the two representations by respectively selecting the Specialized and Specialized with Compression tabs. Figures 10 and 11 respectively show the specialization of the client-server communication protocol before and after the compression phase. Furthermore, by accessing the Renaming tab, it is possible to explicitly inspect the renaming computed by the compression phase and see how nested calls are dramatically compacted by using the corresponding fresh function symbols.

Inspection mode. By activating the inspect tab, Presto shows detailed information of the specialization process that can be particularly useful for a user who wants to experiment with different specializations for analysis and verification purposes. Specifically, for each iteration i of the EqNPE$^{\mathcal{U}}$ algorithm (invoked in Line 4 of Algorithm 1), Presto shows all of the leaves (term variants) in the deployed fV-narrowing trees for the calls that were specialized at the iteration i, together with their corresponding computed substitution (i.e., the substitution component of a term variant). Also, thanks to the interconnection with the narrowing stepper NARVAL (Alpuente et al., 2019b), each fV-narrowing tree computed at iteration i can be graphically visualized and interactively navigated. This is automatically done by simply selecting the desired tree root term to be expanded from the displayed list of calls that were specialized at any iteration (including the last one). This can be done by simply clicking the hyperlink associated with the desired

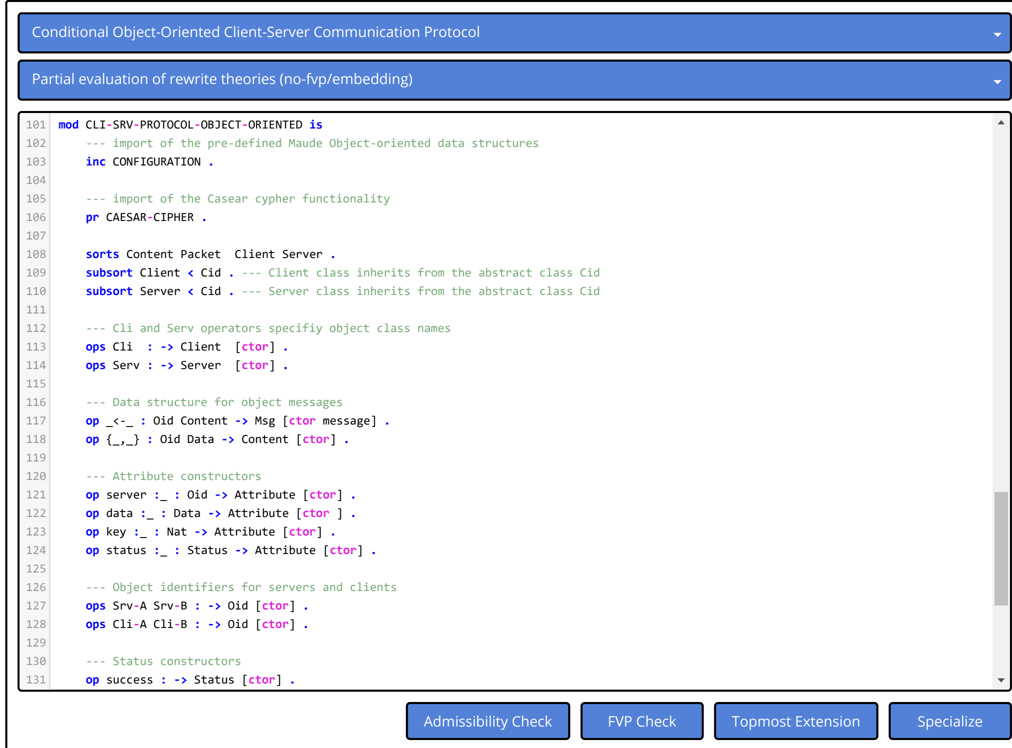

Fig. 9. Loading specifications and selecting the unfolding strategy in Presto.

call. By doing this, the user can perform a fine-grained and stepwise inspection of all narrowing steps, computation of equational unifiers, and interactive exploration of different representations of Maude's narrowing and rewriting search spaces (tree-based or graph-based, source-level or meta-level). For instance, Figure 12 shows that, at iteration 1, a fV-narrowing tree was expanded whose root term is the specialized call `dec(M:Data,s(0))` (Line 47). The root can be narrowed to the leaf a (Line 48) with the computed substitution `{M ↦ b}` (Line 49), which allows the resultant equation `dec(b,s(0)) = a` to be generated (Line 50). Furthermore, Figure 13 shows (a fragment of) the fV-narrowing tree for the specialized call `dec(M:Data,s(0))` of Line 47 that has been generated by the Narval system[24].

Topmost extension transformation. The topmost extension transformation of Section 3.2 has been fully implemented in Presto. Since the topmost property is not required to perform the partial evaluation, it can be applied either *ex-ante* to the original input program (and it is preserved by the specialization) or *ex-post* to the specialized program. The transformation enables narrowing-based search reachability for all topmost modulo Ax programs, and particularly for object-oriented specifications. Reachability goals can be solved in Presto through the connection to the NARVAL system (Alpuente et al., 2019b), which provides a graphical environment for

[24] Variables in Narval are standardized apart by using fresh variable names of the form #n. In our example, #1 is a renaming for variable M.

Fig. 10. Specialized client-server communication protocol before the compression phase.

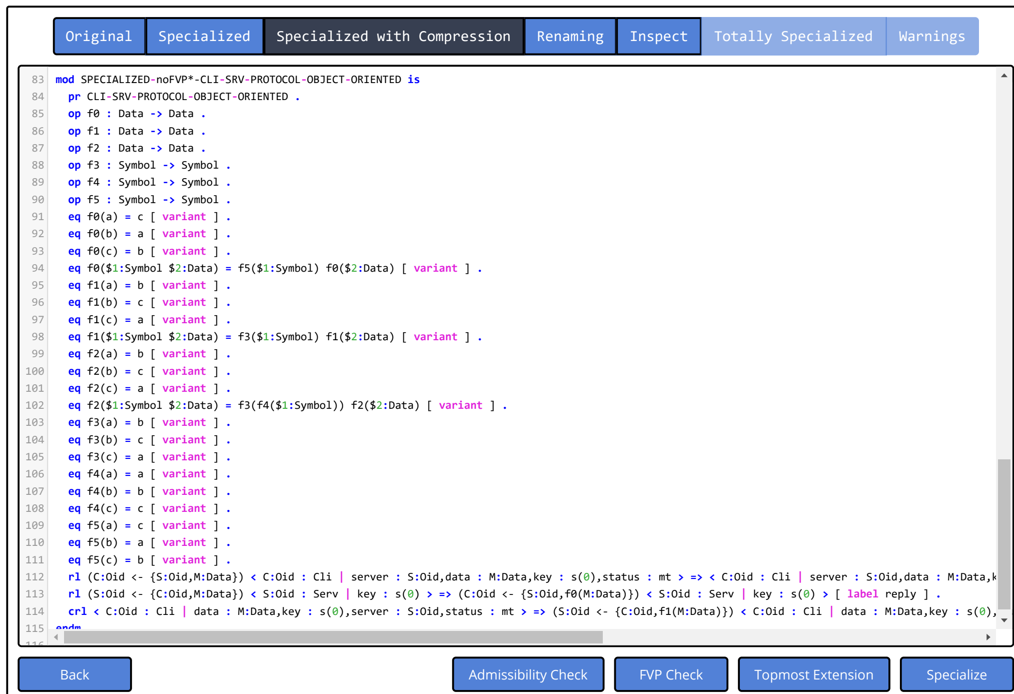

Fig. 11. Specialized client-server communication protocol after the compression phase.

Fig. 12. Inspecting the specialization process.

