Static Correction of Maude Programs with Assertions[☆]

M. Alpuente[a], D. Ballis[b], J. Sapiña[a,*]

[a] *DSIC-ELP, Universitat Politècnica de València*
Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain
[b] *DIMI, University of Udine,*
Via delle Scienze, 206, 33100, Udine, Italy

Abstract

In this paper, we present a novel transformation method for Maude programs featuring both automatic program diagnosis and correction. The input of our method is a reference specification $\mathcal{A}$ of the program behavior that is given in the form of assertions together with an overly general program $\mathcal{R}$ whose execution might violate the assertions. Our correction technique translates $\mathcal{R}$ into a refined program $\mathcal{R}'$ in which every computation is also a computation in $\mathcal{R}$ that satisfies the assertions of $\mathcal{A}$. The technique is first formalized for topmost rewrite theories, and then we generalize it to larger classes of rewrite theories that support nested structured configurations. Our technique copes with infinite space states and does not require the knowledge of any failing run. We report experiments that assess the effectiveness of assertion-driven correction.

Keywords: program repair, assertion checking, program transformation, rewriting logic, equational rewriting, Maude

1. Introduction

Assertion checking is a practical means of validating programs. Program *assertions* often specify "can never happen" conditions that help detect pro-

[☆]This work has been partially supported by the EU (FEDER) and the Spanish MINECO under grant TIN2015-69175-C4-1-R, and by Generalitat Valenciana ref. PROMETEOII/2015/013.

[*]Corresponding author

Email addresses: alpuente@dsic.upv.es (M. Alpuente), demis.ballis@uniud.it (D. Ballis), jsapina@dsic.upv.es (J. Sapiña)

gram faults by testing, via runtime-checking, that the implementation behaves as intended on specific runs. Techniques to automatically detect discrepancies with the expected behavior given by the assertions are slowly making their way into industrial practice, yet the automated generation of valid program fixes is still a challenging problem with practical techniques having arisen only recently [1].

Maude [2] is a high-level programming language and system that efficiently implements Rewriting Logic [3], which is a logic of change that seamlessly unifies a wide variety of models of concurrency. Thanks to its logical basis, Maude provides a precise mathematical model that allows it to be used as a programming language and as a formal verification system. The language integrates: 1) functional, concurrent, logic, and object-oriented computations; 2) rich type structures with sorts, subsorts, and operator overloading; 3) equational reasoning modulo axioms such as commutativity, associativity-commutativity, and associativity-commutativity-identity of functions, which efficiently supports automated reasoning with typed data structures such as lists, sets, and multisets, and typical hierarchical/structural relations such as *is_a* and *part_of*. Maude is implemented as a high-performance interpreter (up to 2.98 million rewrites per second on a standard computer); a compiler is under development that brings this number up to dozens of millions of rewrites per second [4]. Because of its efficient rewriting engine and its metalanguage capabilities, Maude turns out to be an excellent instrument to create rich executable environments for various logics, programming languages, and tools.

Maude's formal tools are numerous and perform different analysis and verification tasks, either statically (e.g., Maude's theorem prover and model checker [2]) or dynamically (Maude's assertion checker [5, 6]). However, to the best of our knowledge, there is no theoretical foundation for integrating assertions into a general methodology for automated program correction in Maude. While the research on automated diagnosis of Maude programs has recently made substantial progress (see [5]), effective fault localization, debugging, and repair still pose some important challenges. In fact, although checking assertions at runtime greatly facilitates finding program bugs, it increases the overall execution time and the induced repairs heavily depend on specific failing runs. Static, automated program repair does not depend on concrete execution traces and has the potential to achieve significant cut downs on the cost of improving software quality, since no special, and time-consuming runtime infrastructure is needed to deliver program fixes.

Our correction transformation works with Maude programs, that is, rewrite theories of the form $\mathcal{R} = (\Sigma, E, R)$, where E is a (confluent and terminating) set of equations and R is a set of rules that is coherent w.r.t. the equations [7] (intuitively, this ensures that a rewrite step with R can always be postponed in favor of deterministically rewriting with E). In our methodology, rewrite theories are equipped with system assertions, with each assertion consisting of a pair $\Pi \,|\, \varphi$, where Π (the *state template*) is a term and φ (the *state invariant*) is a quantifier-free first-order formula with equality. Intuitively, system assertions specify those states such that, for every subterm of the state which fits the algebraic structure of Π with pattern matching substitution σ, the constraints given by $\varphi\sigma$ are satisfied.

