Optimizing Maude Programs via Program Specialization *

M. Alpuente[1], D. Ballis[2], S. Escobar[1], J. Meseguer[3], and J. Sapiña[1]

[1] VRAIN, Universitat Politècnica de València, Valencia, Spain
{alpuente,sescobar,sapina}@upv.es
[2] DMIF, Università degli Studi di Udine, Udine, Italy
demis.ballis@uniud.it
[3] University of Illinois at Urbana-Champaign, Urbana, IL, USA
meseguer@illinois.edu

Abstract. We develop an automated specialization framework for rewrite theories that model concurrent systems. A rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$ consists of two main components: an order-sorted equational theory $\mathscr{E} = (\Sigma, E \uplus B)$ that defines the system states as terms of an algebraic data type and a term rewriting system R that models the concurrent evolution of the system as state transitions. Our main idea is to partially evaluate the underlying equational theory $\mathscr{E}$ to the specific calls required by the rewrite rules of R in order to make the system computations more efficient. The specialization transformation relies on *folding variant narrowing*, which is the symbolic operational engine of Maude's equational theories. We provide three instances of our specialization scheme that support distinct classes of theories that are relevant for many applications. The effectiveness of our method is finally demonstrated in some specialization examples.

1 Introduction

Maude is a high-performance, concurrent functional language that efficiently implements Rewriting Logic (RWL), a logic of change that unifies a wide variety of models of concurrency [38]. Maude is endowed with advanced symbolic reasoning capabilities that support a high-level, elegant, and efficient approach to programming and analyzing complex, highly nondeterministic software systems [24]. Maude's symbolic capabilities are based on equational unification and *narrowing*, a mechanism that extends term rewriting by replacing pattern matching with unification [49], and they provide advanced logic programming features such as unification modulo user-definable equational theories and symbolic reachability analysis in rewrite theories. Intricate computing problems may be effectively and naturally solved in Maude thanks to the synergy of these recently developed symbolic capabilities and classical Maude features, such as: (i) rich type structures with sorts (types), subsorts, and overloading; (ii) equational rewriting modulo various combinations of axioms such as associativity (A), commutativity (C), and identity (U); and (iii) classical reachability analysis in rewrite theories.

Partial evaluation (PE) is a program transformation technique that automatically specializes a program to a part of its input that is known statically (at *specialization* time) [23, 33]. Partial evaluation conciliates generality with efficiency by providing automatic program optimization.

* This work has been partially supported by the EC H2020-EU grant agreement No. 952215 (TAILOR), the EU (FEDER) and the Spanish MCIU under grant RTI2018-094403-B-C32, by Generalitat Valenciana under grant PROMETEO/2019/098, and by NRL under contract number N00173-17-1-G002. Julia Sapiña has been supported by the Generalitat Valenciana APOSTD/2019/127 grant.

In the context of logic programming, partial evaluation is often called partial deduction and allows to not only instantiate input variables with constant values but also with terms that may contain variables, thus providing extra capabilities for program specialization [35, 36]. Early instances of this framework implemented partial evaluation algorithms for different narrowing strategies, including lazy narrowing [12], innermost narrowing [15], and needed narrowing [2, 16].

The *Narrowing-driven partial evaluation* (NPE) scheme for functional logic program specialization defined in [15, 14] and implemented [1] in is strictly more powerful than the PE of both logic programs and functional programs thanks to combining functional reduction with the power of logic variables and unification by means of narrowing. In the *Equational narrowing-driven partial evaluation* (EQNPE) scheme of [7], this enhanced specialization capability was extended to the partial evaluation of order-sorted equational theories. Given a signature Σ of program operators together with their type definition, an equational theory $\mathscr{E} = (\Sigma, E \uplus B)$ combines a set E of equations (that are implicitly oriented from left to right and operationally used as simplification rules) on Σ and a set B of commonly occurring axioms (which are implicitly expressed in Maude as operator attributes using the `assoc`, `comm`, and `id`: keywords) that are essentially used for B-matching[4]. To be executable in Maude, the equational theory $\mathscr{E}$ is required to be *convergent* (i.e., the equations E are confluent, terminating, sort-decreasing, and coherent modulo B). This ensures that every input expression t has one (and only one) *canonical* form $t\downarrow_{\vec{E},B}$ up to B-equality.

This paper addresses the specialization of *rewrite theories* $\mathscr{R} = (\Sigma, E \uplus B, R)$ whose system transitions are specified by rewrite rules R on top of an underlying equational theory $\mathscr{E} = (\Sigma, E \uplus B)$. Altogether, the rewrite theory $\mathscr{R}$ specifies a concurrent system that evolves by rewriting states using *equational rewriting*, i.e., rewriting with the rewrite rules in R *modulo* the equations and axioms in $\mathscr{E}$ [38]. In Maude, rewrite theories can also be *symbolically* executed by narrowing at *two levels*: (i) narrowing with the (typically non-confluent and non-terminating) rules of R modulo $\mathscr{E} = (\Sigma, E \uplus B)$; and (ii) narrowing with the (explicitly) oriented equations $\vec{E}$ modulo the axioms B. They both have practical applications: (i) narrowing with R modulo $\mathscr{E} = (\Sigma, E \uplus B)$ is useful for solving *reachability goals* [43] and *logical model checking* [29]; and (ii) narrowing with $\vec{E}$ modulo B is useful for $\mathscr{E}$-unification and variant computation [31]. Both levels of narrowing should meet some conditions: (i) narrowing with R modulo $\mathscr{E}$ is performed in a "topmost" way (i.e., the rules in R rewrite the global system state) and there must be a finitary equational unification algorithm for $\mathscr{E}$; and (ii) narrowing with $\vec{E}$ modulo B requires that B is a theory with a finitary unification algorithm and that $\mathscr{E}$ is convergent. When $(\Sigma, E \uplus B)$ additionally has the property that a finite complete set of most general *variants*[5] exists for each term, known as the *finite variant property* (FVP), $\mathscr{E}$-unification is finitary and *topmost* narrowing with R modulo the equations and axioms can be effectively performed.

For variant computation and (variant-based) $\mathscr{E}$-unification, the *folding variant narrowing* (or FV-narrowing) strategy of [31] is used in Maude, whose termination is guaranteed for theories that satisfy the FVP (also known as *finite variant theories*). Another important class of rewrite theories are those that satisfy the so-called *constructor finite variant property* (CFVP), i.e., they have a finite number of most general *constructor* variants [40]. Many relevant theories have

[4] For example, assuming a commutative binary operator $*$, the term $s(0) * 0$ matches within the term $X * s(Y)$ *modulo* the commutativity of symbol $*$ with matching substitution $\{X/0, Y/0\}$.

[5] A *variant* [22] of a term t in the theory $\mathscr{E}$ is the canonical (i.e., irreducible) form of $t\sigma$ in $\mathscr{E}$ for a given substitution σ; in symbols, it is represented as the pair $(t\sigma\downarrow_{\vec{E},B}, \sigma)$.

the FVP, including theories of interest for Boolean satisfiability and theories that give algebraic axiomatizations of cryptographic functions used in communication protocols, where FVP and CFVP are omnipresent. CFVP is implied by FVP together with *sufficient completeness* modulo axioms (SC); that is, all function calls (i.e., input terms) reduce to *values* (i.e., ground constructor terms [27, 32]).

Given the rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$, the key idea of our method is to specialize the underlying equational theory $\mathscr{E} = (\Sigma, E \uplus B)$ to the precise use that the rules of R make of the operators that are defined in $\mathscr{E}$. This is done by partially evaluating $\mathscr{E}$ with respect to the *maximal* (or outermost) function calls that can be retrieved from the rules of R, in such a way that $\mathscr{E}$ gets rid of any possible over-generality and the functional computations given by $\mathscr{E}$ are thus greatly compacted. Nevertheless, while the transformation highly contracts the system states, we deliberately avoid making any states disappear since both reachability analysis and logical model checking generally require the whole search space of rewrite theories to be searched (i.e., all system states).

Our specialization algorithm follows the classic control strategy of logic specializers [36], with two separate components: 1) the local control (managed by an unfolding operator [13]) that avoids infinite evaluations and is responsible for the construction of the residual equations for each specialized call; and 2) the global control (managed by an abstraction operator) that avoids infinite iterations of the partial evaluation algorithm and decides which specialized functions appear in the transformed theory. A post-processing compression transformation is finally performed that highly compacts the functional computations occurring in the specialized rewrite theory while keeping the system states as reduced as possible.

We provide three different implementations of the unfolding operator based on FV-narrowing that may include some distinct extra control depending on the FVP/CFVP behavior of the equational theory $\mathscr{E}$. More precisely, we distinguish the following three cases:

1. $\mathscr{E}$ does not fulfill the finite variant property: a subsumption check is performed at each FV-narrowing step that compares the current term with all previous narrowing redexes in the same derivation. The subsumption checking relies on the *order-sorted equational homeomorphic embedding* relation of [8] that ensures all infinite FV-narrowing computations are safely stopped;
2. $\mathscr{E}$ satisfies the finite variant property: FV-narrowing trees are always finite for any input term, and therefore they are completely deployed; and
3. $\mathscr{E}$ satisfies the finite variant property and is also sufficiently complete: we supplement unfolding with an extra "sort downgrading" transformation in the style of [41] that safely rules out variants that are not constructor terms. This means that all specialized calls get *totally evaluated* and the maximum compression is achieved, thereby dramatically reducing the search space for the construction of the specialized theories.

It is worth noting that our specialization system is based on the Maude's narrowing engine and, hence, it respects the limitations and applicability conditions of the current narrowing implementation. In particular, Maude's narrowing (and thus our specializer) does not support conditional equations, built-in operators and special equational attributes (e.g., `owise`). However, advances in narrowing and unification for Maude will enlarge the class of rewrite theories that our specialization technique handles.

Plan of the paper. In Section 2, we introduce a leading example that illustrates the optimization of rewrite theories that we can achieve by using our specialization technique, which we formalize

in Section 3. In Section 4, we focus on finite variant theories that are sufficiently complete and we demonstrate that both properties, SC and FVP, are preserved by our transformation scheme. In Section 5, we instantiate the specialization scheme for the three classes of equational theories already mentioned: theories whose terms may have an infinite number of most general variants, or a finite number of most general variants, or a finite number of most general constructor variants. The proposed methodology is illustrated in Section 6 by describing several specializations of the bank account specification of Section 2 and by presenting some experiments with the partial evaluator Presto that implements our technique. In Section 7, we discuss some related work and we conclude. The complete code of the specialized examples is given in the Appendix, which is only meant to facilitate the review and is not a part of the paper.

2 A Leading Example

Let us motivate the power of our specialization scheme by optimizing a simple rewrite theory that is inspired by [41]. The considered example has been engineered to fulfill the conditions for the applicability of all the three instances of our specialization framework.

Example 1. Consider a rewrite theory that specifies a bank account system with managed accounts. The system automates a simple investment model for the beginner investor that, whenever the account balance exceeds a given investment threshold, the excess balance is automatically moved to investment funds. The system allows deposits and withdrawals to occur nondeterministically, where each withdrawal occurs in two steps: the withdrawal is initiated through a withdrawal request provided that the amount to be withdrawn is less than or equal to the current account balance. Later on, the actual withdrawal is completed. On the contrary, deposits are single-step operations that need to consume explicit deposit messages to be performed. This asymmetric behaviour is due to the fact that the amount in a deposit operation is unbounded, while a withdrawal request is always limited by the account balance. For simplicity, the external operation of the investment portfolio is not considered in the model.

A managed account is modelled as a term

```
< bal: n pend: x overdraft: b threshold: h funds: f >
```

where `n` is the current balance, `x` is the amount of money that is currently pending to be withdrawn, `b` is a Boolean flag that indicates whether or not the account is in the red, `h` is a fixed upper threshold for the account balance, and `funds` represents the amount to be invested by the account manager. Messages of the form `d(m)` and `w(m)` specify deposit and withdrawal operations, where `m` is the amount of money to be, respectively, deposited and withdrawn. A bank account state (or simply state) is a pair `act # msgs`, where `act` is an account and `msgs` a multiset of messages. Monetary values in a state are specified by natural numbers in Presburger's style[6]. State transitions are formalized by the three rewrite rules in Figure 1 (namely, `w-req`, `w`, and `dep`) that respectively implement withdrawal requests, (actual) withdrawals, and deposits.