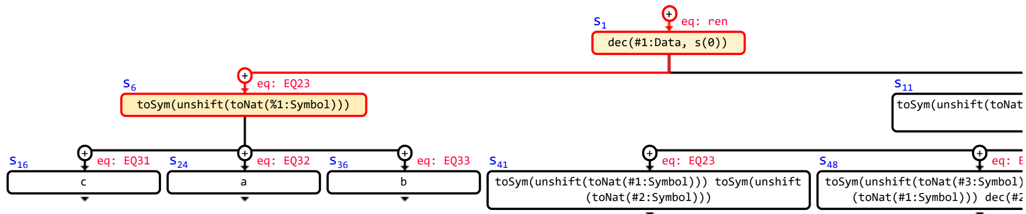

Fig. 13. Fragment of the fV-narrowing tree for `dec(M:Data,s(0))`.

symbolic analysis in Maude. This feature is illustrated in the system quick start guide.

Finite Variant Property and Strong irreducibility checkers. Presto includes a FVP checker for equational theories that is based on the checking procedure described in (Meseguer, 2015). All of the theories that satisfy the FVP can be specialized by using the unfolding operator $\mathcal{U}_{FVP}$ instead of the typically costlier operator $\mathcal{U}_{\overline{FVP}}$. As discussed in Section 3, the checker implements a semi-decidable procedure that terminates when the FVP holds for the equational theory under examination. If the FVP does not hold, the user has to manually stop the checking process or wait for the time-out; otherwise, the process would produce an infinite number of most general variants for some terms. For instance, consider the *Client-Server Communication Protocol with FVP* that is available in the Presto preloaded examples. This Maude specification is a slight mutation of the client-server communication protocol of Example 9 that models the *Caesar* cypher by using a finite variant equational theory. By clicking the Check FVP button, the checking pro-

Operator declaration	FVP	Computed variants
op _-_ : Nat Nat -> Nat .	yes	3
op _>=_ : Nat Nat -> Bool .	yes	3
op _>_ : Nat Nat -> Bool .	yes	3
op `[_`,_`,_`] : Bool Message? Message? -> Message? .	yes	3
op `[_`,_`,_`] : Bool Nat Nat -> Nat .	yes	3
op d : Nat Nat -> Nat .	yes	11
op dec : Message Nat -> Message? .	yes	26
op e : Nat Nat -> Nat .	yes	11
op enc : Message Nat -> Message? .	yes	19
op len : -> Nat .	yes	1
op toNat : Message -> Nat .	yes	4
op toSym : Nat -> Message? .	yes	5
This theory has the Finite Variant Property		

Stop Start

Fig. 14. FVP checking of the Client-Server Communication Protocol with FVP.

cess starts and, in this case, the check succeeds since the flat terms associated with the program operators have only a finite number of most general variants, as illustrated in Figure 14. Hence, the specification could be safely specialized by using the $\mathscr{U}_{FVP}$ unfolding operator without the risk of jeopardizing termination.

Moreover, Presto implements an efficient checker for the $(\vec{E},B)$-strong irreducibility of the left-hand side of the rewrite rules of the input program. As discussed at the end of Section 4.2, strong irreducibility is an essential requirement for the completeness of the specialization algorithm. Figure 15 illustrates the outcome of the strong irreducibility check for a variant of the Client-Server Communication Protocol with FVP that includes the rewrite rule

```
rl [C:CliName,S:ServName,enc(Q:Message,K:Nat),K:Nat,mt]
                      => (S:ServName <- {C:CliName,enc(Q:Message, K:Nat)}) &
                         [C:CliName,S:ServName,Q:Message,K:Nat,mt] .
```

Note that the check fails due to the narrowable subterm `enc(Q:Message, K:Nat)` occurring in the left-hand side of the considered rule.

Iterative specialization. Presto can be automatically re-applied to any generated specialization in an iterative manner. After completing a given specialization for a rewrite theory, the user can perform a further specialization process to re-specialize the given outcome by using a distinct unfolding operator. This is achieved by simply pressing the Specialize button again, which is directly displayed after the specialization and avoids the need to reload the initial web input form to manually enter the intermediate program to be re-specialized. As shown in (Alpuente et al., 2021a), this feature is particularly useful in the case when a given input rewrite theory $\mathscr{R}$ must be specialized using $\mathscr{U}_{\overline{FVP}}$, since its equational theory does not satisfy the FVP, while the resulting specialization $\mathscr{R}'$ hosts an equational theory that does satisfy the FVP so that $\mathscr{R}'$ can be further specialized by using the $\mathscr{U}_{FVP}$ operator. For a detailed session with Presto that illustrates this feature, we refer to the Presto quick start guide.

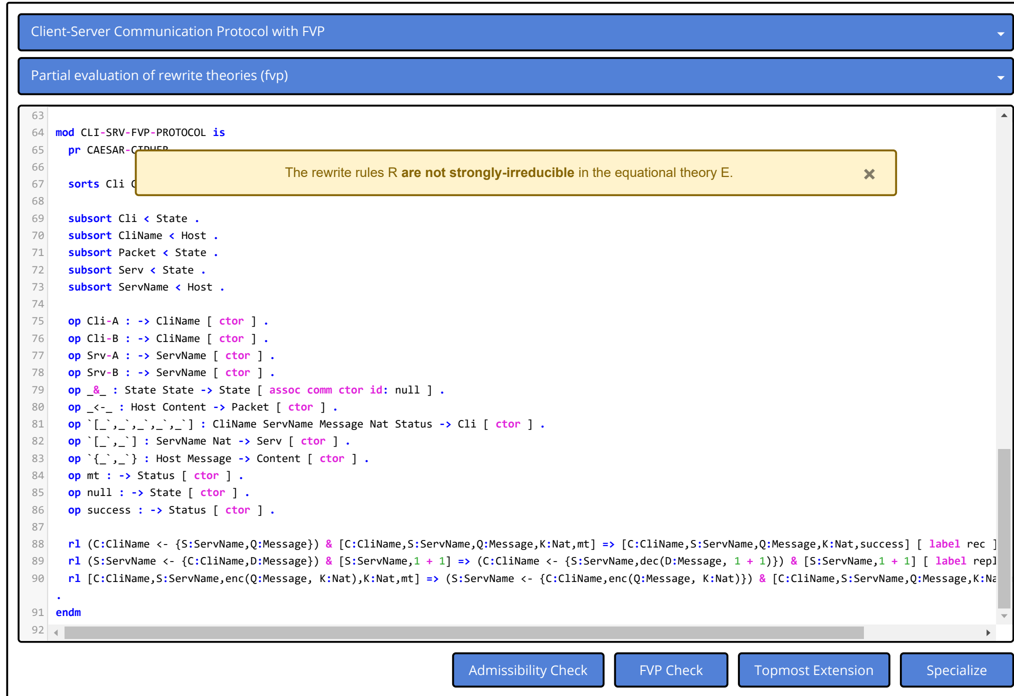

Fig. 15. Strong irreducibility failure for a variant of the Client-Server Communication Protocol with FVP.

Total evaluation. This feature provides a slightly modified version of the unfolding operator $\mathcal{U}_{FVP}$ that implements the total evaluation transformation of (Meseguer, 2020) for sufficiently complete equational theories that have the FVP. By means of a suitable *sort downgrading*, which is described in (Meseguer, 2020), function calls always generate values and cannot be stuck in an intermediate, partial result. In this scenario, unfolding can safely rule out variants that are not constructor terms. This means that all of the specialized calls get totally evaluated and the maximum compression is achieved, thereby dramatically reducing the search space for the construction of the specialized theories as shown in (Alpuente et al., 2021a).

For instance, the specialized rewrite theory *Client-Server Communication Protocol with FVP* can be totally evaluated into the extremely compact rewrite theory of Figure 16, which does not include any equation. Actually, all calls to the encryption and decryption functions have been totally evaluated, and their values have been hard-coded into their place in the specialized rewrite rules.

7 Experimental Evaluation of Presto

Table 1 contains the experiments that we have performed with Presto using an Intel Xeon E5-1660 3.3GHz CPU with 64 GB RAM running Maude v3.0 and considering the average of ten executions for each test. These experiments, together with the source code of all of the examples are also publicly available at Presto's website.