Roughly speaking, a computation in a rewrite theory $\mathcal{R}$ is a sequence of states $\mathcal{C}$, where each state transition in the sequence $\mathcal{C}$ is computed by an application of a rewrite rule of $\mathcal{R}$. Given a set of system assertions $\mathcal{A}$, the notion of correction we seek is based on a binary relation $\leq^{\mathcal{A}}$ on rewrite theories, such that $\mathcal{R}' \leq^{\mathcal{A}} \mathcal{R}$ when $\mathcal{R}'$ is a correction of $\mathcal{R}$ w.r.t. $\mathcal{A}$, meaning that $\mathcal{R}'$ is a valid strenghtening of $\mathcal{R}$ that enforces $\mathcal{A}$. More precisely, $\mathcal{R}' \leq^{\mathcal{A}} \mathcal{R}$ holds when: 1) every computation of $\mathcal{R}'$ is a computation of $\mathcal{R}$ (i.e., no spurious states are produced); and 2) every assertion in $\mathcal{A}$ is satisfied by all states that appear in the computations of $\mathcal{R}'$. Note that points 1 and 2 ensure that $\mathcal{R}'$ is maximal in the sense that only those computations of $\mathcal{R}$ that contain assertion violations are removed from $\mathcal{R}'$.

Our correction transformation essentially works as follows. Starting from an overly general rewrite theory $\mathcal{R} = (\Sigma, E, R)$ (that is, a rewrite theory that contains all desired computations but may disprove some of the assertions), $\mathcal{R}$ is coerced by inserting suitable conditions (abetted by the assertions of $\mathcal{A}$) in the rules of R until a suitable correction is reached which satisfies all the assertions. The inserted conditions are defined by means of new equations that are added to E.

An important feature of our technique is that it applies to Full Maude [2], which is a powerful extension of Maude that is written in Maude itself and that gives support for object-oriented specification and advanced module operations. Particular care is put in the transformation to ensure both that the new equations added to E do not break Maude's executability requirements (i.e confluence, coherence, and termination) and that the transformed rules do not introduce spurious computations. The correction itself does not introduce new discrepancies or regressions, that is, repairing an assertion cannot cause the failure of another assertion downstream.

The automated correction technique that we propose in this paper is beyond the capabilities of current Maude tools. Furthermore, apart from the handling of concurrency, our technique provides semantic guarantees on the corrections: they do not remove good runs and reduce the number of failing assertions to zero. Since all assertions are always satisfied after the correction transformation for all inputs (rather than just for specific executions), runtime assertion checking can be completely omitted after the correction. This eliminates the need for massive assertion testing required by [5, 6] and improves both the safety and efficiency of programs.

A different, preliminary approach for automated program repair was sketched in [6] that is *dynamic* (it relies on concrete failing computations) and does not apply to undiscovered bugs, in contrast to the *static* correction transformation proposed in this work. Also, as a major advantage of this work, the verified program corrections are computed without resorting to time-expensive, monitored runtime environments.

Plan of the paper. After some technical preliminaries in Section 2, we introduce a running example that is used along the paper to illustrate the kind of corrections we aim to produce automatically. Section 3 summarizes our system assertion language, which allows safety properties $\Pi \mid \varphi$ to be defined expressing that any subterm of a reachable state that fits the pattern given by Π has to invariably satisfy φ. Section 4 encodes a checking mechanism for system assertions as an expanded equational theory $\mathcal{E}^{\mathcal{A}}$ (extending the original theory $\mathcal{E}$) that catches every possible assertion violation w.r.t. the assertion set $\mathcal{A}$. Section 5 formalizes our verified correction technique for topmost rewrite theories. Essentially, from the constraints given by the assertions, we synthesize correcting rule conditions that invoke the new functions defined in $\mathcal{E}^{\mathcal{A}}$. These conditions are used to proactively enable/disable the rule computations that are responsible for the undesired behaviors without the need for monitoring the program execution at runtime. Section 6 extends the correction methodology to more complex rewrite theories, including topmost theories modulo structural axioms such as associativity, commutativity and unity, and to *Russian doll* theories, which are structured in a more sophisticated, inductively nested way. Section 7 presents an experimental evaluation of a practical tool, called ÁTAME, that implements our program correction methodology. Its effectiveness is measured from several points of view: code size, execution time, as well as program transformation time. Section 8 discusses some related work and it concludes.