The intended semantics of the three rules is as follows. The rule `w-req` non-deterministically requests to draw money whenever the account balance covers the request. The requested amount `m` is added to the amount of pending withdraw requests and the withdraw message `w(m)` is generated. The rule `w` implements actual withdrawal of money from the account. When the

[6] In [40], natural numbers are encoded by using two constants 0 and 1 and an ACU operator + so that a natural number is either the constant 0 or a finite sequence `1 + 1 ... + 1`.

```
rl [w-req] : < bal: n + m + x pend: x overdraft: false threshold: n + m + x + h funds: f > # msgs
               => < bal: n + m + x pend: x + m overdraft: false threshold: n + h funds: f > #
                  w(m) , msgs .

rl [w] : < bal: n pend: x overdraft: false threshold: n + h funds: f > # w(m),msgs
          => [ m > n ,
               < bal: n pend: x overdraft: true threshold: n + h funds: f > # msgs,
               < bal: n - m pend: x - m overdraft: false threshold: n + h funds: f > # msgs ] .

rl [dep] : < bal: n pend:x overdraft: false threshold: n + m + h funds: f > # d(m),msgs .
            => << bal: n + m pend: x overdraft: false threshold: n + m + h funds: f >> # msgs .
```

Fig. 1. Rewrite rules that model a simple bank account system.

balance is not enough, the account is blocked by setting `overdraft` to `true` and the withdrawal attempt fails (for simplicity, the excess of balance that is moved to investment funds is never moved back). If not in overdraft, money can be deposited in the account by processing the deposit message `d(m)` using rule `dep`.

```
--- Encoding of natural numbers with constants 0,1 and ACU operator +
ops 0 1 : -> Nat [ctor] .
op _+_ : Nat Nat -> Nat [ctor assoc comm id: 0] .

---  greater-than operator
op _>_ : Nat Nat -> Bool .
eq m + n + 1 > n = true [variant] .
eq n > n + m = false [variant] .

--- monus  function
op _-_ : Nat Nat -> Nat .
eq n - (n + m) = 0 [variant] .
eq (n + m) - n = m [variant] .

--- if-then-else construct
op [_,_,_] : Bool State State -> State .
eq [ true,s,s' ] = s [variant] .
eq [ false,s,s' ] = s' [variant] .

--- Account balance simplification
op << bal:_ pend:_ overdraft:_ threshold:_ funds:_ >> : Nat Nat Bool Nat Nat -> Account .
eq << bal: n + h pend: m overdraft: b threshold: h funds: f >>
       = << bal: n pend: m overdraft: false threshold: h funds: f + 1 >> [variant] .
eq << bal: n pend: m overdraft: false threshold: n + h funds: f >>
       =  < bal: n pend: m overdraft: false threshold: n + h funds: f > [variant] .
```

Fig. 2. Companion equational theory for the bank account system.

The auxiliary functions that are used by the three rules implement the pre-agreed, automated investment policy for a given threshold. They update the account's state by means of an equational theory whose operators and equations are shown in Figure 2. The equational theory extends Presburger's arithmetic with the operators over natural numbers `_>_` and `_-_`, together with the if-then-construct `[_,_,_]` and an auxiliary version « . . . » of the operator < . . . > that ensures that the current balance `n` is below the current threshold `h`; otherwise, it sets the balance to

`n mod h` and increments the funds by `n div h`, where `div` is the division for natural numbers and `mod` is the remainder of the division; both operations are encoded by successive subtractions. Roughly speaking, this operator allows money to be moved from the `bal` attribute to the `funds` attribute, whenever the balance exceeds the threshold `h`. Note that the amount of money in the investment funds is measured in `h` units (1, 2, ...), which indicate the client's wealth category (the higher the category, the greater the investment advantages). The attribute `variant` is used to identify the equations to be considered by the FV-narrowing strategy.

The considered equational theory has neither the FVP nor the CFVP since, for instance, the term « `bal: n pend: x overdraft: false threshold: h funds: f` » has an infinite number of (incomparable) most general (constructor) variants

```
( < bal: n' pend: x overdraft: false threshold: h funds: f + 1 >,{n/(n' + h)})
                ⋮
( < bal: n' pend: x overdraft: false threshold: h funds: f + 1 + ...+ 1 >,{n/(n' + h +...+ h)})
```

```
eq f0($4,$5 + $4,$1 + $4,$6,$2,$3)
      = < bal: $5 pend: $1 overdraft: false threshold: $5 + $4 + $6 funds: $2 > # $3 [ variant ] .
eq f0($5 + $3,$5 + $3 + $4,$5,$6,$1,$2)
      = < bal: $4 pend: 0 overdraft: false threshold: $5 + $3 + $4 + $6 funds: $1 > # $2 [ variant ] .
eq f0(1 + $5 + $4,$5,$1,$6,$2,$3)
      = < bal: $5 pend: $1 overdraft: true threshold: $5 + $6 funds: $2 > # $3 [ variant ] .

rl [w-req-s] : < bal: n + x + m pend: x overdraft: false threshold: n + h funds: f > # msgs
               => < bal: n + x + m pend: x + m overdraft: false threshold: n + h funds: f > # msgs,w(m) .
rl [w-s] : < bal: n pend: x overdraft: false threshold: n + h funds: f > # msgs,w(m)
               => f0(m,n,x,h,f,msgs) .
rl [dep-s] : < bal: n pend: x overdraft: false threshold: n + m + h funds: f > # msgs,d(m)
               => < bal: n + m pend: x overdraft: false threshold: n + m + h funds: f > # msgs .
```

Fig. 3. Specialized bank account system.

Nonetheless, this is not an obstacle to applying our specialization technique as it can naturally handle theories that may not fulfill the FVP, whereas the total evaluation method of [41] can only be applicable to theories that satisfy both, FVP and SC. Actually, in this specific case, the application of our technique generates the highly optimized rewrite theory that is shown in Figure 3, which improves several aspects of the input bank account theory. First, the specialized equational theory is much more compact (3 equations vs 8 equations); indeed, all of the original defined functions are replaced by a much simpler, newly introduced function `f0` that is used to update bank accounts. Furthermore, `f0` exhibits an optimal performance since any call is normalized in just one reduction step, thereby providing fast bank account updates. Actually, the partially evaluated equational theory runs one order of magnitude faster than the original one, as shown in Section 6. This happens because the right-hand sides of the equations defining `f0` are constructor terms; hence, they do not contain any additional function call that must be further simplified.

Second, the specialized equational theory satisfies the FVP, which enables $\mathscr{E}$-unification, complete variant generation, and also symbolic reachability in the specialized bank account system via narrowing with rules modulo $\mathscr{E}$ while they were not feasible in the original rewrite theory.

Third, the original rewrite rules have also been simplified: the new deposit rule `dep-s` gets rid of the operator «`bal:_ pend:_ overdraft:_ threshold:_ funds:_ `», while the rewrite rule `w-s` replaces the complex nested call structure in the right-hand side of the rule `w` with a much simpler and equivalent call to function `f0`.

A detailed account of the specialization process for this example is given in Section 6.

3 Specialization of Rewrite Theories

In this section, we briefly present the specialization procedure $\text{NPER}_{\mathscr{A}}^{\mathscr{U}}$, 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 R. The procedure $\text{NPER}_{\mathscr{A}}^{\mathscr{U}}$ extends the equational, narrowing-driven partial evaluation algorithm $\text{EQNPE}_{\mathscr{A}}^{\mathscr{U}}$ of [7], which applies to equational theories and is parametric on an unfolding operator $\mathscr{U}$ that is used to construct finite narrowing trees for any given expression, and on an abstraction operator $\mathscr{A}$ that guarantees global termination.

3.1 Narrowing and Folding Variant Narrowing in Maude

Equational, $(R, E \uplus B)$-narrowing computations are natively supported by Maude version 3.0 for unconditional rewrite theories. Before explaining how narrowing computations are handled within our framework, let us introduce some technical notions and notations that we need.

Let Σ be a *signature* that includes typed operators (also called function symbols) of the form $f\colon s_1 \dots s_m \to s$ where s_i, 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'). Σ is assumed to be *preregular*, so each term t has a least sort under $<$, denoted $ls(t)$. Binary operators in Σ may have an axiom declaration attached that specifies any combinations of algebraic laws such as associativity (`assoc`), commutativity (`comm`), and identity (`id`). By $ax(f)$, we denote the set of algebraic axioms for the operator f. By $\mathscr{T}_\Sigma(\mathscr{X})$, we denote the usual non-ground term algebra built over Σ and the set of (typed) variables $\mathscr{X}$. By $\mathscr{T}_\Sigma$, we denote the ground term algebra over Σ. By notation $x : s$, we denote a variable x with sort s. Any expression $\overline{t_n}$ denotes a finite sequence $t_1 \dots t_n$, $n \geq 0$, of terms. 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 $\mathscr{X}$ to $\mathscr{T}_\Sigma(\mathscr{X})$. Substitutions are written as $\sigma = \{X_1 \mapsto t_1, \dots, X_n \mapsto t_n\}$. The identity substitution is denoted by *id*. Substitutions are homomorphically extended to $\mathscr{T}_\Sigma(\mathscr{X})$. The application of a substitution σ to a term t is denoted by $t\sigma$. The restriction of σ to a set of variables $V \subset \mathscr{X}$ 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 $t = t'$, where $t, t' \in \mathscr{T}_{\Sigma,\mathsf{s}}(\mathscr{X})$ for some sort $\mathsf{s} \in \mathsf{S}$. An equational theory is a pair $(\Sigma, E \uplus B)$ that consists of a signature Σ, a set E of Σ-equations, and a set B of equational axioms (e.g., associativity, commutativity, and identity) declared for some binary operators in Σ. The equational theory $\mathscr{E}$ induces a congruence relation $=_{\mathscr{E}}$ on $\mathscr{T}_\Sigma(\mathscr{X})$.

A term t is more (or equally) general than t' modulo $\mathscr{E}$, denoted by $t \leq_{\mathscr{E}} t'$, if there is a substitution γ such that $t' =_{\mathscr{E}} t\gamma$. A substitution θ is more (or equally) general than σ modulo $\mathscr{E}$,

denoted by $\theta \leq_{\mathscr{E}} \sigma$, if there is a substitution γ such that $\sigma =_{\mathscr{E}} \theta\gamma$, i.e., for all $x \in \mathscr{X}, x\sigma =_{\mathscr{E}} x\theta\gamma$. Also, $\theta \leq_{\mathscr{E}} \sigma\ [V]$ iff there is a substitution γ such that, for all $x \in V, x\sigma =_{\mathscr{E}} x\theta\gamma$. An *$\mathscr{E}$-unifier* for a Σ-equation $t = t'$ is a substitution σ such that $t\sigma =_{\mathscr{E}} t'\sigma$.

Similarly to equational rewriting, where syntactic pattern-matching is replaced with matching modulo $\mathscr{E}$ (or $\mathscr{E}$-matching), in equational narrowing syntactic unification is replaced by *equational* unification (or $\mathscr{E}$-unification). More precisely, in a topmost[7] rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$, with $\mathscr{E} = (\Sigma, E \uplus B)$, equational narrowing is supported in Maude by means of a *three-layer* machinery [21]:

1. An $(R, E \uplus B)$-narrowing step from s to t with a rule $l \Rightarrow r$ in R can be performed iff there is a $\mathscr{E}$-unifier θ of the equation $s = l$ such that $t = r\theta$.
2. In turn, each $\mathscr{E}$-unification problem $s =^?_{\mathscr{E}} l$ of Point 1 is solved by using *folding variant* narrowing in the equational theory $\mathscr{E}$ that computes a finite, minimal and complete set of $\mathscr{E}$-unifiers for $s = l$ under suitable requirements [31]. Following [44], this is done by *equationally* narrowing the term $s =?= l$ (that encodes the unification problem $s =^?_{\mathscr{E}} l$) to an extra constant tt for denoting *success* in the rewrite theory $\mathscr{R}_0 = (\Sigma \cup \{=?=, tt\}, B, \vec{E} \cup \{\varepsilon\})$, where the extra[8] rewrite rule $\varepsilon = (X =?= X \Rightarrow tt)$ has been added to $\vec{E}$ in order to mimic unification of two terms (modulo B) as a narrowing step[9] that uses ε.
3. For each folding variant narrowing step using a rule in $\vec{E}$ modulo B in Point 2, B-unification algorithms are employed, allowing any combination of symbols that satisfy any combination of associativity, commutativity, and identity axioms [25].