We have considered the following benchmark programs: *Cond-Cli-Srv-OO*, the conditional, object-oriented, client-server communication protocol of Example 9; *Cli-Srv*, a re-encoding of *Cond-Cli-Srv-OO* that does not use either Maude built-in object-oriented features or conditional

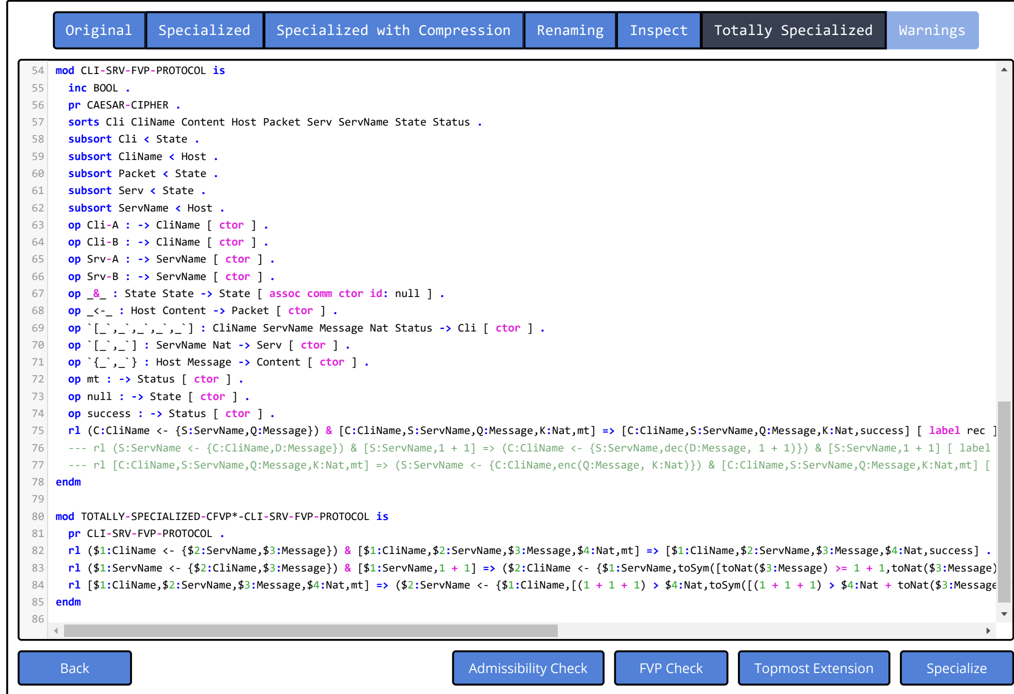

Fig. 16. Total evaluation of the Client-Server Communication Protocol with FVP.

rules; *Cli-Srv-Mod*, a variant of *Cli-Srv* where we introduce an extra function (i.e., the `mod` function that computes the remainder of the integer division) in the underlying equational theory that is commonly used in protocol specification and makes the key generation heavier; *Cli-Srv-FVP*, a variant of *Cli-Srv* for one-symbol messages whose equational theory meets the FVP and differs from the one in Example 10 in using Presburger's encoding for natural numbers; *Diffie-Hellman*, the well-known network protocol for safely sharing keys between two nodes over an untrusted channel; *Cond-Diffie-Hellman-OO*, a conditional, object-oriented specification of the Diffie-Hellman protocol; *Handshake-KMP*, an unconditional rewrite theory that specifies a simple handshake protocol in which a client sends an arbitrary long and noisy message `M` to a server. The handshake succeeds if the server can recognize a secret handshake sequence `P` inside the client message `M` by matching `P` against `M` via the well-known KMP string matching algorithm; and *Cond-Handshake-KMP-OO*, a conditional, object-oriented version of *Handshake-KMP*.

As for the experiments with *Cli-Srv*, *Cli-Srv-Mod* and *Cond-Cli-Srv-OO*, we considered input messages of three different sizes: from twenty-five thousand symbols to one million symbols. This cannot be done for *Cli-Srv-FVP* since it encodes one-symbol messages, and, hence, we cannot use this parameter for benchmarking purposes. Similarly, *Handshake-KMP and* Cond-Handshake-KMP-OO *consider* fixed-size messages of 350 symbols, while *Diffie-Hellman* and *Cond-Diffie-Hellman-OO* are specialized w.r.t. fixed-size keys (modelled as natural numbers obtained though exponentiation). For these experiments, we use as benchmark parameter the number of client requests sent over a time-bounded period of time. We considered one, five, and ten millions requests. The column *Size* in Table 1 indicates either the message length parameter or the number of client requests according to the benchmark program considered.

For each experiment, we executed the original specification $\mathcal{R}$ and the specialized one $\mathcal{R}'$ on

Program	*Size*	$\#Rews_{\mathcal{R}}$	$\#Rews_{\mathcal{R}'}$	*Reduction*	$T_{\mathcal{R}}$ (ms)	$T_{\mathcal{R}'}$ (ms)	*Speedup*
Cli-Srv	25K	650,012	100,002	84.62%	47	15	3.13
	100K	2,600,115	400,002	84.62%	261	87	3.00
	1M	26,000,100	4,000,002	84.62%	5,630	2,434	2.31
Cli-Srv-Mod	25K	8,900,012	100,002	98.88%	780	14	55.71
	100K	35,600,115	400,002	98.88%	3,723	87	42.79
	1M	356,000,100	4,000,002	98.88%	56,087	2,399	23.38
Diffie-Hellman	1M	302,000,000	2,000,000	99.34%	21,163	584	36.24
	5M	1,510,000,000	10,000,000	99.34%	104,959	3,087	34.00
	10M	3,020,000,000	20,000,000	99.34%	207,189	6,361	32.57
Cli-Srv-FVP	1M	12,000,000	4,000,000	66.67%	976	339	2.88
	5M	60,000,000	20,000,000	66.67%	4,866	1,680	2.90
	10M	120,000,000	40,000,000	66.67%	10,305	3,310	3.11
Handshake-KMP	1M	1,575,500,000	1,500,000	99.90%	67,646	546	123.89
	5M	7,877,500,000	7,500,000	99.90%	336,147	3,042	110.50
	10M	15,755,000,000	15,000,000	99.90%	655,962	6,077	107.94
Cond-Cli-Srv-OO	25K	1,500,000	200,000	86.67%	118	36	3.28
	100K	6,000,000	800,000	86.67%	405	125	3.24
	1M	60,000,000	8,000,000	86.67%	4,017	1,250	3.21
Cond-Diffie-Hellman-OO	1M	695,000,000	1,000,000	99.86%	48,048	822	58.45
	5M	3,475,000,000	5,000,000	99.86%	249,551	4,060	61.47
	10M	6,950,000,000	10,000,000	99.86%	526,527	8,107	64.95
Cond-Handshake-KMP-OO	1M	3,833,996,167	1,000,000	99.97%	189,779	873	217.39
	5M	19,169,996,167	5,000,000	99.97%	923,511	4,330	213.28
	10M	38,339,996,167	10,000,000	99.97%	1,886,552	8,898	212.02

Table 1. *Experimental results for the specialization of rewrite theories with* Presto.

the very same input states, and we recorded the following data: the execution times (in ms) $T_{\mathcal{R}}$ and $T'_{\mathcal{R}}$, the total number of rewrites $\#Rews_{\mathcal{R}}$ and $\#Rews_{\mathcal{R}'}$, the percentage of *reduction* in terms of number of rewrites, and the specialization *speedup* computed as the ratio $T_{\mathcal{R}}/T_{\mathcal{R}'}$.

Our figures show that the specialized rewrite theories achieve a significant improvement in execution time when compared to the original theory, with an average speedup for these benchmarks of 59.24. Particularly remarkable is the performance improvement of the *Cond-Handshake-KMP-OO* which reaches two orders of magnitude for the case when the specialized maximal calls within the rewrite theory partially instantiate the input pattern and input message. Such an impressive performance is largely due to the compression phase of the specialization algorithm that, in this case, completely deconditionalizes the rewrite rules, thereby yielding as outcome a faster, unconditional rewrite theory. This does not happen for *Cond-Cli-Srv-OO* which exhibits a smaller speedup. For this benchmark, complete deconditionalization cannot be achieved as shown in Example 9, hence some heavy conditional computations are kept in the specialized program as well.