2. Equational Theories and Rewrite Theoriess

Let us recall some important notions that are relevant to this work. We assume some basic knowledge of term rewriting [8] and Rewriting Logic [3]. Some familiarity with the Maude language [9, 2] is also required.

Maude is a Rewriting Logic [3] specification and verification system whose operational engine is mainly based on a very efficient implementation of rewriting. Maude's basic programming statements are equations and rewrite rules. Equations express deterministic computations leading to a unique final result, and are used to model system states as terms of an algebraic data type. Rules express transitions between states and are used to naturally express concurrent, nondeterministic, and possibly nonterminating system computations.

We consider an (order-sorted) *signature* Σ of operators (i.e, function symbols), with a finite poset of sorts $(S, <)$ that models the usual subsort relation [9]. The connected components of $(S, <)$ are the equivalence classes $[s]$ corresponding to the least equivalence relation $\equiv_<$ containing $<$. We assume that each equivalence class of sorts contains a *top* sort that is a supersort of every other sort in the class. Given a sort s, $top(s)$ denotes the top sort of s. Additionally, we assume that the signature Σ includes a distinguished sort $\top$ that conceptually represents a universal supersort of all sorts in S; i.e., $\top$ types every possibile term[1]. The $\top$ sort is used to define auxiliary, universal operators that can be applied to every term independently of its specific sort.

An operator f of Σ in prefix notation is specified by notation $f : s_1 \dots s_n \to s$, $n \geq 0$, where $s_1 \dots s_n$ denote the sequence of argument sorts (that is, the arity of f), and s is the sort of the return value. When the arity of f is the empty sequence, f is called *constant*. An operator of Σ in mixfix notation can be specified by using underscores as place holders for the input arguments (e.g. $\_ \otimes \_ : s_1 s_2 \to s$). A finite, possibly empty, sequence of sorts is denoted by $\vec{s}$.

We also consider an S-sorted family $\mathcal{V} = \{\mathcal{V}_s\}_{s \in S}$ of disjoint variable sets. $\mathcal{T}(\Sigma, \mathcal{V})_s$ and $\mathcal{T}(\Sigma)_s$ are the sets of terms and ground terms of sort s, respectively. We write $\mathcal{T}(\Sigma, \mathcal{V})$ and $\mathcal{T}(\Sigma)$ for the corresponding term algebras. The set of variables that occur in a term t is denoted by $\mathcal{V}ar(t)$. By notation $x : s$, we denote that variable x has sort s. A simple syntactic

[1] Actually, in Maude, $\top$ is specified by the keyword `Universal`, which does not denote a real sort; it is instead a place holder for any known sort.

condition on Σ and $(S, <)$, called *preregularity* [9], ensures that each (well-formed) term t always has a least-sort possible among all sorts in S, which is denoted by $ls(t)$.

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 $\mathcal{P}os(t)$ denote the set of positions of t. By $t_{|w}$, we denote the *subterm* of t at position w, and by $t[s]_w$, we denote the result of *replacing the subterm* $t_{|w}$ by the term s in t. By $root(t)$, we denote the operator of t that occurs at position Λ.