Example 2. Consider the (partial) specification of integer numbers defined by the equations $E = \{\mathsf{X} + 0 = \mathsf{X}, \mathsf{X} + \mathsf{s}(\mathsf{Y}) = \mathsf{s}(\mathsf{X} + \mathsf{Y}), \mathsf{p}(\mathsf{s}(\mathsf{X})) = \mathsf{X}, \mathsf{s}(\mathsf{p}(\mathsf{X})) = \mathsf{X}\}$, where variables X, Y are of sort `Int`, operators `p` and `s` respectively stand for the predecessor and successor functions, and B contains the commutativity axiom $\mathsf{X} + \mathsf{Y} = \mathsf{Y} + \mathsf{X}$. Also consider that the program signature Σ contains a binary state constructor operator $||\_,\_||$: `Int Int` $\rightarrow$ `State` for a new sort `State` that models a simple network of processes that are either performing a common task (denoted by the first component of the state) or have finished the task (denoted by the second component). The system state $t = ||\,\mathsf{s}(0), \mathsf{s}(0) + \mathsf{p}(0)\,||$ can be rewritten to $||\,0, \mathsf{s}(0)\,||$ (modulo the equations of E and the commutativity of $+$) using the following rule that specifies the system dynamics:

$$||\,\mathsf{A}, \mathsf{B}\,|| \Rightarrow ||\,\mathsf{p}(\mathsf{A}), \mathsf{s}(\mathsf{B})\,||, \text{ where } \mathsf{A} \text{ and } \mathsf{B} \text{ are variables of sort } \texttt{Int} \tag{1}$$

Also, a (topmost) narrowing reachability goal from $||\,\mathsf{V} + \mathsf{V}, 0 + \mathsf{V}\,||$ to $||\,\mathsf{p}(0), \mathsf{s}(0)\,||$ succeeds (in one step) with computed substitution $\{\mathsf{V} \mapsto 0\}$, which is essentially calculated by first computing an $\mathscr{E}$-unifier σ of the input term $||\,\mathsf{V} + \mathsf{V}, 0 + \mathsf{V}\,||$ and the left-hand side $||\,\mathsf{A}, \mathsf{B}\,||$ of rule (1), $\sigma = \{\mathsf{A}/(\mathsf{V} + \mathsf{V}), \mathsf{B}/\mathsf{V}\}$. Second, an $\mathscr{E}$-unifier σ' is computed between the instantiated right-hand side $||\,\mathsf{p}(\mathsf{V} + \mathsf{V}), \mathsf{s}(\mathsf{V})\,||$ and the target state $||\,\mathsf{p}(0), \mathsf{s}(0)\,||$, $\sigma' = \{\mathsf{V} \mapsto 0\}$. Third, the composition $\sigma\sigma' = \{\mathsf{A} \mapsto 0 + 0, \mathsf{B} \mapsto 0, \mathsf{V} \mapsto 0\}$ is simplified into $\{\mathsf{A} \mapsto 0, \mathsf{B} \mapsto 0, \mathsf{V} \mapsto 0\}$ and finally restricted to the variable V in the input term, yielding $\{\mathsf{V} \mapsto 0\}$. Note that this narrowing

[7] Besides the topmost assumption for $\mathscr{R}$, we also consider the classical executability restriction that the set R of rules is coherent with E modulo B (intuitively, this ensures that a rewrite step with R can always be postponed in favor of deterministically rewriting with E modulo B).

[8] In an order-sorted setting, multiple equations are actually used to cover any possible sort in $\mathscr{R}$.

[9] For example, by using ε, the term $s(0) * 0 =?= U * s(V)$ FV-narrows to tt (modulo commutativity of $*$), and the computed narrowing substitution does coincide with the unifier modulo commutativity of the two argument terms, i.e., $\{U \mapsto 0, V \mapsto 0\}$.

derivation might signal a possible programming error in rule (1) since the number of processes in the first component of the state becomes negative.

The main idea of folding variant narrowing is to "fold" the search space of all FV-narrowing computations by using subsumption modulo B. That is, folding variant narrowing avoids computing any variant that is a substitution instance modulo B of a more general variant. Note that this notion is quite different from the classical folding operation of Burstall and Darlington's fold/unfold transformation scheme [20], where unfolding is essentially the replacement of a call by its body, with appropriate substitutions, and folding is the inverse transformation, i.e., the replacement of some piece of code by an equivalent function call. In [31], folding variant narrowing was proved to be complete and minimal for variant generation w.r.t. $(\vec{E},B)$-normalized substitutions and it terminates for all inputs provided that the theory has the FVP.

FV-narrowing derivations correspond to sequences $t_0 \leadsto_{\sigma_0,\vec{e}_0,B} t_1 \leadsto_{\sigma_1,\vec{e}_1,B} \cdots \leadsto_{\sigma_n,\vec{e}_{n-1},B} t_n$, where $t \leadsto_{\sigma,\vec{e},B} t'$ (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 *equational unifier* σ. Assuming that the initial term t is normalized, each step $t \leadsto_{\sigma,\vec{e},B} t'$ (or 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). The composition $\sigma_0\sigma_1\sigma_{n-1}$ of all the unifiers along a narrowing sequence leading to t_n (restricted to the variables of t_0) is the computed substitution of this sequence. The set of all FV-narrowing computations for a term t in $\mathscr{E}$ can be represented as a tree-like structure, denoted by $VN^{\circlearrowleft}_{\mathscr{E}}(t)$, that we call the FV-narrowing tree of t in $\mathscr{E}$.

An equational theory has the *finite variant property* (FVP) (or it is called a *finite variant theory*) iff there is a finite and complete set of most general variants for each term. Intuitively, the $(\vec{E},B)$-variants of t are the "irreducible patterns" $(t\sigma)_{\downarrow_{\vec{E},B}}$ to which t can be narrowed, with computed substitution σ, by applying the oriented equations $\vec{E}$ modulo B. For instance, there is an infinite number of variants for the term `(0 + Y:Int)` in the equation theory of Example 2; e.g., $(\mathtt{Y{:}Int}, id)$, $(\mathtt{0}, \{\mathtt{Y{:}Int} \mapsto \mathtt{0}\})$,$(\mathtt{s(0)}, \{\mathtt{Y{:}Int} \mapsto \mathtt{s(0)}\})$, $(\mathtt{s(Z{:}Int)}, \{\mathtt{Y{:}Int} \mapsto \mathtt{s(Z{:}Int)}\})$, $(\mathtt{p(0)}, \{\mathtt{Y{:}Int} \mapsto \mathtt{p(0)}\})$, ...

A preorder relation of generalization between variants provides a notion of *most general variant* and also a notion of completeness of a set of variants. Formally, 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 $t \leq_{\mathscr{E}} t'$ and $\sigma \leq_{\mathscr{E}} \sigma'$. For the term $0+\mathtt{Y{:}Int}$, the most general variant is $(Y : \mathtt{Int}, id)$ since any other variant can be obtained by equational instantiation.

Example 3. Consider the definition of the (associative and commutative) Boolean conjunction operator $\wedge$ given by $E = \{\mathtt{X} \wedge \mathtt{true} = \mathtt{X}, \mathtt{X} \wedge \mathtt{false} = \mathtt{false}\}$, where variable `X` belongs to sort `Bool` and constants `true` and `false` stand for the corresponding Boolean values. There are five most general variants modulo associativity and commutativity for the term $\mathtt{X} \wedge \mathtt{Y}$, which are: $\{(\mathtt{X} \wedge \mathtt{Y}, \mathtt{id}), (\mathtt{Y}, \{\mathtt{X} \mapsto \mathtt{true}\}), (\mathtt{X}, \{\mathtt{Y} \mapsto \mathtt{true}\}), (\mathtt{false}, \{\mathtt{X} \mapsto \mathtt{false}\}), (\mathtt{false}, \{\mathtt{Y} \mapsto \mathtt{false}\})\}$.

The theory of Example 3 satisfies the FVP, whereas the equational theory of Example 2 does not have the FVP since there is an infinite number of most general variants for the term $\mathtt{X{:}Int} + \mathtt{Y{:}Int}$. It is generally undecidable whether an equational theory has the FVP [19]; a semi-decision procedure is given in [39] (and implemented in [9]) that works well in practice and another technique based on the dependency pair framework is given in [31]. The procedure

in [39] works by computing the variants of 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); the theory does have the FVP iff there is a finite number of most general variants for every such term [39].

3.2 Partial Evaluation of Equational Theories

Given a convergent equational theory $\mathscr{E} = (\Sigma, E \uplus B)$ and a set Q of terms (henceforth called *specialized calls*), we define a transformation $\text{EQNPE}^{\mathscr{U}}_{\mathscr{A}}$ that derives a new equational theory $\mathscr{E}'$ which computes the same answers (and values) as $\mathscr{E}$ for any input term t that is a recursive instance (modulo B) of the specialized calls in Q. This means that all of the subterms of t (including itself) are a substitution instance of some term in Q. The transformation $\text{EQNPE}^{\mathscr{U}}_{\mathscr{A}}$ has two parameters, an *unfolding operator* $\mathscr{U}$ and an *abstraction operator* $\mathscr{A}$, whose precise meaning is clarified below.

The equational theory $\mathscr{E}$ to be specialized is *decomposed as a simple rewrite theory* $\vec{\mathscr{E}} = (\Sigma, B, \vec{E})$ (henceforth $\vec{\mathscr{E}}$ is called a *decomposition* of $\mathscr{E}$), whose only equations are the equational axioms in B and where the equations in E are explicitly oriented from left to right as the set $\vec{E}$ of rewrite rules. The axioms B satisfy the following extra assumptions [30]: 1) *regularity*, i.e., for each $t = t'$ in B, we have that the set of variables in t and t' is the same, 2) *linearity*, i.e., for each $t = t'$ in B, each variable occurs only once in t and in t'; 3) *sort-preservation*, i.e., for each $t = t'$ in B and substitution σ, $ls(t\sigma) = ls(t'\sigma)$, and furthermore, all variables in t and t' have a common top sort; and 4) B has a finitary and complete unification algorithm, which implies that B-matching is decidable.

The transformation consists of iterating two consecutive actions:

i) Symbolic execution (*Unfolding*). A finite, possibly partial folding variant narrowing tree for each input term t of Q[10] 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 FV-narrowing tree.

ii) Search for regularities (*Abstraction*). In order to guarantee that all calls that may occur at runtime are *covered* by the specialization, every (sub-)term in any leaf of the tree must be *equationally closed* w.r.t. Q. This notion extends the classical PD closedness by:
   1) considering B-equivalence of terms;
   2) considering a natural partition of the signature as $\Sigma = \mathscr{D} \uplus \Omega$, where Ω are the *constructor* symbols, which are used to define the (irreducible) values of the theory, and $\mathscr{D} = \Sigma \setminus \Omega$ are the *defined* symbols, which are evaluated away by equational rewriting.
   3) recursing over the term structure (in order to handle nested function calls). Roughly speaking, a term u is equationally 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 equationally closed w.r.t. $Q = \{a \bullet X, Y \bullet a\}$ or $\{X \bullet Y\}$, but it is not with Q being $\{a \bullet X\}$; however, it would be closed if $\bullet$ were commutative.

[10] For simplicity, we assume that Q is normalized w.r.t. the equational theory $\mathscr{E}$. If this were not the case, for each $t \in Q$ that is not in canonical form such that $t\downarrow_{\vec{E},B} = C(\overline{t_i})$, where $C(\ )$ is the (possibly empty) constructor context of $t\downarrow_{\vec{E},B}$ and $\overline{t_i}$ are the maximal calls in $t\downarrow_{\vec{E},B}$, we would replace t in Q with the normalized terms $\overline{t_i}$ and add a suitable "bridge" equation $t = C(\overline{t_i})$ to the resulting specialization.

Given the set $\mathscr{L}$ of leaves in the FV-narrowing trees for the partially evaluated calls in Q, in order to properly add to Q the non-closed (sub-)terms occurring in the terms of $\mathscr{L}$, an abstraction operator $\mathscr{A}(Q,\mathscr{L},B)$ is applied that yields a new set of terms which may need further evaluation. The abstraction operator $\mathscr{A}(Q,\mathscr{L},B)$ ensures that the resulting set of terms "covers" (modulo B) the calls previously specialized and that equational closedness modulo B is preserved throughout successive abstractions.

Steps i) and ii) are iterated as long as new terms are generated, and the abstraction operator $\mathscr{A}$ guarantees that only finitely many expressions are evaluated, thus ensuring global termination.