We note that none of these specializations could be performed by using our earlier partial evaluator Victoria, which cannot handle rewrite theories but only equational theories. Thus, the specialization of rewrite theories supported by Presto could not be achieved in Victoria unless a complex hack is introduced not only at the level of the theory signature but also by providing a suitable program infrastructure that simulates rewrite rule nondeterminism through deterministic, equational evaluation.

We do not benchmark the specialization times since they are almost negligible (for most cases $<$100 ms). Nonetheless, we compared the performance of the unfolding operators $\mathcal{U}_{FVP}$ and

$\mathscr{U}_{\overline{FVP}}$, when both can be applied in the specialization process (i.e., when the rewrite theory to be specialized includes a finite variant equational theory). The most significant difference appears in the *Cli-Srv-FVP* theory, where NPER$^{\mathscr{U}_{FVP}}$ is 56% faster (on average) than NPER$^{\mathscr{U}_{\overline{FVP}}}$. This figure is expected and favors the use of $\mathscr{U}_{FVP}$ whenever it is possible since it avoids any overhead caused by homeomorphic embedding checks performed by $\mathscr{U}_{\overline{FVP}}$. Finally, we benchmarked the symbolic execution of long-winded *reachability goals* in rewrite theories that include a finite variant equational theory such as the pathological theory of Example 11. The achieved improvement is also striking, with an average speedup of 16.46.

Let us finally discuss how Presto compares in practice with existing on-line specialization systems with related transformation power. A narrowing-based partial evaluator for the lazy functional logic language Curry is described in (Hanus and Peemöller, 2014; Peemöller, 2017) whose implementation can be seen as an instance of the generic narrowing-based partial evaluation framework of (Alpuente et al., 1998b). This system improves a former prototype in (Albert et al., 2002) by taking into account (mutually recursive) let expressions and non-deterministic operations, while the PE system of (Albert et al., 2002) was restricted to confluent programs. Obviously, the protocol benchmarks in this paper cannot be directly specialized by using Curry's partial evaluator since neither evaluation modulo algebraic axioms nor concurrency are supported by Curry's partial evaluator; this would require artificially rewriting the program code so that any comparison would be meaningless. In the opposite direction, Presto cannot manage the specialization of higher-order functions that is achieved by (Hanus and Peemöller, 2014) since RWL is not a higher-order logic. Furthermore, our partial evaluator cannot deal with many other important Curry features such as monadic I/O or encapsulated search. In order to provide some evidence of how Presto performs on classical specialization examples, we reproduced the standard *kmp-match* test[25] (i.e., without using any algebraic axioms), and we benchmarked the speed and specialization that is achieved by Presto on this program. This example is particularly interesting because it is a kind of transformation that neither (conventional) PE nor deforestation can perform automatically while conjunctive partial deduction[26] of logic programs and positive supercompilation of functional programs can pass the test (Alpuente et al., 1998a; Jørgensen et al., 1996). The speedup achieved by Presto on this test for an input list with 500,000 elements is 58, which is comparable to the speedup of the last release of Victoria (actually, the KMP specialized programs that are generated by Presto and Victoria are essentially the same and they both yield an optimal specialized string matcher). Curry's partial evaluator also produces an optimal KMP specialization and the achieved speedup for input lists of $500,000$ elements is 12.22, which is obtained by using an unlimited unfolding strategy that, however, may risk nontermination (Peemöller, 2017). The corresponding specialization time of Curry's partial evaluator reported in (Peemöller, 2017) is 1.61 seconds.

We compared the specialization times achieved by Presto in comparison to Victoria as follows. In order to reduce the noise on the small time for a single specialization, we benchmarked the total specialization time for a loop consisting of 1000 specializations of the KMP program, and then we divided the accumulated specialization time by 1000. The resulting specialization times of Victoria and Presto are 6.6 ms and 5.6 ms, respectively, which shows that Presto

[25] This test is often used to compare the strength of specializers: the specialization of a semi-naïve pattern matcher for a fixed pattern into an efficient algorithm (Sørensen et al., 1994).

[26] The transformation power of conjunctive partial deduction is comparable to narrowing-driven partial evaluation as they both achieve unfold/fold-like program transformations such as tupling and deforestation within a fully automated partial evaluation framework (Alpuente et al., 1998a).

outperforms Victoria with a performance improvement of 15%. This is noteworthy since the implementation of Presto is considerably more ambitious, and it provides much greater coverage compared to the simpler approach of Victoria.

8 Conclusion

The use of logic programming and narrowing for system analysis has been advocated since the narrowing-based NRL protocol analyzer (Meadows, 1996), which was developed in Prolog. In this article, we have shown that concurrent software analysis is a promising new application area for narrowing-based program specialization. We believe that Maude's partial evaluator Presto is not only relevant for the (multi-paradigm) logic programming community working on source code optimization, but it can also be a valuable tool for anyone who is interested in the symbolic analysis and verification of concurrent systems. The main reason why Presto is so effective in this area is that it not only achieves huge speedup for important classes of rewrite theories, but it can also cut down an infinite (folding variant) narrowing space to a finite one for the underlying equational theory $\mathcal{E}$. By doing this, any $\mathcal{E}$-unification problem can be finitely solved and symbolic, narrowing-based analysis with rules R modulo $\mathcal{E}$ can be effectively performed. Moreover, in many cases, the specialization process transforms a rewrite theory whose operators obey algebraic axioms, such as associativity, commutativity, and unity, into a much simpler rewrite theory with the same semantics and no structural axioms so that it can be run in an independent rewriting infrastructure that does not support rewriting or narrowing modulo axioms. Finally, some costly analyses that may require significant (or even unaffordable) resources in both time and space, can now be safely and effectively performed.

The current specialization system goes beyond the scope of the original NPER framework of (Alpuente et al., 2021b), which only works on unconditional rewrite theories. Presto is now "feature complete" and stable. As such, it is not only suitable for experimental use but also for production use. Since Presto is a fully automatic online specializer, its usage is simple and accessible for Maude users, and its interactive features make the specialization process fully checkable. For instance, the user can simply click on a specialized call to see its unfolding tree and all of the derived resultants.

Besides the applications outlined in this article, further applications could benefit from the optimization of variant generation that is achieved by Presto. For instance, an important number of applications (and tools) are currently based on narrowing-based variant generation: for example, the protocol analyzers Maude-NPA (Escobar et al., 2009) and Tamarin (Meier et al., 2013), Maude debuggers and program analyzers (Alpuente et al., 2016; Alpuente et al., 2012; Alpuente et al., 2014a), termination provers, variant-based satisfiability checkers, coherence and confluence provers, and different applications of symbolic reachability analysis (Durán et al., 2020a). Also, as stated in (Leuschel and Lehmann, 2000), there are big advantages for using the partial deduction approach to model checking due to the built-in support of logic programming for non-determinism and unification. This approach might also be of great value in the optimization of Maude's logical model checkers for infinite-state systems, where narrowing, equational unification, and equational abstractions play a primary role (Escobar and Meseguer, 2007; Bae and Meseguer, 2015). We plan to investigate this in future research.

We also plan to implement new unfolding operators in Presto that can achieve the total evaluation of (Meseguer, 2020) for the case of equational theories that are sufficiently complete modulo axioms but do not have the FVP. This requires significant research effort since total evaluation

relies on the notion of constructor variant and there does not exist in the literature a finite procedure to compute a finite, minimal, and complete set of most general constructor variants for theories that do not have the FVP, even if this set exists. Another interesting strand for further research is dealing with the recently proposed Maude strategy language (Durán et al., 2020a) in program specialization.

References

Aguirre, L., Martí-Oliet, N., Palomino, M., and Pita, I. (2014). Conditional Narrowing Modulo in Rewriting Logic and Maude. In Escobar, S., editor, *Rewriting Logic and Its Applications*, pages 80–96. Springer International Publishing.

Albert, E., Alpuente, M., Falaschi, M., and Vidal, G. (1998). INDY User's Manual. Technical Report DSIC-II/12/98, UPV. Available at: `http://elp.webs.upv.es/soft/indy/indy.html`.