A *substitution* $\sigma \equiv \{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$ is a mapping from the set of variables $\mathcal{V}$ to the set of terms $\mathcal{T}(\Sigma, \mathcal{V})$, which is equal to the identity everywhere except for a set of variables $\{x_1, \ldots, x_n\}$. By ε, we denote the *identity* substitution. The application of a substitution σ to a term t, denoted $t\sigma$, is defined by induction on the structure of terms:

$$t\sigma = \begin{cases} x\sigma & \text{if } t = x, x \in \mathcal{V} \\ f(t_1\sigma, \ldots, t_n\sigma) & \text{if } t = f(t_1, \ldots, t_n), n \geq 0 \end{cases}$$

Given two terms s and t, a substitution σ is the *matcher* of t in s, if $s\sigma = t$. The term t is an *instance* of the term s, iff there exists a matcher σ of t in s. Given two substitutions θ and θ', their *composition* $\theta\theta'$ is defined as $t(\theta\theta') = (t\theta)\theta'$ for every term t. We recall that composition is associative. Given a binary relation $\rightsquigarrow$, we denote the usual *transitive* (resp., *transitive and reflexive*) closure of $\rightsquigarrow$ by $\rightsquigarrow^+$ (resp., $\rightsquigarrow^*$).

A labelled *conditional* equation, or simply (*conditional*) equation, is an expression of the form $[l] : \lambda = \rho$ *if* C, where l is a label (i.e., a name that identifies the equation), $\lambda, \rho \in \mathcal{T}(\Sigma, \mathcal{V})$ (with $ls(\lambda) \equiv_< ls(\rho)$), and C is a (possibly empty) sequence $c_1 \wedge \ldots \wedge c_n$, where each c_i is a Boolean expression[2]. When the condition C is empty, we simply write $[l] : \lambda = \rho$.

A labelled *conditional* rewrite rule, or simply (*conditional*) rule, is an expression of the form $[l] : \lambda \Rightarrow \rho$ *if* C, where l is a label, $\lambda, \rho \in \mathcal{T}(\Sigma, \mathcal{V})$ (with $ls(\lambda) \equiv_< ls(\rho)$), and C is a (possibly empty) conjunction of Boolean expressions $c_1 \wedge \ldots \wedge c_n$.[3] When the condition C is empty, we simply write $[l] : \lambda \Rightarrow \rho$.

[2]Actually, Maude supports different kinds of conditions in equations such as equational conditions, membership tests, and matching conditions. Nonetheless, all of them can be interpreted as Boolean expressions whose canonical form is a truth value.

[3]Note that we prevent rule conditions from including Maude rewrite expressions, since

When no confusion can arise, rule and equation labels $[l]$ are often omitted. The term λ (resp., ρ) is called *left-hand side* (resp. *right-hand side*) of the rule $\lambda \Rightarrow \rho$ *if* C (resp. equation $\lambda = \rho$ *if* C).

Roughly speaking, a conditional rewrite theory [3] seamlessly combines a set R of conditional rewrite rules (or conditional term rewriting system, CTRS), with an equational theory $\mathcal{E}$ (also possibly conditional) that may include ordinary equations and axioms, i.e., distinguished equations expressing algebraic laws such as associativity (A), commutativity (C), and unity (U) of function symbols. Within this framework, the system states are typically represented as elements of an algebraic data type that is specified by the equational theory, while the system computations are modeled via the rewrite rules, which describe transitions between states and are applied modulo the equations and axioms.

More formally, an *(order-sorted) equational theory* $\mathcal{E}$ is a pair (Σ, E), where Σ is an order-sorted signature, $E = \Delta \cup B$ with Δ being a collection of (oriented) conditional equations and B a collection of equational axioms such as associativity, commutativity, and unity that can be associated with any binary operator of Σ. The equational theory $\mathcal{E}$ induces a congruence relation on the term algebra $\mathcal{T}(\Sigma, \mathcal{V})$, which is denoted by $=_E$.

A *conditional rewrite theory* (or simply, rewrite theory) is a triple $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an order-sorted equational theory and R is a set of conditional rewrite rules.

2.1. Modeling Concurrent Systems in Maude: Our running example

Concurrent systems can be formalized as rewrite theories. In Maude, rewrite theories are encoded by means of system modules, which are syntactic containers that list the signature (i.e. sorts, subsorts, and operators along with their axioms), and the variables in play, as well as the equations and rewrite rules of the rewrite theory to be specified. As for the algebraic axioms, they are indirectly expressed in Maude as attributes of their corresponding operator (using the `assoc`, `comm` and `id:` keywords) and are only used for B-matching. The user is unburdened from having to give explicit equational definitions of some operators (e.g., equality `==`, inequality `=/=`, arithmetic operations) since Maude provides them in a built-in way. Each

---

they are not currently supported by our correction technique that only handles deterministic, conditional rewrite rules.

syntactic element in the module is declared by using a rather intuitive keyword. For instance, sorts and subsorts are specified by keywords `sort` and `subsort`, operators by `op`, conditional equations and rules by `ceq` and `crl`, while unconditional equations and rules are specified by `eq` and `rl`. Keywords `sorts` and `ops` are abbreviations that can be used to declare multiple sorts and operators in a single line. Finally, the `ctor` attribute is used to specify constructor operators that are used to define program data structures.