Note that the algorithm does not explicitly compute a partially evaluated equational theory. It does so implicitly, by computing a (generally augmented) set Q' of partially evaluated terms that unambiguously determine the desired partial evaluation of the equations in E as the set E' of *resultants* $t\sigma = t'$ associated with the derivations in the narrowing tree from a root $t \in Q'$ to a 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 GENTHEORY$(Q',(\Sigma,E \uplus B))$ that delivers the partially evaluated equational theory $\mathscr{E}' = (\Sigma',E' \uplus B')$ univocally determined by Q' and the original equational theory $\mathscr{E} = (\Sigma,E \uplus B)$, with $\Sigma' = \Sigma$ and $B' = B$.

3.3 The NPER$^{\mathscr{U}}_{\mathscr{A}}$ Scheme for the Specialization of Rewrite Theories

The specialization of the (topmost) rewrite theory $\mathscr{R} = (\Sigma,E \uplus B,R)$ is achieved by partially evaluating its underlying equational theory $\mathscr{E} = (\Sigma,E \uplus B)$ w.r.t. the rules R, which is done by using the partial evaluation procedure EQNPE$^{\mathscr{U}}_{\mathscr{A}}$ of Section 3.2. By providing suitable unfolding (and abstraction) operators, different instances of the specialization scheme can be defined.

The NPER$^{\mathscr{U}}_{\mathscr{A}}$ procedure is outlined in Algorithm 1. Roughly speaking, the procedure consists of two phases.

Phase 1) *Partial Evaluation.* It applies the EQNPE$^{\mathscr{U}}_{\mathscr{A}}$ 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 functions calls* that appear in the $(\vec{E},B)$-normalized version R' of the rewrite rules of R. We must normalize the rules in R before initializing Q because, for each t in Q, the FV-narrowing tree for t is not rooted by t but by $t{\downarrow_{\vec{E},B}}$; hence, we would lose the connection between the partially evaluated functions and the rules of R if the rules were not correspondingly normalized.
Given $\Sigma = (\mathscr{D} \uplus \Omega)$, a maximal function call in a term t is any outermost subterm of t that is rooted by a defined function symbol appearing in the equations of E. Given a rewrite rule $s \Rightarrow t$ of R, by $mcalls(s \Rightarrow t)$, we denote the set of all the maximal function calls that occur in s and t. By abuse, $mcalls(R)$ is the set of all maximal calls in the rewrite 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 univocally derived by executing GENTHEORY$(Q',(\Sigma,E \uplus B))$.

Phase 2) *Compression.* It consists of a refactoring transformation that computes a new, much more compact equational theory $\mathscr{E}'' = (\Sigma'',E'' \uplus B'')$ where the equations of E' are simplified by renaming similar expressions w.r.t. an independent renaming function ρ that is derived from the set of specialized calls Q' so that unused symbols, unneeded axioms, and unnecessary repetition of variables are removed.

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

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

1: **function** $\text{NPER}^{\mathscr{U}}_{\mathscr{A}}(\mathscr{R})$
2: $R' \leftarrow \{(l\downarrow_{\vec{E},B}) \Rightarrow (r\downarrow_{\vec{E},B}) \mid l \Rightarrow r \in R\}$
3: $Q \leftarrow mcalls(R')$
Phase 1. Partial Evaluation
4: $Q' \leftarrow \text{EQNPE}^{\mathscr{U}}_{\mathscr{A}}((\Sigma, E \uplus B), Q)$
5: $(\Sigma, E' \uplus B) \leftarrow \text{GENTHEORY}(Q', (\Sigma, E \uplus B))$
Phase 2. Compression
6: Let ρ be an independent renaming for Q in
7: $\Sigma'' \leftarrow (\Sigma \setminus \{f \mid f \text{ occurs in } E \setminus E'\}) \cup \{root(\rho(t)) \mid t \in Q\}$
8: $B'' = \{ax(f) \in B \mid f \in \Sigma \cap \Sigma''\}$
9: $E'' \leftarrow \bigcup_{t \in Q} \{\rho(t)\theta = RN_\rho(t') \mid t\theta = t' \in E'\}$
10: $R'' \leftarrow \{RN_\rho(l) \Rightarrow RN_\rho(r) \mid l \Rightarrow r \in R'\}$

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

11: **return** $\mathscr{R}' = (\Sigma'', E'' \uplus B'', R'')$

More precisely, for each t of sort s in Q'such that its root symbol is f, we define $\rho(t) = f_t(\overline{x_n : s_n})$, 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 Q' and is different from the root symbol of any other renamed term $\rho(t')$, for $t' \in Q'$.

Given the renaming ρ, the compression algorithm computes a new equation set E'' by replacing each call in E' by a call to the corresponding renamed function according to ρ. 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 must be renamed by recursively replacing each call in the given expression by a call to the corresponding renamed function (according to ρ). This is done by means of the function RN_ρ.

Furthermore, a new rewrite rule set R'' is also produced by consistently applying RN_ρ to the $(\vec{E}, B)$-normalized rewrite rules of R'. Specifically, each rewrite rule $l \Rightarrow r$ in R' is transformed into the rewrite rule $RN_\rho(l) \Rightarrow RN_\rho(r)$, in which every maximal function call t in the rewrite rule is recursively renamed according to the independent renaming ρ taking into account the term equivalences given by B.

Given the rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$ and its specialization $\mathscr{R}' = \text{NPER}^{\mathscr{U}}_{\mathscr{A}}(\mathscr{R})$, all of the executability conditions that are satisfied by $\mathscr{R}$ are also satisfied by $\mathscr{R}' = (\Sigma', E' \uplus B', R')$, including the fact that $\vec{\mathscr{E}}' = (\Sigma', B', \vec{E}')$ is a decomposition and that R' is (ground) coherent w.r.t. E' modulo B'. Also, because of the correctness and completeness of $\text{EQNPE}^{\mathscr{U}}_{\mathscr{A}}$, which states a strong correspondence between the variant computations of $\mathscr{E}$ and $\mathscr{E}'$ [7], the renaming function ρ that is used to generate $\mathscr{R}'$ is a bisimulation between the transition systems $(\mathscr{T}_{\Sigma/(E \uplus B)}, \to_{R/(E \uplus B)})$ and $(\mathscr{T}_{\Sigma'/(E' \uplus B')}, \to_{R'/(E' \uplus B')})$.

4 Total Evaluation and Constructor Variants

In [41], a theory transformation $\mathscr{R} \mapsto \mathscr{R}^{\Omega}_{l,r}$ is defined that relies on the division of Σ as $\mathscr{D} \uplus \Omega$ together with the notion of *most general constructor variant* that we describe in the following. Roughly speaking, $\mathscr{R}^{\Omega}_{l,r}$ is obtained from $\mathscr{R}$ by transforming each rewrite $l \Rightarrow r$ in $\mathscr{R}$ into a totally evaluated rule $l' \Rightarrow r'$, where l' and r' are constructor Ω-terms. More precisely, any call appearing in l (resp. r) to a function that is defined in $\mathscr{E}$ is replaced in l' (resp. r') by its constructor normal form w.r.t. $\mathscr{E}$.

4.1 Constructor Term Variants, Sufficient Completeness, and the CFVP

In order-sorted equational logic, the notion of constructor symbols and constructor terms are more intricate and essential than in standard term rewriting. Let us denote by $[t]_B$ the B-equivalence class of t, i.e., terms t' that are B-equivalent to t, in symbols $t' =_B t$ for all $t' \in [t]_B$. Given a decomposition $(\Sigma, B, \vec{E})$, quite often the signature Σ has a natural division as a disjoint union $\Sigma = \mathscr{D} \uplus \Omega$, where the elements of the *canonical algebra* $\mathscr{C}_{\vec{E},B} = \{[t\downarrow_{\vec{E},B}]_B \mid t \in \mathscr{T}_{\Sigma}\}$ (that is, the values computed by $\vec{E}, B$-simplification) are Ω-terms, whereas the function symbols $f \in \mathscr{D}$ are viewed as defined functions which are evaluated away by $\vec{E}, B$-simplification. The subsignature Ω (with same poset of sorts as Σ) is then called a *constructor subsignature* of Σ. We call a decomposition $(\Sigma, B, \vec{E})$ *sufficiently complete* with respect to the constructor subsignature Ω iff for each $t \in \mathscr{T}_{\Sigma}$ we have: (i) $t\downarrow_{\vec{E},B} \in \mathscr{T}_{\Omega}$; and (ii) if $u \in \mathscr{T}_{\Omega}$ and $u =_B v$, then $v \in \mathscr{T}_{\Omega}$. Condition (ii) ensures that if any element in a B-equivalent class is a Ω-term, all other elements in the class are also Ω-terms. We also say that $(\Sigma, B, \vec{E})$ is *sufficiently complete* w.r.t. Ω and input term $t \in \mathscr{T}_{\Sigma}$ if conditions (i) and (ii) hold for t. In the following, the `ctor` operator attribute of Maude is used to highlight the constructor symbols of an equational theory so that the constructor subsignature can be easily read off the Maude code.

A decomposition $\vec{\mathscr{E}} = (\Sigma, B, \vec{E})$ *protects* another decomposition $\vec{\mathscr{E}}_0 = (\Sigma_0, B_0, \vec{E}_0)$ iff $\mathscr{E}_0 \subseteq \mathscr{E}$, i.e., $\Sigma_0 \subseteq \Sigma$, $B_0 \subseteq B$, $E_0 \subseteq E$, and for all $t, t' \in \mathscr{T}_{\Sigma_0}(\mathscr{X})$ we have: (i) $t =_{B_0} t' \iff t =_B t'$, (ii) $t = t\downarrow_{\vec{E}_0,B_0} \iff t = t\downarrow_{\vec{E},B}$, and (iii) $\mathscr{C}_{\vec{E}_0,B_0} = \mathscr{C}_{\vec{E},B}|_{\Sigma_0}$, where $\mathscr{C}_{\vec{E},B}|_{\Sigma_0}$ agrees with $\mathscr{C}_{\vec{E},B}$ in the interpretation of all sorts and operations in Σ_0 and discards everything in $\Sigma \setminus \Sigma_0$. The decomposition $\vec{\mathscr{E}}_{\Omega} = (\Omega, B_{\Omega}, \vec{E_{\Omega}})$ is a *constructor* decomposition of $\vec{\mathscr{E}} = (\Sigma, B, \vec{E})$ iff 1) $\vec{\mathscr{E}}$ protects $\vec{\mathscr{E}}_{\Omega}$; and 2) $\vec{\mathscr{E}}$ is sufficiently complete w.r.t. its constructor subsignature Ω.

Throughout the paper, we assume that the set B of axioms respects the constructors in any decomposition $\vec{\mathscr{E}} = (\Sigma, B, \vec{E})$. In other words, if an axiom in B can be applied to a constructor term, then the result is a constructor term.

Example 4. Consider the following Maude functional module that encodes an equational theory $\mathscr{E} = (\Sigma, E \uplus B)$ for natural numbers modulo 2, with an equation that collapses natural numbers into the canonical forms `0` and `s(0)`.

```
fmod OS-NAT/2 is
  sorts Nat Zero One .
  subsort Zero One < Nat .
  op 0 : -> Zero [ctor] .
  op s : Zero -> One [ctor] .
  op s : Nat -> Nat .
  eq s(s(0)) = 0 [variant] .
endfm
```

Let us denote by $\vec{\mathscr{E}} = (\Sigma, B, \vec{E})$ the decomposition of the considered theory. The signature Σ can be naturally decomposed into $\mathscr{D} \uplus \Omega$, where

$$\mathscr{D} = \{\mathtt{s} : \mathtt{Nat} \to \mathtt{Nat}\} \text{ and } \Omega = \{\mathtt{0} :\to \mathtt{Zero}, \mathtt{s} : \mathtt{Zero} \to \mathtt{One}\}.$$

Then, the decomposition $(\Omega, \emptyset, \emptyset)$ is a constructor decomposition of $\vec{\mathscr{E}}$.

Given $\Sigma = \mathscr{D} \uplus \Omega$, it is possible to strengthen the notion of term variants to that of *constructor variants* [40].

Definition 1 (Constructor Variant [40]**).** *Let $\vec{\mathscr{E}} = (\Sigma, B, \vec{E})$ be a decomposition and let $(\Omega, B_\Omega, \vec{E}_\Omega)$ be a constructor decomposition of $\vec{\mathscr{E}}$. Given a term $t \in \mathscr{T}_\Sigma(\mathscr{X})$, we say that a variant (t', θ) of t is a* constructor variant *if $t' \in \mathscr{T}_\Omega(\mathscr{X})$ (i.e., the set of non-ground constructor terms).*

The following example illustrates the notion of constructor variant in Maude.