Albert, E., Alpuente, M., Hanus, M., and Vidal, G. (1999). A Partial Evaluation Framework for Curry Programs. In Ganzinger, H., McAllester, D. A., and Voronkov, A., editors, *Logic Programming and Automated Reasoning, 6th International Conference, LPAR'99, Tbilisi, Georgia, September 6-10, 1999, Proceedings*, volume 1705 of *Lecture Notes in Computer Science*, pages 376–395. Springer.

Albert, E., Gallagher, J. P., Gómez-Zamalloa, M., and Puebla, G. (2009). Type-based homeomorphic embedding for online termination. *Inf. Process. Lett.*, 109(15):879–886.

Albert, E., Hanus, M., and Vidal, G. (2002). A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002:1–34.

Alpuente, M., Ballis, D., Cuenca-Ortega, A., Escobar, S., and Meseguer, J. (2019a). ACUOS^2: A High-Performance System for Modular ACU Generalization with Subtyping and Inheritance. In *Proceedings of the 16th European Conference on Logics in Artificial Intelligence (JELIA 2019)*, volume 11468 of *Lecture Notes in Computer Science*, pages 171–181. Springer.

Alpuente, M., Ballis, D., Escobar, S., Meseguer, J., and Sapiña, J. (2021a). Optimizing Maude Programs via Program Specialization. In Gallagher, J. P., Giacobazzi, R., and López-García, P., editors, *Festschrift volume in honor of Manuel Hermenegildo (submitted, invited contribution)*. Elsevier Science. Available at: `http://elp.webs.upv.es/papers/ABEMS20-Festschrift.pdf`.

Alpuente, M., Ballis, D., Escobar, S., and Sapiña, J. (2019b). Symbolic Analysis of Maude Theories with Narval. *Theory and Practice of Logic Programming*, 19(5–6):874–890.

Alpuente, M., Ballis, D., Escobar, S., and Sapiña, J. (2021b). Optimization of Rewrite Theories by Equational Partial Evaluation. *Journal of Logical and Algebraic Methods in Programming*. To appear. Available at: `http://elp.webs.upv.es/papers/ABES21-jlamp.pdf`.

Alpuente, M., Ballis, D., Frechina, F., and Romero, D. (2012). Backward Trace Slicing for Conditional Rewrite Theories. In *Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2012)*, volume 7180 of *Lecture Notes in Computer Science*, pages 62–76. Springer.

Alpuente, M., Ballis, D., Frechina, F., and Sapiña, J. (2016). Debugging Maude Programs via Runtime Assertion Checking and Trace Slicing. *Journal of Logical and Algebraic Methods in Programming*, 85:707–736.

Alpuente, M., Ballis, D., and Romero, D. (2014a). A Rewriting Logic Approach to the Formal Specification and Verification of Web Applications. *Science of Computer Programming*, 81:79–107.

Alpuente, M., Ballis, D., and Sapiña, J. (2019c). Static Correction of Maude Programs with Assertions. *Journal of Systems and Software*, 153:64–85.

Alpuente, M., Ballis, D., and Sapiña, J. (2020a). Efficient Safety Enforcement for Maude Programs via Program Specialization in the ÁTAME system. *Mathematics in Computer Science*, 14(3):591–606.

Alpuente, M., Comini, M., Escobar, S., Iborra, J., and Falaschi, M. (2010). A Compact Fixpoint Semantics for Term Rewriting Systems. *Theor. Comput. Sci.*, 411(37):3348Ð3371.

Alpuente, M., Cuenca-Ortega, A., Escobar, S., and Meseguer, J. (2020b). A Partial Evaluation Framework for Order-Sorted Equational Programs modulo Axioms. *Journal of Logical and Algebraic Methods in Programming*, 110:1–36.

Alpuente, M., Cuenca-Ortega, A., Escobar, S., and Meseguer, J. (2020c). Order-sorted Homeomorphic Embedding modulo Combinations of Associativity and/or Commutativity Axioms. *Fundamenta Informaticae*, 177(3-4):297–329.

Alpuente, M., Escobar, S., Espert, J., and Meseguer, J. (2014b). A Modular Order-Sorted Equational Generalization Algorithm. *Information and Computation*, 235:98–136.

Alpuente, M., Escobar, S., and Iborra, J. (2009a). Termination of Narrowing Revisited. *Theoretical Computer Science*, 410(46):4608–4625.

Alpuente, M., Escobar, S., Meseguer, J., and Ojeda, P. (2008). A Modular Equational Generalization Algorithm. In *Proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2008)*, volume 5438 of *Lecture Notes in Computer Science*, pages 24–39. Springer.

Alpuente, M., Escobar, S., Meseguer, J., and Ojeda, P. (2009b). Order-Sorted Generalization. *Electronic Notes in Theoretical Computer Science*, 246:27–38.

Alpuente, M., Escobar, S., Meseguer, J., and Sapiña, J. (2021c). Order-sorted Equational Generalization Algorithm Revisited. *Annals of Mathematics and Artificial Intelligence*. Available at: https://doi.org/10.1007/s10472-021-09771-1.

Alpuente, M., Falaschi, M., Julián, P., and Vidal, G. (1997). Specialization of Lazy Functional Logic Programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 1997)*, pages 151–162. Association for Computing Machinery.

Alpuente, M., Falaschi, M., and Vidal, G. (1998a). A Unifying View of Functional and Logic Program Specialization. *ACM Computing Surveys*, 30(3es):9es.

Alpuente, M., Falaschi, M., and Vidal, G. (1998b). Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844.

Alpuente, M., Lucas, S., Hanus, M., and Vidal, G. (2005). Specialization of Functional Logic Programs based on Needed Narrowing. *Theory and Practice of Logic Programming*, 5(3):273–303.

Bae, K., Escobar, S., and Meseguer, J. (2013). Abstract Logical Model Checking of Infinite-State Systems Using Narrowing. In *Proceedings of the 24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, volume 21 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 81–96. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Bae, K. and Meseguer, J. (2015). Model Checking Linear Temporal Logic of Rewriting Formulas under Localized Fairness. *Science of Computer Programming*, 99:193–234.

Bouchard, C., Gero, K. A., Lynch, C., and Narendran, P. (2013). On Forward Closure and the Finite Variant Property. In *Proceedings of the 9th International Symposium on Frontiers of Combining Systems (FroCos 2013)*, volume 8152 of *Lecture Notes in Computer Science*, pages 327–342. Springer.

Burstall, R. M. and Darlington, J. (1977). A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67.

Cadar, C. and Sen, K. (2013). Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM*, 56(2):82–90.

Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R., and Talcott, C. (2020). Maude Manual (Version 3.0). Technical report, SRI International Computer Science Laboratory. Available at: `http://maude.cs.uiuc.edu`.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2007a). *All About Maude: A High-Performance Logical Framework*. Springer.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2007b). *All About Maude: A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer-Verlag.

Cook, W. R. and Lämmel, R. (2011). Tutorial on Online Partial Evaluation. In *Proceedings of the IFIP Working Conference on Domain-Specific Languages (DSL 2011)*, volume 66 of *Electronic Proceedings in Theoretical Computer Science*, pages 168–180. Open Publishing Association.

Cousot, P. and Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252. Association for Computing Machinery.

De Schreye, D., R.Glück, Jørgensen, J., Leuschel, M., Martens, B., and Sørensen, M. H. (1999). Conjunctive Partial Deduction: Foundations, Control, Algorithms, and Experiments. *The Journal of Logic Programming*, 41(2-3):231–277.

de Waal, D. A. and Gallagher, J. P. (1994). The Applicability of Logic Program Analysis and Transformation to Theorem Proving. In Bundy, A., editor, *12th International Conference on Automated Deduction (CADE'94)*, volume 814 of *Lecture Notes in Computer Science*, pages 207–221. Springer.

Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., and Talcott, C. (2020a). Programming and Symbolic Computation in Maude. *Journal of Logical and Algebraic Methods in Programming*, 110.

Durán, F., Lucas, S., and Meseguer, J. (2008). MTT: The Maude Termination Tool (System Description). In *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR 2008)*, volume 5195 of *Lecture Notes in Computer Science*, pages 313–319. Springer.