Example 5. Consider the functional module `OS-NAT/2` of Example 4 and the term `s(X)`, with `X:Nat`. There exist only two most general variants for `s(X)` in the equational theory $\mathscr{E}$ encoded by `OS-NAT/2`: namely, $(\mathtt{0}, \{\mathtt{X} \mapsto \mathtt{s(0)}\})$ and $(\mathtt{s(X)}, \mathtt{id})$. The former is also a constructor variant since the constructor `0` belongs to the constructor subsignature Ω of $\mathscr{E}$. Conversely, the latter is not a constructor variant since $\mathtt{s} : \mathtt{Nat} \to \mathtt{Nat}$ in `s(X)` is a defined symbol. Nonetheless, note that there exists the constructor variant $(\mathtt{s(Y)}, \mathtt{X} \mapsto \mathtt{Y} : \mathtt{Zero})$, where the sort of variable `X:Nat` is downgraded to sort `Zero`, which is a constructor variant that is less general than $(\mathtt{s(X)}, \mathtt{id})$.

The notion of most general variant can be trivially extended to constructor variants. By $[\![t]\!]^{\Omega}_{\vec{E},B}$, we denote the set of most general constructor variants for the term t. Given an equational theory $\mathscr{E} = (\Sigma, E \uplus B)$, we say that $\mathscr{E} = (\Sigma, E \uplus B)$ has the *constructor finite variant property* (CFVP) (or it is called a *finite constructor variant theory*) iff for all $t \in \mathscr{T}_\Sigma(\mathscr{X})$, $[\![t]\!]^{\Omega}_{\vec{E},B}$ is a finite set. By abuse, we often say that a decomposition $(\Sigma, B, \vec{E})$ has the FVP (resp. CFVP) when $\mathscr{E} = (\Sigma, E \uplus B)$ is a finite variant theory (resp. a constructor finite variant theory).

An algorithm for computing the complete set of most general constructor variants $[\![t]\!]^{\Omega}_{\vec{E},B}$ is provided in [40] for a decomposition $(\Sigma, B, \vec{E})$ that satisfies the FVP, has a constructor decomposition $(\Omega, E_\Omega, B_\Omega)$, and satisfies the extra *preregular-below* condition [40], which essentially ensures that Σ does not contain any overloaded symbol with a constructor typing that lies below a defined typing for the same symbol. Roughly speaking, the algorithm has two phases. First, the signature Σ of $(\Sigma, B, \vec{E})$ is refined into a new signature Σ^c that introduces a new sort $\sharp s$ for each sort s in the sort poset of Σ. Also, this sort refinement is extended to subsort relations, and constructor operators to precisely identify the constructor terms of the decomposition. Two functions, $(\_)^\bullet$ and its inverse $(\_)_\bullet$, are respectively used to map sorts of Σ to sorts of Σ^c and sorts of Σ^c to sorts of Σ. These functions are homomorphically extended to terms and substitutions in the usual way. Then, $[\![t]\!]^{\Omega}_{\vec{E},B}$ is distilled from the set of most general variants $[\![t]\!]_{\vec{E},B}$ by using unification modulo B in the following way:

$$[\![t]\!]^{\Omega}_{\vec{E},B} = \{(t'\tau^\bullet, (\sigma\tau^\bullet)_{|Var(t)}) \mid (t', \sigma) \in [\![t]\!]_{\vec{E},B}, \tau \in CSU_B(t' = x{:}(ls(t'))_\bullet)\}$$

where $CSU_B(t = t')$ denotes the complete set of unifiers of $t = t'$ modulo B.

Example 6. Consider the FVP functional module OS-NAT/2 of Example 4. Its associated decomposition has a constructor decomposition as shown in Example 4, and also meets the *preregular below* condition. Indeed, although the successor operator in OS-NAT/2 is overloaded, its constructor typing $\mathtt{s} : \mathtt{Zero} \to \mathtt{One}$ is below the defined typing $\mathtt{s} : \mathtt{Zero} \to \mathtt{One}$, since Zero < Nat and One < Nat. Hence, the algorithm in [40] can be used to compute $[\![t]\!]^{\Omega}_{\vec{E},B}$.

The complete set of most general constructor variants for s(X:Nat)

$$[\![\mathtt{s}(\mathtt{X})]\!]^{\Omega}_{\vec{E},B} = \{\{(\mathtt{0}, \mathtt{X} \mapsto \mathtt{s}(\mathtt{0})), (\mathtt{s}(\mathtt{X}'), \mathtt{X} \mapsto \mathtt{X}' : \mathtt{Zero})\}$$

is derived from $[\![\mathtt{s}(\mathtt{X})]\!]_{\vec{E},B} = \{\{(\mathtt{0}, \mathtt{X} \mapsto \mathtt{s}(\mathtt{0})), (\mathtt{s}(\mathtt{X}), \mathtt{id})\}$ as follows.

The constructor variant $(\mathtt{0}, \mathtt{X} \mapsto \mathtt{s}(\mathtt{0}))$ is a variant in $[\![\mathtt{s}(\mathtt{X})]\!]_{\vec{E},B}$ and the (trivial) unification problem $\mathtt{0} = \mathtt{Y} : \sharp\mathtt{Zero}$ provides a unifier $\tau^{\bullet}$ that leaves $(\mathtt{0}, \mathtt{X} \mapsto \mathtt{s}(\mathtt{0}))$ unchanged.

The constructor variant $(\mathtt{s}(\mathtt{X}'), \mathtt{X} \mapsto \mathtt{X}' : \mathtt{Zero})$ derives from the variant $v = (\mathtt{s}(\mathtt{X}), \mathtt{id}) \in [\![\mathtt{s}(\mathtt{X})]\!]_{\vec{E},B}$ by solving the unification problem $\mathtt{s}(\mathtt{X}) = \mathtt{Y} : \sharp\mathtt{Nat}$ which yields the computed unifier $\tau = \{\mathtt{X} \mapsto \mathtt{X}' : \sharp\mathtt{Zero}, \mathtt{Y} \mapsto \mathtt{s}(\mathtt{X}')\}$; hence, $\tau^{\bullet} = \{\mathtt{X} \mapsto \mathtt{X}' : \mathtt{Zero}, \mathtt{Y} \mapsto \mathtt{s}(\mathtt{X}')\}$, and finally by applying $\tau^{\bullet}$ to v, we get $(\mathtt{s}(\mathtt{X}'), \{\mathtt{X} \mapsto \mathtt{X}' : \mathtt{Zero}\})$.

For any decomposition $(\Sigma, B, \vec{E})$, note that FVP implies CFVP when there exists a constructor decomposition of $(\Sigma, B, \vec{E})$.

The following result establishes that the FVP and/or SC nature of an input equational theory is preserved by the $\mathrm{NPER}^{\mathscr{U}}_{\mathscr{A}}$ transformation.

Theorem 1 (FVP and SC preservation). *Let $\mathscr{R} = (\Sigma, E \uplus B, R)$ be a rewrite theory with $\mathscr{E} = (\Sigma, E \uplus B)$ such that $\vec{\mathscr{E}} = (\Sigma, B, \vec{E})$ is a decomposition. Let $\mathscr{R}' = \mathrm{NPER}^{\mathscr{U}}_{\mathscr{A}}(\mathscr{R})$ be a specialization of $\mathscr{R}$ under the renaming ρ such that $\mathscr{R}' = (\Sigma', E' \uplus B', R')$. Let $Q = \{t | s \mapsto t \in \rho\}$ so that $\mathscr{E}' = (\Sigma', E' \uplus B')$ is Q-closed modulo B'. Then, it holds that*

1. *If $\vec{\mathscr{E}}$ satisfies the FVP, then $\vec{\mathscr{E}}'$ satisfies the FVP;*
2. *Given $\Sigma = \mathscr{D} \uplus \Omega$, if $\vec{\mathscr{E}}$ satisfies SC w.r.t. Ω, then $\vec{\mathscr{E}}'$ satisfies SC w.r.t. Ω for every input term that is Q-closed modulo B'.*

4.2 Total Evaluation of Rewrite Theories

The total evaluation transformation $\mathscr{R} \mapsto \mathscr{R}^{\Omega}_{l,r}$ is achieved by computing the set of most general constructor variants $[\![\langle l, r\rangle]\!]^{\Omega}_{\vec{E},B}$, for each $l \Rightarrow r$ in $\mathscr{R}$. More specifically, $\mathscr{R}$ is transformed into $\mathscr{R}^{\Omega}_{l,r}$ by replacing the set of rewrite rules of $\mathscr{R}$ with

$$R^{\Omega}_{l,r} = \{l' \to r' \mid l \to r \in R \wedge (\langle l', r'\rangle, \sigma) \in [\![\langle l, r\rangle]\!]^{\Omega}_{\vec{E},B}\}$$

Correctness of the transformation $\mathscr{R} \mapsto \mathscr{R}^{\Omega}_{l,r}$ (or more precisely, the isomorphism between the ground canonical algebras of $\mathscr{R}$ and $\mathscr{R}^{\Omega}_{l,r}$) is established in [41] and is ensured when the equational theory $\mathscr{E} = (\Sigma, E \uplus B)$ in $\mathscr{R}$ satisfies the FVP, has a constructor decomposition $(\Omega, B_{\Omega}, E_{\Omega})$, and satisfies the *preregular below* condition.

Example 7. Consider a rewrite theory $\mathscr{R}$ that includes a single rewrite rule

```
rl [ Y:Nat ] => [ s(Y:Nat) ] .
```

and the finite variant equational theory $\mathscr{E} = (\Sigma \cup \{[\_] : \texttt{Nat} \to \texttt{State}\}, E \uplus B)$ that extends the equational theory of Example 4 with the constructor operator $[\_] : \texttt{Nat} \to \texttt{State}$. Note that the decomposition $\vec{\mathscr{E}} = (\Sigma \cup \{[\_] : \texttt{Nat} \to \texttt{State}\}, B, \vec{E})$ of $\mathscr{E}$ has a constructor decomposition $(\Omega, \emptyset, \emptyset)$ where $\Sigma = \Omega \uplus \mathscr{D}$ with $\Omega = \{[\_] : \texttt{Nat} \to \texttt{State}, \texttt{s} : \texttt{Zero} \to \texttt{One}, \texttt{0} :\to \texttt{Zero}\}$ and $\mathscr{D} = \{\texttt{s} : \texttt{Nat} \to \texttt{Nat}\}$, and clearly satisfies the *preregular below* condition for the very same argument exposed in Example 6. Hence, the transformation $\mathscr{R} \mapsto \mathscr{R}^{\Omega}_{l,r}$ can be applied to $\mathscr{R}$ thereby specializing the original rule into the two following rewrite rules

```
rl [ s(0) ] => [ 0 ] .
rl [ X:Zero ] => [ s(X:Zero) ] .
```

which are obtained from the computation of $[\![\langle [\texttt{Y} : \texttt{Nat}], [\texttt{s}(\texttt{Y} : \texttt{Nat})] \rangle]\!]^{\Omega}_{\vec{E},B}$.

In Section 5, we show how the $\mathscr{R} \mapsto \mathscr{R}^{\Omega}_{l,r}$ transformation can be mimicked as an instance of our NPER$^{\mathscr{U}}_{\mathscr{A}}$ scheme and we formulate two additional instances of the generic algorithm that can deal with a rewrite theory that does not satisfy the FVP and/or SC. Furthermore, sometimes we can transform a theory that satisfies SC but not FVP into a specialized theory that satisfies both SC and FVP so that the above transformation can be applied.

5 Instantiating the Specialization Scheme for Rewrite Theories

Given a rewrite theory $\mathscr{R} = (\Sigma, E \uplus B, R)$, with $\mathscr{E} = (\Sigma, B, \vec{E})$ being a decomposition of $(\Sigma, E \uplus B)$, the equational theory $\mathscr{E}$ in $\mathscr{R}$ may or may not meet sufficient completeness (SC) or the finite variant property (FVP). In this section, we particularize the specialization scheme of Section 3 by considering the following three[11] possible scenarios:

1. $\mathscr{E}$ meets SC and the FVP (hence, it has the CFVP);
2. $\mathscr{E}$ does not meet SC but it meets the FVP;
3. $\mathscr{E}$ does not meet the FVP.

Recall the parameterized NPER$^{\mathscr{U}}_{\mathscr{A}}$ algorithm of Section 3.3 relies on two generic operators: an unfolding operator $\mathscr{U}$ that defines the unfolding rule used to determine when and how to terminate the construction of the narrowing trees; and an abstraction operator $\mathscr{A}$ that is used to guarantee that the set of terms obtained during partial evaluation (i.e., the set of deployed narrowing trees) is kept finite and progressively covers (modulo B) all of the specialized calls. The instantiation of the scheme requires particularizing these two parameters in order to specify a terminating, correct and complete partial evaluation for $\mathscr{E}$. In the following, we provide three different implementations for the tandem $\mathscr{U}/\mathscr{A}$, and we show how they work in practice on some use cases that cover all three scenarios.

5.1 Unfolding Operators

Let us first provide three possible implementations of the unfolding operator $\mathscr{U}$ that are respectively able to deal with: (a) equational theories that satisfy the SC and FVP (hence, satisfy the CFVP); (b) any equational theory that satisfies the FVP; and (c) equational theories that do not

[11] The case when $\mathscr{E}$ satisfies SC but not the FVP is not considered because there is no technique to compute the finite set of most general constructor variants in this case, which is a matter for future research.

satisfy the FVP. Since $(\Sigma, B, \vec{E})$ is a decomposition of $(\Sigma, E \uplus B)$, all the considered implementations adopt the folding variant narrowing strategy to build the narrowing trees which are needed to specialize the input theory.

(a) Consider the case when $\mathscr{E} = (\Sigma, E \uplus B)$ satisfies all of the conditions required for the correctness of the transformation $\mathscr{R} \mapsto \mathscr{R}^{\Omega}_{l,r}$. In particular, $\mathscr{E}$ is SC and has the FVP. Let Σ^c be the sort-refinement of the signature Σ presented in Section 4.1, where $(\_)_\bullet$ (resp., $(\_)^\bullet$) is the function that maps the sorts of Σ into the sorts of Σ^c (resp., the sorts of Σ^c into the sorts of Σ). Then, we define the following unfolding operator that totally evaluates Q in the decomposition $\vec{\mathscr{E}}$

$$\mathscr{U}_{cfvp}(Q, \vec{\mathscr{E}}) = \bigcup_{t \in Q} \{(x : s)\sigma^\bullet \mid t \leadsto^*_{\sigma, \vec{E}, B} \; x : (s_\bullet) \wedge t\sigma \neq x\sigma \wedge s = ls(t)\}$$

(b) When the equational theory $\mathscr{E}$ does not satisfy SC but does satisfy the FVP, FV-narrowing trees are always finite objects that can be effectively constructed in finite time. Therefore, in this specific case, we define the following unfolding operator that constructs the complete FV-narrowing tree for any possible call.

$$\mathscr{U}_{fvp}(Q, \vec{\mathscr{E}}) = \bigcup_{t \in Q} \{t' \mid t \leadsto^!_{\sigma, \vec{E}, B} t' \in VN^{\circlearrowleft}_{\vec{\mathscr{E}}}(t)\}$$

where $t \leadsto^!_{\sigma, \vec{E}, B} t'$ denotes a FV-narrowing derivation from t to the term t' to which no FV-narrowing steps can be applied.

(c) Finally, when $\mathscr{E}$ does not meet the finite variant property, $\mathscr{U}_{fvp}(Q, \vec{\mathscr{E}})$ cannot be applied since the FVN strategy may lead to the creation of an infinite narrowing tree for some specialized calls in Q. In this case, the unfolding rule must implement a form of local control that stops the expansion of infinite derivations in the FV-narrowing tree. A solution to this problem has already been provided in [7] by means of an unfolding operator that computes a finite (possibly partial) FV-narrowing tree fragment for every specialized call t in Q. Narrowing derivations in the tree are stopped when no further FV-narrowing step can be performed or potential non-termination is detected by applying a subsumption check at each FV-narrowing step. The subsumption check is based on an *equational order-sorted* extension of the classical homeomorphic embedding relation [8] that is commonly used to ensure termination of symbolic methods and program optimization techniques.
Roughly speaking, a homeomorphic embedding relation is a structural preorder under which a term t is greater than (i.e., it embeds) another term t', written as $t \rhd t'$, 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 relations have become very popular to ensure termination of *symbolic* transformations because, provided the signature is finite, for every infinite sequence of terms $t_1, t_2, \ldots$, there exist $i < j$ such that $t_i \trianglelefteq t_j$. In other words, the embedding relation is a well-quasi order (wqo) [34]. Therefore, when iteratively computing a sequence $t_1, t_2, \ldots, t_n$, finiteness of the sequence can be guaranteed by using the embedding as a whistle: 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 $\trianglelefteq$ 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.
By $\mathscr{U}_{\overline{fvp}}(Q, \vec{\mathscr{E}})$, we denote this unfolding operator whose full formalization is given in [7].

5.2 Abstraction Operators

We consider two implementations of the abstraction operator: the first one deals with equational theories that are sufficiently complete and satisfy the finite variant property, while the second one covers the other two possible scenarios that we highlighted at the beginning of Section 5.

(a) When the equational theory $\mathscr{E}$ satisfies SC and has the FVP so that the unfolding operator $\mathscr{U}_{cfvp}(Q,\vec{\mathscr{E}})$ is applied, there is no need for an abstraction process. By construction of the $\mathscr{U}_{cfvp}(Q,\vec{\mathscr{E}})$ operator, the leaves of the tree are constructor terms; hence, they do not include any uncovered function call that needs to be abstracted by a further iteration of the partial evaluation process as constructor terms are trivially B-closed w.r.t. Q. Therefore, in this case, we can simply define $\mathscr{A}_{cfvp}(Q,\mathscr{L},B) = Q$, thus returning the very same set of specialized calls Q.

(b) As for the remaining cases, there is no guarantee that the leaves of the narrowing trees are B-closed w.r.t. the specialized calls in Q. Indeed, when the equational theory $\mathscr{E}$ does not satisfy either sufficient completeness or the finite variant property, the operators $\mathscr{U}_{fvp}(Q,\vec{\mathscr{E}})$ and $\mathscr{U}_{\overline{fvp}}(Q,\vec{\mathscr{E}})$ might deliver uncovered function calls to be abstracted. To overcome this problem, we simply resort to the abstraction procedure of [7], which relies on an *equational order sorted extension* of the pure, syntactical least general generalization algorithm [10] so that not too much precision is lost despite the abstraction.

Roughly speaking, the syntactic generalization problem for two or more expressions, in a pure syntactic and untyped setting, means finding their *least general generalization*, i.e., the least general expression t such that all of the given expressions are instances of t under appropriate substitutions. For instance, the expression *sibling(X,Y)* is a generalizer of both *sibling(john,sam)* and *sibling(tom,sam)*, but their least general generalizer is *sibling(X,sam)*.

In [10], the notion of least general generalization is extended to the order-sorted modulo axioms setting, where function symbols can obey any combination of associativity, commutativity, and identity axioms (including the empty set of such axioms). For instance, the least general generalizer of *sibling(sam,john)* and *sibling(tom,sam)* is still *sibling(X,sam)*, when *sibling* is a commutative symbol. In general, there is no unique lgg in the framework of [10], due to both the order-sortedness and to the equational axioms. Nonetheless, for the case of modular combinations of associativity and commutativity axioms, there is always a finite, minimal, and complete set of equational lggs (E-lggs) so that any other generalizer has at least one of them as a B-instance.

Therefore, in the case when the equational theory $\mathscr{E}$ does not satisfy either sufficient completeness or the finite variant property, we consider the abstraction operator $\mathscr{A}_{Elgg}(Q,\mathscr{L},B)$, which returns a set Q' of specialized calls that abstracts the set $Q \cup \mathscr{L}$ by using the generalization process formalized in [7] that ensures that Q' is B-closed w.r.t. $Q \cup \mathscr{L}$.

The use of folding variant narrowing in the definition of the three unfolding operators $\mathscr{U}_{\overline{fvp}}$, $\mathscr{U}_{fvp}$, and $\mathscr{U}_{cfvp}$, together with the abstraction operators $\mathscr{A}_{cfvp}$ and $\mathscr{A}_{Elgg}$, provides good overall behavior regarding both the elimination of intermediate data structures and the propagation of information.

6 Specializing the Bank Account System

In this section, we describe the precise specialization process that obtains the specialized bank account system of Example 1. For convenience, we denote by $\mathscr{R}_b$ the rewrite theory that specifies the bank account system. $\mathscr{R}_b$ includes the three rewrite rules of Figure 1 and the equational theory $\mathscr{E}_b$ of Figure 2. The theory $\mathscr{E}_b$ also contains algebraic axioms associated with two operators: 1) the associative and commutative, constructor operator `_+_ : Nat Nat -> Nat` with identity `0` (used to model natural numbers); and 2) the associative and commutative, constructor operator `_,_: MsgConf MsgConf -> MsgConf` with identity `mt` (used to model multisets of deposit and withdrawal messages). The whole Maude specification of the bank account system is given in Appendix A.

As shown in Example 1, $\mathscr{E}_b$ is not a finite variant theory; therefore, the specialization of $\mathscr{R}_b$ can only be performed by using the unfolding operator $\mathscr{U}_{\overline{fvp}}$ despite the fact that it is sufficiently complete. Indeed, the other two operators (namely, $\mathscr{U}_{fvp}$ and $\mathscr{U}_{cfvp}$) are only applicable to equational theories that satisfy the finite variant property. In other words, the specialization of $\mathscr{R}_b$ is achieved by using the NPER$^{\mathscr{U}}_{\mathscr{A}}$ scheme instance with $\mathscr{U}=\mathscr{U}_{\overline{fvp}}$ and $\mathscr{A} = \mathscr{A}_{Elgg}$.

The specialization algorithm starts Phase 1 by normalizing the rewrite rules of $\mathscr{R}_b$ (Line 2 of Algorithm 1) w.r.t. $\mathscr{E}_b$. In this specific case, normalization only affects the `dep` rewrite rule, while `w-req` and `w` rules are left unchanged. The normalized version of `dep` is

```
rl [dep-n] : < bal: n pend: x overdraft: false threshold: n + m + h funds: f > # msgs,d(m)
          => < bal: n + m pend: x overdraft: false threshold: n + m + h funds: f > # msgs .
```

Rule normalization allows a first optimization to be achieved since the `dep` rule is simplified into `dep-n` by removing the operator « `bal:_ pend:_ overdraft:_ threshold:_ funds:_` » from the right-hand side of the `dep` rule. At this point, all the maximal function calls are extracted from the normalized rules and stored in the set Q (Line 3 of Algorithm 1). Note that only the right-hand side of the `w` rule contains calls to the underlying equational theory $\mathscr{E}_b$. More precisely, the set Q of maximal function calls is $Q = \{rhs(\mathtt{w})\}$, where $rhs(\mathtt{w})$ is the right-hand side of `w`. The algorithm proceeds by partially evaluating $\mathscr{E}_b$ w.r.t.Q by an instance of the EQNPE$^{\mathscr{U}}_{\mathscr{A}}$ scheme with $\mathscr{U}_{\overline{fvp}}$ (in tandem with $\mathscr{A}_{Elgg}$) (Line 4 of Algorithm 1) and Phase 1 terminates by generating a rather complex and textually large specialized equational theory $\mathscr{E}'_b$ that contains 17 equations as shown in Appendix B. In Phase 2, the algorithm compresses the computed equational theory $\mathscr{E}'_b$ into a more compact theory $\mathscr{E}''_b$ that just contains four newly introduced functions (namely, `f1`, `f2`, `f3`, `f4`) that rename common nested calls and remove unused symbols. Furthermore, it propagates the computed renaming to the rewrite rules to let them access the new functions of $\mathscr{E}''_b$. The resulting specialization $\mathscr{R}'_b$ for $\mathscr{R}_b$ is shown in Appendix C.

It is worth noting that the equational theory $\mathscr{E}''_b$ in $\mathscr{R}'_b$ has the finite variant property. This can be automatically proven by using the FVP checker in [9] on $\mathscr{E}''_b$, or by simply observing that the functions `f1`,`f2`, `f3`, `f4` are all defined by non-recursive equations and do not contain nested function calls in their left-hand sides, which suffices to ensure the FVP for $\mathscr{E}''_b$ [11]. Furthermore, the constructor decomposition of $\mathscr{E}''_b$ has a signature which is trivially preregular below the signature of $\mathscr{E}''_b$, since there are no overloaded operators with both a constructor typing and a defined typing. Additionally, by Theorem 1 $\mathscr{E}''_b$ is sufficiently complete.