Durán, F., Meseguer, J., and Rocha, C. (2020b). Ground Confluence of Order-Sorted Conditional Specifications Modulo Axioms. *Journal of Logical and Algebraic Methods in Programming*, 111:100513.

Escobar, S. (2014). Functional Logic Programming in Maude. In *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi (SAS 2014)*, volume 8373 of *Lecture Notes in Computer Science*, pages 315–336. Springer.

Escobar, S., Meadows, C., and Meseguer, J. (2009). Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties. In *Foundations of Security Analysis and Design V*

(FOSAD 2007/2008/2009 Tutorial Lectures), volume 5705 of *Lecture Notes in Computer Science*, pages 1–50. Springer.

Escobar, S. and Meseguer, J. (2007). Symbolic Model Checking of Infinite-State Systems Using Narrowing. In *Proceedings of the 18th International Conference on Term Rewriting and Applications (RTA 2007)*, volume 4533 of *Lecture Notes in Computer Science*, pages 153–168. Springer.

Escobar, S., Sasse, R., and Meseguer, J. (2012). Folding Variant Narrowing and Optimal Variant Termination. *The Journal of Logic and Algebraic Programming*, 81(7–8):898–928.

Fay, M. (1979). First Order Unification in an Equational Theory. In *Proceedings of the 4th International Conference on Automated Deduction (CADE 1979)*, pages 161–167. Academic Press, Inc.

Gallagher, J. P. (1993). Tutorial on Specialisation of Logic Programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 1993)*, pages 88–98. Association for Computing Machinery.

Gallagher, J. P. and Bruynooghe, M. (1990). The Derivation of an Algorithm for Program Specialisation. In Warren, D. H. D. and Szeredi, P., editors, *Seventh International Logic Programming Conference (ICLP 1990)*, pages 732–746. MIT Press.

Garavel, H., Tabikh, M., and Arrada, I. (2018). Benchmarking Implementations of Term Rewriting and Pattern Matching in Algebraic, Functional, and Object-Oriented Languages - The 4th Rewrite Engines Competition. In *Proceedings of the 12th International Workshop on Rewriting Logic and its Applications (WRLA 2018)*, volume 11152 of *Lecture Notes in Computer Science*, pages 1–25. Springer.

Hanus, M. (1994). The Integration of Functions into Logic Programming: From Theory to Practice. *The Journal of Logic Programming*, 19/20:583–628.

Hanus, M. (1997). Integration of Declarative Paradigms: Benefits and Challenges. *ACM SIGPLAN Notices*, 32(1):77–79.

Hanus, M. and Peemöller, B. (2014). A Partial Evaluator for Curry. In *Proceedings of the 28th International Workshop on (Constraint) Logic Programming (WLP 2014)*, volume 1335, pages 155–171. CEUR-WS.org.

Jones, N. D., Gomard, C. K., and Sestoft, P. (1993). *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.

Jørgensen, J., Leuschel, M., and Martens, B. (1996). Conjunctive Partial Deduction in Practice. In *Proceedings of the 6th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 1996)*, volume 1207 of *Lecture Notes in Computer Science*, pages 59–82. Springer.

Leuschel, M. (1998). On the Power of Homeomorphic Embedding for Online Termination. In *Proceedings of the 5th International Symposium on Static Analysis (SAS 1998)*, volume 1503 of *Lecture Notes in Computer Science*, pages 230–245. Springer.

Leuschel, M., Elphick, D., Varea, M., Craig, S., and Fontaine, M. (2006). The Ecce and Logen partial evaluators and their web interfaces. In Hatcliff, J. and Tip, F., editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM 2006)*, pages 88–94. ACM.

Leuschel, M. and Gruner, S. (2001). Abstract Conjunctive Partial Deduction Using Regular Types and Its Application to Model Checking. In Pettorossi, A., editor, *Logic Based Program Synthesis and Transformation, 11th International Workshop (LOPSTR 2001)*, volume 2372 of *Lecture Notes in Computer Science*, pages 91–110. Springer.

Leuschel, M. and Lehmann, H. (2000). Solving Coverability Problems of Petri nets by Partial Deduction. In Gabbrielli, M. and Pfenning, F., editors, *2nd international ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, pages 268–279. ACM.

Lloyd, J. W. and Shepherdson, J. C. (1991). Partial Evaluation in Logic Programming. *The Journal of Logic Programming*, 11(3-4):217–242.

Lucas, S. and Meseguer, J. (2016). Normal forms and normal theories in conditional rewriting. *J. Log. Algebraic Methods Program.*, 85(1):67–97.

Martens, B. and Gallagher, J. (1995). Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In *Proceedings of the 12th International Conference on Logic Programming (ICLP 1995)*, pages 597–611. The MIT Press.

Martí-Oliet, N. and Meseguer, J. (2002). Rewriting Logic: Roadmap and Bibliography. *Theoretical Computer Science*, 285(2):121–154.

Meadows, C. (1996). The NRL Protocol Analyzer: An Overview. *The Journal of Logic Programming*, 26(2):113–131.

Meier, S., Schmidt, B., Cremers, C., and Basin, D. A. (2013). The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013)*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer.

Meseguer, J. (1992a). Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155.

Meseguer, J. (1992b). Multiparadigm Logic Programming. In *Proceedings of the 3rd International Conference on Algebraic and Logic Programming (ALP 1992)*, volume 632 of *Lecture Notes in Computer Science*, pages 158–200. Springer.

Meseguer, J. (2015). Variant-Based Satisfiability in Initial Algebras. In *Proceedings of the 4th International Workshop for Safety-Critical Systems (FTSCS 2015)*, volume 596 of *Communications in Computer and Information Science*, pages 3–34. Springer.

Meseguer, J. (2020). Generalized Rewrite Theories, Coherence Completion, and Symbolic Methods. *Journal of Logical and Algebraic Methods in Programming*, 110.

Meseguer, J. (2021). Symbolic Computation in Maude: Some Tapas. In Fernández, M., editor, *Proc. 30th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2020)*, volume 12561 of *Lecture Notes in Computer Science*, pages 3–36. Springer.

Meseguer, J., Palomino, M., and Martí-Oliet, N. (2008). Equational Abstractions. *Theoretical Computer Science*, 403(2–3):239–264.

Meseguer, J. and Thati, P. (2007). Symbolic Reachability Analysis Using Narrowing and its Application to Verification of Cryptographic Protocols. *Higher-Order and Symbolic Computation*, 20(1–2):123–160.

Ölveczky, P. C. and Meseguer, J. (2008). The Real-Time Maude Tool. In *Proceedings of the 14th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *Lecture Notes in Computer Science*, pages 332–336. Springer.

Peemöller, B. (2017). *Normalization and Partial Evaluation of Functional Logic Programs*. PhD thesis, University of Kiel, Germany.

Plotkin, G. (1970). "a note on inductive generalization". In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press.

Roşu, G. (2017). $\mathbb{K}$: A Semantic Framework for Programming Languages and Formal Analysis Tools. In *Dependable Software Systems Engineering*, volume 50 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 186–206. IOS Press.

Rodríguez, A., Durán, F., Rutle, A., and Kristensen, L. M. (2019). Executing Multilevel Domain-Specific Models in Maude. *Journal of Object Technology*, 18(2):4:1–21.

Serbanuta, T. and Rosu, G. (2006). Computationally Equivalent Elimination of Conditions. In Pfenning, F., editor, *Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings*, volume 4098 of *Lecture Notes in Computer Science*, pages 19–34. Springer.

Slagle, J. R. (1974). Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. *Journal of the ACM*, 21(4):622–642.

Sørensen, M. H., Glück, R., and Jones, N. D. (1994). Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC. In *Proceedings of the 5th European Symposium on Programming (ESOP 1994)*, volume 788 of *Lecture Notes in Computer Science*, pages 485–500. Springer.

Viry, P. (2002). Equational Rules for Rewriting Logic. *Theoretical Computer Science*, 285(2):487–517.