Therefore, $\mathscr{R}'_b$ can be further specialized by applying the NPER$^{\mathscr{U}}_{\mathscr{A}}$ scheme instantiated with $\mathscr{U}=\mathscr{U}_{cfvp}$ and $\mathscr{A} = \mathscr{A}_{cfvp}$. The final outcome is the optimized and extremely compact specialization $\mathscr{R}''_b$ shown in Example 1 (Figure 3) that only includes three equations modeling the new invented function `f0`.

As a final remark, $\mathscr{R}''_b$ can be further optimized by a simple post-processing unfolding transformation that achieves the very same total evaluation of [41]. It suffices to encode each rewrite rule $l \Rightarrow r$ in $\mathscr{R}''_b$ with a term $l \,|\, r$ (where $\_\,|\,\_$ is a fresh operator not appearing in the equational theory) and solve the reachability goal $l \,|\, r \rightsquigarrow_\sigma (x : ls(l)_\bullet \,|\, y : ls(r)_\bullet)$. The instantiated leaves $l' \,|\, r'$ are *constructor* terms $(x : s)\sigma^\bullet \,|\, (y : s)\sigma^\bullet$ that correspond to the totally evaluated rules $l' \Rightarrow r'$.

For instance, the `w-s` rewrite rule in $\mathscr{R}''_b$ can be totally evaluated by solving the reachability goal with initial state

```
< bal: n pend: x overdraft: false threshold: n + h funds: f > # msgs,w(m) | f0(m,n,x,h,msgs)
```

that yields the specialized and totally evaluated withdrawal rules:

```
rl [w-s-1] : < bal: n + m + x pend: m overdraft: false threshold: n + m + x + h funds: f >
               # msgs, w(m + x)
                 => < bal: n pend: 0 overdraft: false threshold: n + m + x + h  funds: f > # msgs .

rl [w-s-2] : < bal: n + m pend: m + x overdraft: false threshold: n + m + h funds: f > # msgs, w(m)
                 => < bal: n pend: x overdraft: false threshold: n + m + h funds: f > # msgs .

rl [w-s-3] : < bal: n pend: y overdraft: false threshold: n + h  funds: f > # msgs, w(1 + n + x)
                 => < bal: n pend: y overdraft: true threshold: n + h  funds: f > # msgs .
```

The transformation leaves `w-req-s` and `dep-s` unchanged because these rules do not contain any function call to be unfolded.

Our specialization framework has been implemented in the Presto system [46], which provides all the functionality previously described in this paper. Table 1 contains some experiments that we have performed with an extension of the rewrite theory of Example 1 that is given by the Maude module `Fully-Managed-Account`, where deposits are fully automated by increasing balance accounts with a huge amount in a single step. Therefore there is no need to explicitly provide deposit messages in the input terms. By doing so, we avoid to feed Presto with huge input terms (with millions of deposits) whose parsing time might heavily affect the overall performance of the specialization process, thereby providing a more precise and fair experimental analysis.

Specifically, four distinct specializations of the rewrite theory under examination have been computed. Since the original specification `Fully-Managed-Account` does not satisfy the FVP, we first computed the specialized rewrite theory `FMA-Specialized` by using the tandem $\mathscr{U}_{\overline{fvp}}/\mathscr{A}_{Elgg}$. The obtained specialization does satisfy all of the conditions that are required to be further specialized by using either the tandem $\mathscr{U}_{fvp}/\mathscr{A}_{Elgg}$ or $\mathscr{U}_{cfvp}/\mathscr{A}_{cfvp}$ (in particular, it satisfies SC and has the FVP); hence, we have also independently computed the two corresponding (re-)specializations, `FMA-Specialized-FVP` and `FMA-Specialized-CFVP`. Also, we derived the total evaluation `FMA-Specialized-TE` from `FMA-Specialized-CFVP`.

For each experiment, we recorded the execution time $T_{\mathscr{R}'}$ of each specialization for five rewrite sequences with an increasing number of rewrite rule applications (from 100 thousands to 10 millions of applications). The considered sequences originate from the very same input term, hence input processing impacts on each experiment in the same way. Then, we compared $T_{\mathscr{R}'}$ with the execution time $T_{\mathscr{R}}$ in the original specification. These parameters allow us to precisely measure the degree of equational optimization achieved by Presto for a given rewrite theory. Indeed, the relative speedups for each specialization are computed as the ratio $T_{\mathscr{R}}/T_{\mathscr{R}'}$. We also measured the size of each specialization as the number of its rewrite rules and equations.

Our figures show an impressive performance improvement in all of the considered experiments, with an average speedup of 20.52. In the worst case, we get a totally evaluated rewrite the-

	Fully-Managed-Account		FMA-Specialized			FMA-Specialized-FVP			FMA-Specialized-CFVP			FMA-Specialized-TE		
Size	*Rls/Eqs*	*T*(ms)	*Rls/Eqs*	*T*(ms)	*Speedup*	*Rls/Eqs*	*T*(ms)	*Speedup*	*Rls/Eqs*	*T*(ms)	*Speedup*	*Rls/Eqs*	*T*(ms)	*Speedup*
100K		1,398		65	21.51		63	22.19		63	22.19		96	14.56
500K		7,175		337	21.29		308	23.30		308	23.30		483	14.86
1M	3/14	14,472	3/17	680	21.28	3/3	602	24.04	3/3	599	24.16	5/0	998	14.50
5M		72,096		3,469	20.78		3,068	23.50		3,053	23.61		5,049	14.28
10M		141,919		6,805	20.86		6,149	23.08		6,127	23.16		10,162	13.97

Table 1. Benchmarks for the fully managed bank account system.

ory FMA-Specialized-TE that runs 13.97 times faster than the original system, while the highest speedup (24.16) is achieved by the (doubly specialized) theory FMA-Specialized-CFVP. Interestingly, the totally evaluated specification FMA-Specialized-TE is the most compact one (5 rules and 0 equations), nonetheless it provides the smallest, yet significant ($\sim$14%), optimization. This happens because all the equations have been removed from FMA-Specialized-TE so that all of the equational computations are now *embedded* into system computations that are performed by applying rewrite rules, which is notoriously less efficient in Maude than the deterministic rewriting with equations. We also note that, since FMA-Specialized satisfies both SC and the FVP, for this particular benchmark the rewrite theories that are obtained by (re-)specializing FMA-Specialized using $\mathscr{U}_{fvp}$ and $\mathscr{U}_{cfvp}$ essentially achieve the same optimization.

Full details of these benchmarks together with further experiments are available at http://safe-tools.dsic.upv.es/presto.

7 Related work and Conclusion

In the related literature, there are very few semantic-preserving transformations for rewrite theories. Since Maude is a reflective language, many tools are built in Maude that rely on theory transformations that preserve specific properties such as invariants or termination behavior. Full-Maude [21], Real-Time Maude [45], MTT [26], and Maude-NPA [28] are prominent examples. Equational abstraction [42, 17] 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 [50] between rules, equations and axioms is necessary for executability purposes and also relies on rewrite theory transformations [41]. Also the semantic $\mathbb{K}$-framework [47] and the model transformations of [48] are based on sophisticated program transformations that both preserve the reduction semantics of the original theory. Nonetheless they do not aim to program optimization.

It is worth noting that our first transformation (for sufficiently complete, finite variant theories) must not be seen as a simple recast, in terms of partial evaluation, of the theory transformation of [41] since it has at least two extra advantages: 1) it seamlessly integrates the transformation of [41] within a unified, automated specialization framework for rewrite theory optimization; and 2) we have shown how we can automatically transform an equational theory that does not satisfy the FVP into a CFVP theory that can then be totally evaluated, while the original theory could not.

Our specialization technique can have a tremendous impact on the symbolic analysis of concurrent systems that are modeled as rewrite theories in Maude. The main reason why our technique is so effective in this area is that it not only achieves huge speedup for relevant 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 $\mathscr{E}$. By doing this, any $\mathscr{E}$-unification problem can be finitely solved and symbolic, narrowing-based analysis with R modulo $\mathscr{E}$ 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 no structural axioms so that it can be run in an independent rewriting infrastructure that does not support rewriting or narrowing modulo axioms. This allows some costly analyses that may require significant (or even unaffordable) resources, both in time and space, to be effectively performed.

Finally, 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 [28], Tamarin [37], AKiSs [18], Maude debuggers and program analysers [5, 4, 6, 3], termination provers, model checkers, variant-based satisfiability checkers, coherence and confluence provers, and different applications of symbolic reachability analysis [24].

Bibliography

[1] E. Albert, M. Alpuente, M. Falaschi, and G. Vidal. Indy User's Manual. Technical Report DSIC-II/12/98, Department of Computer Systems and Computation, Universitat Politècnica de València, 1998.

[2] E. Albert, M. Alpuente, M. Hanus, and G. Vidal. A Partial Evaluation Framework for Curry Programs. In *Proceedings of the 6th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 1999)*, volume 1705 of *Lecture Notes in Computer Science*, pages 376–395. Springer, 1999.

[3] M. Alpuente, D. Ballis, F. Frechina, and D. Romero. 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, 2012.

[4] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Assertion-based Analysis via Slicing with ABETS (system description). *Theory and Practice of Logic Programming*, 16(5–6):515–532, 2016.

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

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

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

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

[9] M. Alpuente, A. Cuenca-Ortega, S. Escobar, and J. Sapiña. Inspecting Maude Variants with GLINTS. *Theory and Practice of Logic Programming*, 17(5–6):689–707, 2017.

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

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

[12] M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. 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, 1997.

[13] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Safe Folding/Unfolding with Conditional Narrowing. In *Proceedings of the 6th International Joint Conference on Algebraic and Logic Programming (ALP 1997)*, volume 1298 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1997.

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

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

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

[17] K. Bae, S. Escobar, and J. Meseguer. 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, 2013.

[18] D. Baelde, S. Delaune, I. Gazeau, and S. Kremer. Symbolic Verification of Privacy-Type Properties for Security Protocols with XOR. In *Proceedings of the 30th International Symposium on Computer Security Foundations (CSF 2017)*, pages 234–248. IEEE Computer Society Press, 2017.

[19] C. Bouchard, K. A. Gero, C. Lynch, and P. Narendran. 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, 2013.

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

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

[22] H. Comon-Lundh and S. Delaune. The Finite Variant Property: How to Get Rid of Some Algebraic Properties. In *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA 2005)*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005.

[23] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany*, volume 1110 of *Lecture Notes in Computer Science*. Springer, 1996.

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

[25] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, and C. Talcott. Associative Unification and Symbolic Reasoning Modulo Associativity in Maude. In *Proceedings of the 12th International Workshop on Rewriting Logic and its Applications (WRLA 2018)*, volume 11152 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2018.

[26] F. Durán, S. Lucas, and J. Meseguer. 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, 2008.

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

[28] S. Escobar, C. Meadows, and J. Meseguer. 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, 2009.

[29] S. Escobar and J. Meseguer. 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, 2007.

[30] S. Escobar, J. Meseguer, and R. Sasse. Variant Narrowing and Equational Unification. *Electronic Notes in Theoretical Computer Science*, 238(3):103–119, 2009.

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

[32] I. Gnaedig and H. Kirchner. Computing Constructor Forms with Non Terminating Rewrite Programs. In *Proceedings of the 8th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2006)*, pages 121–132. Association for Computing Machinery, 2006.

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

[34] M. Leuschel. Improving Homeomorphic Embedding for Online Termination. In *Proceedings of the 8th International Workshop on Logic Programming Synthesis and Transformation (LOPSTR 1998)*, volume 1559 of *Lecture Notes in Computer Science*, pages 199–218. Springer, 1998.

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

[36] B. Martens and J. Gallagher. 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, 1995.

[37] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. 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, 2013.

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

[39] J. Meseguer. 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, 2015.

[40] J. Meseguer. Variant-based Satisfiability in Initial Algebras. *Science of Computer Programming*, 154:3–41, 2018.

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

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

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

[44] A. Middeldorp and E. Hamoen. Counterexamples to Completeness Results for Basic Narrowing. In *Proceedings of the 3rd International Conference on Algebraic and Logic Programming (ALP 1992)*, volume 632 of *Lecture Notes in Computer Science*, pages 244–258. Springer, 1992.

[45] P. C. Ölveczky and J. Meseguer. 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, 2008.

[46] The Presto Website, 2020. Available at: http://safe-tools.dsic.upv.es/presto.

[47] G. Roşu. $\mathbb{K}$: 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, 2017.