Appendix A Proofs of Technical Results

Theorem 3.1 (strong completeness of topmost extension). *Let $\mathscr{R} = (\Sigma, E \uplus B, R)$ be a topmost modulo Ax rewrite theory, with $Ax \in B$, and let $\mathscr{E} = (\Sigma, E \uplus B)$ be a convergent, finite variant equational theory where B-unification is decidable. Let $\hat{\mathscr{R}} = (\hat{\Sigma}, E \uplus B, \hat{R})$ be the topmost extension of $\mathscr{R}$ and let $G = (\exists X)\ t \longrightarrow^* t'$ be a reachability goal in $\mathscr{R}$. If σ is a solution for G in $\mathscr{R}$, then there exists a term t'' such that $\{t\} \rightsquigarrow^*_{\hat{R},\mathscr{E}} \{t''\}$ with computed substitution θ and $\sigma =_{\mathscr{E}} \theta\eta$, where η is an $\mathscr{E}$-unifier of t'' and t'.*

Proof
Let $\mathscr{R} = (\Sigma, E \uplus B, R)$ be a topmost modulo Ax rewrite theory and $Ax \in B$, and let $\mathscr{E} = (\Sigma, E \uplus B)$ be a convergent, finite variant equational theory where B-unification is decidable.

Let $\hat{\mathscr{R}} = (\hat{\Sigma}, E \uplus B, \hat{R})$ be the topmost extension of $\mathscr{R}$. Let $G = ((\exists X)\ t \longrightarrow^* t')$ be a reachability goal for $\mathscr{R}$. If σ is a solution for G in $\mathscr{R}$, then $t\sigma \rightarrow^*_{R,\mathscr{E}} t'\sigma$. By Proposition 3.1, we have $\{t\}\sigma \rightarrow^*_{\hat{R},\mathscr{E}} \{t'\}\sigma$, which means that σ is a solution in $\hat{\mathscr{R}}$ for $\hat{G} = ((\exists X)\ \{t\} \longrightarrow^* \{t'\})$.

Since $\mathscr{E} = (\Sigma, E \uplus B)$ is a convergent, finite variant equational theory where B-unification is decidable, $\mathscr{E}$-unification is decidable. Furthermore, since $\hat{\mathscr{R}}$ is topmost and $\mathscr{E}$-unification is decidable, $\rightsquigarrow_{\hat{R},\mathscr{E}}$ is complete and thus there exist $\{t\} \rightsquigarrow^*_{\hat{R},\mathscr{E}} \{t''\}$ with a computed substitution θ and an $\mathscr{E}$-unifier η such that $\theta\eta =_{\mathscr{E}} \sigma$. □

Proposition 3.1 (correctness and completeness of the topmost extension). *Let $\mathscr{R} = (\Sigma, E \uplus B, R)$, with $\mathscr{E} = (\Sigma, E \uplus B)$, be a topmost modulo Ax theory and let $\hat{\mathscr{R}}$ be the topmost extension of $\mathscr{R}$. For any terms t_i and t_f of sort Config, $t_i \rightarrow^*_{R,\mathscr{E}} t_f$ iff $\{t_i\} \rightarrow^*_{\hat{R},\mathscr{E}} \{t_f\}$.*

Proof
The case when Ax =ACU has been stated in Lemma 5.3 of (Meseguer and Thati, 2007). Here, we prove the case when Ax =AC, which involves two extension rewrite rules, namely, ($\{X \otimes \lambda\} \Rightarrow \{X \otimes \rho\}$ if C) and ($\{\lambda\} \Rightarrow \{\rho\}$ if C). The remaining cases are straightforward adaptations of the following proof scheme.
We have to prove the following two implications for the case when Ax =AC:

($\rightarrow$) for any term t_i and t_f of sort *Config*, $t_i \rightarrow^*_{R,\mathscr{E}} t_f$ implies $\{t_i\} \rightarrow^*_{\hat{R},\mathscr{E}} \{t_f\}$;
($\leftarrow$) for any term t_i and t_f of sort *Config*, $\{t_i\} \rightarrow^*_{\hat{R},\mathscr{E}} \{t_f\}$ implies $t_i \rightarrow^*_{R,\mathscr{E}} t_f$.

($\rightarrow$) Assume that $t_i \rightarrow^*_{R,\mathscr{E}} t_f$, where t_i and t_f are arbitrary terms of sort *Config*. Then, $t_i \rightarrow^*_{R,\mathscr{E}} t_f$ has the form

$$t_i = t_0 \rightarrow_{R,\mathscr{E}} \ldots \rightarrow_{R,\mathscr{E}} t_{n-1} \rightarrow_{R,\mathscr{E}} t_n = t_f, \text{for some natural number } n \geq 0.$$

We proceed by induction on the length n of the rewriting sequence $t_i \rightarrow^*_{R,\mathscr{E}} t_f$.

$n = 0$. Immediate since there are no rewrite steps.

$n > 0$. By induction hypothesis, we have

$$t_i = t_0 \rightarrow^*_{R,\mathscr{E}} t_{n-1} \text{ implies } \{t_i\} = \{t_0\} \rightarrow^*_{\hat{R},\mathscr{E}} \{t_{n-1}\}. \quad \text{(A1)}$$

Thus, in order to prove ($\rightarrow$), we just need to show

$$\{t_{n-1}\} \rightarrow_{\hat{R},\mathscr{E}} \{t_n\}, \quad \text{(A2)}$$

whenever $t_{n-1} \rightarrow_{R,\mathscr{E}} t_n$. The computation step $t_{n-1} \rightarrow_{R,\mathscr{E}} t_n$ in the rewrite theory $\mathscr{R}$ can be expanded into the following rewrite sequence

$$t_{n-1} \overset{r,\sigma,w}{\rightarrow}_{R,B} \tilde{t}_{n-1} \rightarrow^*_{\vec{E},B} \tilde{t}_{n-1}\downarrow_{\vec{E},B} = t_n$$

where $r = (\lambda \Rightarrow \rho \text{ if } C) \in R$. Here, we distinguish two cases according to the value of the position $w \in Pos(t_{n-1})$: $w = \Lambda$ and $w \neq \Lambda$.

$(w = \Lambda)$ In this case, $t_{n-1} =_B \lambda\sigma$ and $\tilde{t}_{n-1} =_B \rho\sigma$ by the definition of $\rightarrow_{R,B}$. Furthermore, since $\mathscr{R}$ is topmost modulo Ax, $\lambda\sigma$ and $\rho\sigma$ have sort *Config*. From these facts, it immediately follows that

$$\{t_{n-1}\} =_B \{\lambda\sigma\} \overset{\hat{r},\sigma,\Lambda}{\rightarrow}_{\hat{R},B} \{\rho\sigma\} =_B \{\tilde{t}_{n-1}\} \rightarrow^*_{\vec{E},B} \{\tilde{t}_{n-1}\}\downarrow_{\vec{E},B} = \{t_n\}$$

with $\hat{r} = \{\lambda\} \Rightarrow \{\rho\}$ if $C \in \hat{R}$. Hence, $\{t_{n-1}\} \rightarrow_{\hat{R},\mathscr{E}} \{t_n\}$ when $w = \Lambda$.

$(w \neq \Lambda)$ Since $\mathscr{R}$ is topmost modulo Ax and $w \neq \Lambda$, there exist $u_i \in \mathscr{T}_\Sigma(\mathscr{X})$ of sort *Config*, $i = 1,\ldots,k$, with $k > 1$ such that

$$t_{n-1} = u_1 \otimes \ldots \otimes u_k$$

and $t_{n-1} \in \mathscr{T}_\Sigma(\mathscr{X})$ of sort *Config*. Now, since $t_{n-1} \overset{r,\sigma,w}{\rightarrow}_{R,B} \tilde{t}_{n-1} \rightarrow^*_{\vec{E},B} t_n$, with $Ax = AC$ and $r = (\lambda \Rightarrow \rho \text{ if } C)$,

$$\begin{aligned} t_{n-1} = u_1 \otimes \ldots \otimes u_k &=_{AC} u_{\pi(1)} \otimes \ldots \otimes u_{\pi(m)} \otimes \lambda\sigma \\ &\overset{r,\sigma,w}{\rightarrow}_{R,B} u_{\pi(1)} \otimes \ldots \otimes u_{\pi(m)} \otimes \rho\sigma \rightarrow^*_{\vec{E},B} t_n \end{aligned}$$

where $1 \leq m \leq k-1$, and $\pi\colon \{1,...,m\} \rightarrow \{1,\ldots,k\}$ is an injective function that selects a permutation of m u_i's within $u_1 \otimes \ldots \otimes u_k$. Hence, we can build the following rewrite sequence

$$\begin{aligned} \{t_{n-1}\} = \{u_1 \otimes \ldots \otimes u_k\} &=_{AC} \{u_{\pi(1)} \otimes \ldots \otimes u_{\pi(m)} \otimes \lambda\sigma\} \\ &\overset{\hat{r},\hat{\sigma},\Lambda}{\rightarrow}_{\hat{R},B} \{u_{\pi(1)} \otimes \ldots \otimes u_{\pi(m)} \otimes \rho\sigma\} \rightarrow^*_{\vec{E},B} \{t_n\} \end{aligned}$$

with $\hat{\sigma} = \sigma \cup \{X \mapsto u_{\pi(1)} \otimes \ldots \otimes u_{\pi(m)}\}$, and $\hat{r} = (\{X \otimes \lambda\} \Rightarrow \{X \otimes \rho\} \text{ if } C) \in \hat{R}$. This proves that $\{t_{n-1}\} \rightarrow_{\hat{R},\mathscr{E}} \{t_n\}$ also in the case when $w \neq \Lambda$.
Finally, by using the induction hypothesis A1 and the rewrite step A2, we easily derive the implication $(\rightarrow)$.

$(\leftarrow)$ Assume that $\{t_i\} \rightarrow^*_{\hat{R},\mathscr{E}} \{t_f\}$, where t_i and t_f are arbitrary terms of sort *Config*. Then, $\{t_i\} \rightarrow^*_{\hat{R},\mathscr{E}} \{t_f\}$ is of the form

$$\{t_i\} = \{t_0\} \rightarrow_{\hat{R},\mathscr{E}} \ldots \rightarrow_{\hat{R},\mathscr{E}} \{t_{n-1}\} \rightarrow_{\hat{R},\mathscr{E}} \{t_n\} = \{t_f\}$$

for some natural number $n \geq 0$. We proceed by induction on the length n of the computation $\{t_i\} \rightarrow^*_{\hat{R},\mathscr{E}} \{t_f\}$.

$n = 0$. Immediate, since there are no rewrite steps.

$n > 0$. This case is analogous to the proof of the inductive step of Case $(\rightarrow)$. By induction hypothesis, we have

$$\{t_i\} = \{t_0\} \rightarrow^*_{\hat{R},\mathscr{E}} \{t_{n-1}\} \text{ implies } t_i = t_0 \rightarrow^*_{R,\mathscr{E}} t_{n-1}. \tag{A3}$$

Therefore, it suffices to show that $t_{n-1} \rightarrow^*_{R,\mathscr{E}} t_n$ and combine this result with the induction hypothesis to finally prove Case ($\leftarrow$).
By hypothesis, $\{t_{n-1}\} \rightarrow_{\hat{R},\mathscr{E}} \{t_n\}$, which can be expanded into the following rewrite sequence

$$\{t_{n-1}\} \overset{\hat{r},\hat{\sigma},\Lambda}{\rightarrow}_{\hat{R},B} \{\tilde{t}_{n-1}\} \rightarrow^*_{\vec{E},B} \{\tilde{t}_{n-1}\}\downarrow_{\vec{E},B} = \{t_n\} \tag{A4}$$

where $\hat{r} \in \hat{R}$, and $\{t_{n-1}\}, \{\tilde{t}_{n-1}\}, \{t_n\} \in \mathscr{T}_{\hat{\Sigma}}(\mathscr{X})$ of sort *State*. Observe that the first rewrite step of the rewrite sequence (A4) must occur at position Λ, since the rewrite theory $\hat{\mathscr{R}}$ is topmost.
Here, we distinguish two cases according to the form of the rewrite rule $\hat{r} \in \hat{R}$ applied in $\{t_{n-1}\} \overset{\hat{r},\hat{\sigma},\Lambda}{\rightarrow}_{\hat{R},B} \{\tilde{t}_{n-1}\}$.
By Definition 2, $\hat{r}$ is either $\{\lambda\} \Rightarrow \{\rho\}$ if C or $\{X \otimes \lambda\} \Rightarrow \{X \otimes \rho\}$ if C, as $Ax = AC$ and $\{t_{n-1}\}, \{\tilde{t}_{n-1}\} \in \mathscr{T}_{\hat{\Sigma}}(\mathscr{X})$ of sort *State*.

Case ($\hat{r} = (\{\lambda\} \Rightarrow \{\rho\}$ if C)). In this case, $\{t_{n-1}\} \overset{\hat{r},\hat{\sigma},\Lambda}{\rightarrow}_{\hat{R},B} \{\tilde{t}_{n-1}\}$ assumes the following form:

$$\{t_{n-1}\} =_{AC} \{\lambda\sigma\} \overset{\hat{r},\sigma,\Lambda}{\rightarrow}_{\hat{R},B} \{\rho\sigma\} =_{AC} \{\tilde{t}_{n-1}\}.$$

Now, by Definition 2, $\lambda\sigma$ and $\rho\sigma$ are terms of sort *Config*; thus, we can also apply $r = (\lambda \Rightarrow \rho$ if $C) \in R$ to $\lambda\sigma$, thereby obtaining the following computation

$$t_{n-1} =_{AC} \lambda\sigma \overset{r,\sigma,\Lambda}{\rightarrow}_{R,B} \rho\sigma \rightarrow^*_{\vec{E},B} (\rho\sigma\downarrow_{\vec{E},B}) = t_n$$

which corresponds to $t_{n-1} \rightarrow_{R,\mathscr{E}} t_n$ when $\hat{r} = (\{\lambda\} \Rightarrow \{\rho\}$ if C).

Case ($\hat{r} = (\{X \otimes \lambda\} \Rightarrow \{X \otimes \rho\}$ if C)). In this case, $\{t_{n-1}\} \overset{\hat{r},\hat{\sigma},\Lambda}{\rightarrow}_{\hat{R},B} \{\tilde{t}_{n-1}\}$ must have the following form:

$$\{t_{n-1}\} =_{AC} \{u \otimes \lambda\hat{\sigma}\} \overset{\hat{r},\hat{\sigma},\Lambda}{\rightarrow}_{\hat{R},B} \{u \otimes \rho\hat{\sigma}\} =_{AC} \{\tilde{t}_{n-1}\}$$

where $u, \lambda\hat{\sigma}, \rho\hat{\sigma} \in \mathscr{T}(\Sigma,\mathscr{V})_{Config}$, and $\hat{\sigma} = \{X \mapsto c\} \cup \sigma$, for some substitution σ.
Now, by Definition 2, variable X does not occur in either λ or ρ; this implies that $\lambda\hat{\sigma} = \lambda\sigma$ and $\rho\hat{\sigma} = \rho\sigma$. Therefore, we can construct the following computation:

$$t_{n-1} =_{AC} u \otimes \lambda\hat{\sigma} = u \otimes \lambda\sigma \overset{r,\sigma,w}{\rightarrow}_{R,B} u \otimes \rho\sigma = u \otimes \rho\hat{\sigma} =_{AC} \tilde{t}_{n-1} \rightarrow^*_{\vec{E},B} t_n$$

where $r = (\lambda \Rightarrow \rho$ if $C) \in R$ and $w \in Pos(u \otimes \lambda\sigma)$ is the position of the term $\lambda\sigma$ inside $u \otimes \lambda\sigma$.
Hence, $t_{n-1} \rightarrow_{R,\mathscr{E}} t_n$ even in the case when $\hat{r} = (\{X \otimes \lambda\} \Rightarrow \{X \otimes \rho\}$ if C).

□