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

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

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

A Full specification of the Bank Account System

```
fmod NAT-PRES-MONUS is
  pr TRUTH-VALUE .
  sorts Nat NzNat Zero .
  subsort Zero NzNat < Nat .
  op 0 : -> Zero [ctor] .
  op 1 : -> NzNat [ctor] .
  op _+_ : NzNat Nat -> NzNat [ctor assoc comm id: 0] .
  op _+_ : Nat Nat -> Nat [ctor assoc comm id: 0] .
  vars n m : Nat .
  vars b b' : Bool .
  op _>_ : Nat Nat -> Bool .
  eq m + n + 1 > n = true [variant] .
  eq n > n + m = false [variant] .
  op _>=_ : Nat Nat -> Bool .
  eq m + n >= n =  true [variant] .
  eq n >= m + n + 1 = false [variant] .
  op _-_ : Nat Nat -> Nat .
  eq n - (n + m) = 0 [variant] .
  eq (n + m) - n = m [variant] .
endfm

mod MANAGED-ACCOUNT is
  pr NAT-PRES-MONUS .
  sorts Account Msg MsgConf State .
  subsort Msg < MsgConf .
  op < bal:_pend:_overdraft:_threshold:_funds:_ > : Nat Nat Bool Nat Nat -> Account [ctor] .
  op << bal:_pend:_overdraft:_threshold:_funds:_ >> : Nat Nat Bool Nat Nat -> Account .
  op mt : -> MsgConf [ctor] .
  op w : Nat -> Msg [ctor] .
  op d : Nat -> Msg [ctor] .
  op _,_ : MsgConf MsgConf -> MsgConf [ctor assoc comm id: mt] .
  op _#_ : Account MsgConf -> State [ctor] .
  op [_,_,_] : Bool State State -> State .
  vars n m x h : Nat .
  var b : Bool .
  vars s s' : State .
  var msgs : MsgConf .
  eq [true,s,s'] = s [variant] .
  eq [false,s,s'] = s' [variant] .
  eq << bal: (n + h) pend: m overdraft: b:Bool threshold: h funds: f >>
   = << bal: n pend: m overdraft: b:Bool threshold: h funds: f + 1 >> [variant] .
  eq << bal: n pend: m overdraft: b:Bool threshold: n + h funds: f >>
   =  < bal: n pend: m overdraft: b:Bool threshold: n + h funds: f > [variant] .

  rl [w-req] : < bal: n + m + x pend: x overdraft: false threshold: n + h funds: f > # msgs
            => < bal: n + m + x pend: x + m overdraft: false threshold: n + h funds: f >
               # w(m),msgs .

  rl [w] : < bal: n pend: x overdraft: false threshold: n + h funds: f > # w(m),msgs
        => [ m > n,
             < bal: n pend: x overdraft: true threshold: n + h funds: f > # msgs,
             < bal: (n - m) pend: (x - m) overdraft: false threshold: n + h funds: f > # msgs ]  .

  rl [dep] : < bal: n pend: x overdraft: false threshold: n + m + h funds: f >
               # d(m),msgs
          => << bal: (n + m) pend: x overdraft: false threshold: n + m + h funds: f >> # msgs .
endm
```

B Specialization of the Bank Account System $\mathscr{R}_b$

```
eq [$5 > $1,
    < bal: $1 pend: $5 + $6 overdraft:true limit: $1 + $2 funds: $3 > # $4,
    < bal: $1 - $5 pend: ($5 + $6) - $5 overdraft: false limit: $1 + $2 funds: $3 > # $4]
 = [$5 > $1,
    < bal: $1 pend: $5 + $6 overdraft: true limit: $1 + $2 funds: $3 > # $4,
    < bal: $1 - $5 pend: $6 overdraft: false limit: $1 + $2 funds: $3 > # $4] [ variant ] .

eq [$5 > $5,
    <bal: $5 pend: $1 + $5 overdraft:true limit: $5 + $2 funds: $3 > # $4,
    < bal: 0 pend: $1 overdraft: false limit: $5 + $2 funds: $3 > # $4]
 = < bal: 0 pend: $1 overdraft: false limit: $5 + $2 funds: $3 > # $4 [ variant ] .

eq [$5 > $1 + $5,
    < bal: $1 + $5 pend: $5 + $6 overdraft:true limit: $1 + $5 + $2 funds: $3 > # $4,
    < bal: ($1 + $5) - $5 pend: ($5 + $6) - $5 overdraft: false limit: $1 + $5 + $2
      funds: $3 > # $4]
 = < bal: $1 pend: $6 overdraft: false limit: $1 + $5 + $2 funds: $3 > # $4 [ variant ] .

eq [$5 > $5 + $6,
    < bal: $5 + $6 pend: $1 + $5 overdraft: true limit: $5 + $6 + $2 funds: $3 > # $4,
    < bal: ($5 + $6) - $5 pend: $1 overdraft: false limit: $5 + $6 + $2 funds: $3 > # $4]
 = < bal: $6 pend: $1 overdraft: false limit: $5 + $6 + $2 funds: $3 > # $4 [ variant ] .

eq [($4 + $5) > $4,
   < bal: $4 pend: $4 + $5 + $6 overdraft: true limit: $4 + $1 funds: $2 > # $3,
   < bal: $4 - $4 + $5 pend: ($4 + $5 + $6) - $4 + $5 overdraft: false limit: $4 + $1
     funds: $2 > # $3]
 = [($4 + $5) > $4,
   < bal: $4 pend: $4 + $5 + $6 overdraft: true limit: $4 + $1 funds: $2 > # $3,
   < bal: 0 pend: $6 overdraft: false limit: $4 + $1 funds: $2 > # $3] [ variant ] .

eq [($4 + $5) > $4 + $5,
    < bal: $4 + $5 pend: $4 overdraft: true limit: $4 + $5 + $1 funds: $2 > # $3,
    < bal: 0 pend: 0 overdraft: false limit: $4 + $5 + $1 funds: $2 > # $3]
 = < bal: 0 pend: 0 overdraft: false limit: $4 + $5 + $1 funds: $2 > # $3 [ variant ] .

eq [($4 + $5) > $4 + $5 + $6,
    < bal: $4 + $5 + $6 pend: $4 overdraft: true limit: $4 + $5 + $6 + $1 funds: $2 > # $3,
    < bal: ($4 + $5 + $6) - $4 + $5 pend: 0 overdraft: false limit: $4 + $5 + $6 + $1
      funds:$2 > # $3]
 = < bal: $6 pend: 0 overdraft: false limit: $4 + $5 + $6 + $1 funds: $2 > # $3 [ variant ] .

eq [($5 + $6) > $1,
    < bal: $1 pend: $5 overdraft: true limit: $1 + $2 funds: $3 > # $4,
       < bal: $1 - $5 + $6 pend: $5 - $5 + $6 overdraft: false limit: $1 + $2 funds: $3 > # $4]
 = [($5 + $6) > $1,
    < bal: $1 pend: $5 overdraft: true limit: $1 + $2 funds: $3 > # $4,
    < bal: $1 - $5 + $6 pend: 0 overdraft: false limit: $1 + $2 funds: $3 > # $4] [ variant ] .

eq [($5 + $6) > $1 + $5 + $6,
    < bal: $1 + $5 + $6 pend: $5 overdraft: true limit: $1 + $5 + $6 + $2 funds: $3 > # $4,
    < bal: ($1 + $5 + $6) - $5 + $6 pend: $5 - $5 + $6 overdraft: false limit: $1 + $5 + $6 + $2
      funds: $3 > # $4]
 = < bal: $1 pend: 0 overdraft: false limit: $1 + $5 + $6 + $2 funds: $3 > # $4 [ variant ] .

rl [dep] : < bal: n pend: x overdraft: false threshold: n + m + h funds: f > # msgs,d(m)
   => < bal: n + m pend: x overdraft: false threshold: n + m + h funds: f > # msgs.
rl [w] : < bal: n pend: x overdraft: false threshold: n + h funds: f > # msgs,withdraw(m)
   => [m > n,< bal: n pend: x overdraft: true threshold: n + h funds: f >
      # msgs, < bal: n - m pend: x - m overdraft: false threshold: n + h funds: f >
      # msgs] .
rl [w-req] : < bal: n + m + x pend: x overdraft: false threshold: n + m + x + h funds: f > # msgs
   => < bal: n + m + x pend: m + x overdraft: false threshold: n + m + x + h funds: f > # msgs,w(m) .
```

C Specialization of the Bank Account System $\mathscr{R}_b$ with compression

```
eq f0($5, $5 + $6, $1, $2, $3, $4)
   = < bal: $6 pend: $1 overdraft: false threshold: $5 + $6 + $2 funds: $3 > # $4 [ variant ] .

eq f0(1 + $1 + $6, $1, $2, $3, $4, $5)
   = < bal: $1 pend: 1 + $1 + $2 + $6 overdraft: true threshold: $1 + $3 funds: $4 > # $5 [ variant ] .

eq f1($5, $1, $5 + $6, $2, $3, $4) = f0($5, $1, $6, $2, $3, $4) [ variant ] .

eq f1($5, $1 + $5, $5 + $6, $2, $3, $4)
   = < bal: $1 pend: $6 overdraft: false threshold: $1 + $5 + $2 funds: $3 > # $4 [ variant ] .

eq f1($4 + $5, $4, $4 + $5 + $6, $1, $2, $3) = f2($4, $5, $6, $1, $2, $3) [ variant ] .

eq f1($5 + $6, $1, $5, $2, $3, $4) = f3($5, $6, $1, $2, $3, $4) [ variant ] .

eq f1($5 + $6, $1 + $5 + $6, $5, $2, $3, $4)
   = < bal: $1 pend: 0 overdraft: false threshold: $1 + $5 + $6 + $2 funds: $3 > # $4 [ variant ] .

eq f1(1 + $1 + $6, $1, $2, $3, $4, $5)
   = < bal: $1 pend: $2 overdraft: true threshold: $1 + $3 funds: $4 > # $5 [ variant ] .

eq f1($4 + $5 + $6 + $7, $4 + $5, $4 + $6, $1, $2, $3)
   = f4($4, $5, $6, $7, $1, $2, $3) [ variant ] .

eq f2($1, 1 + $6, $2, $3, $4, $5)
   = < bal: $1 pend: 1 + $1 + $2 + $6 overdraft: true threshold: $1 + $3 funds: $4 > # $5 [ variant ] .

eq f2($5, 0, $1, $2, $3, $4)
   = < bal: 0 pend: $1 overdraft: false threshold: $5 + $2 funds: $3 > # $4 [ variant ] .

eq f3($4, $5, $4 + $5 + $6, $1, $2, $3)
   = < bal: $6 pend: 0 overdraft: false threshold: $4 + $5 + $6 + $1 funds: $2 > # $3 [ variant ] .

eq f3($4 + $6, 1 + $5 + $7, $4 + $5, $1, $2, $3)
   = < bal: $4 + $5 pend: $4 + $6 overdraft: true threshold: $4 + $5 + $1 funds: $2 > # $3 [ variant ] .

eq f3(1 + $4 + $6, $5 + $7, $4 + $5, $1, $2, $3)
   = < bal: $4 + $5 pend: 1 + $4 + $6 overdraft: true threshold: $4 + $5 + $1 funds: $2 > # $3 [ variant ] .

eq f4($4, $5, $6, 1 + $7, $1, $2, $3)
   = < bal: $4 + $5 pend: $4 + $6 overdraft: true threshold: $4 + $5 + $1 funds: $2 > # $3 [ variant ] .

eq f4($4, $5, 0, 0, $1, $2, $3)
   = < bal: 0 pend: 0 overdraft: false threshold: $4 + $5 + $1 funds: $2 > # $3 [ variant ] .

eq f4($4, $5, 1 + $6, $7, $1, $2, $3)
   = < bal: $4 + $5 pend: 1 + $4 + $6 overdraft: true threshold: $4 + $5 + $1 funds: $2 > # $3 [ variant ] .

rl [w] : < bal: n pend: x overdraft: false threshold: n + h funds: f > # msgs,w(m)
   => f1(m, n, x, h, f, msgs) .
rl [dep] : < bal: n pend: x overdraft: false threshold: n + m + h funds: f > # msgs,d(m)
   => < bal: n + m pend: x overdraft: false threshold: n + m + h funds: f > # msgs.
rl [w-req] : < bal: n + m + x pend: x overdraft: false threshold: n + m + x + h funds: f > # msgs
   => < bal: n + m + x pend: m + x overdraft: false threshold: n + m + x + h funds: f > # msgs,w(m) .
```