The following Maude program will be used as a running example throughout the paper.

Example 2.1

The following Maude system module `CONTAINER-TERMINAL` encodes a rewrite theory that formalizes a concurrent system that models cargo manipulation in a container terminal. We simplify the model by disregarding some details that are irrelevant to our discussion such as the existence of unique container identifiers or the alignment of ship and cargo destinations.

Maude has very liberal views on identifiers which provides a very flexible syntax when combined with mixfix notation. However, it has consequences; the most obvious is its sensitivity to whitespace, which requires writing blank characters around all of the key tokens and even before the terminal '.' that ends each statement.

```
 1 mod CONTAINER-TERMINAL is pr INT + EXT-BOOL .

 2  sorts Container Cargo Ship Fleet State .
 3  subsort Container < Cargo .
 4  subsort Ship < Fleet .

 5  op c : Int -> Container [ctor] .
 6  op <_,_|_> : Int Int Cargo -> Ship [ctor] .
 7  op _:_ : Fleet Cargo -> State [ctor] .
 8  op none : -> Fleet [ctor] .
 9  op __ : Fleet Fleet -> Fleet [ctor assoc comm id: none] .
10  op nil : -> Cargo [ctor] .
11  op _,_ : Cargo Cargo -> Cargo [ctor assoc id: nil] .

12  vars W MAXW MAXS : Int .
13  vars CG CG1 CG2 : Cargo .
14  var FL : Fleet .

15  op size : Cargo -> Int .
16  eq size(nil) = 0 .
```

```
17  eq size(c(W),CG) = 1 + size(CG) .
18  op weight : Cargo -> Int .
19  eq weight(nil) = 0 .
20  eq weight(c(W),CG) = W + weight(CG) .
21  op isFull : Cargo -> Bool .
22  eq isFull(nil) = true .
23  eq isFull(c(W),CG) = (W == maxW_Container) and isFull(CG) .
24  op maxW_Container : -> Int .
25  eq maxW_Container = 5 .

26  crl [stow] : < MAXW,MAXS | CG > FL : c(W),CG1 =>
27               < MAXW,MAXS | CG,c(W) > FL : CG1
28               if weight(CG,c(W)) <= MAXW .

29  rl [unstow] : < MAXW,MAXS | c(W),CG > FL : CG1 =>
30                < MAXW,MAXS | CG > FL : CG1,c(W) .

31  crl [load] : < MAXW,MAXS | CG > FL : CG1,c(W),CG2 =>
32               < MAXW,MAXS | CG > FL : CG1,c(W + 1),CG2
33                if not(isFull(c(W))) .

34   rl [unload] : < MAXW,MAXS | CG > FL : CG1,c(W),CG2 =>
35                 < MAXW,MAXS | CG > FL : CG1,c(W - 1), CG2 .
36 endm
```

In our specification, system states are modeled by means of terms of sort `State` with the form `FL : CG`, where `FL` is a fleet (i.e., a multiset of ships) and `CG` is the cargo at the container terminal ready to be loaded (i.e., a list of containers).

A container of weight `W` is defined by a term `c(W)` of sort `Container`. A collection of containers is built by means of an AU binary operator `_,_` (whose *unity* or identity element is the constant `nil`) that basically models a list data structure. Specifically, a list of containers is a term of sort `Cargo`, whose form is either $\texttt{c(W}_1\texttt{),c(W}_2\texttt{),}\ldots\texttt{,c(W}_n\texttt{)}$ or `nil`.

Containers are stowed on (resp. unstowed from) ships following a *first in, first out* strategy. A ship with maximum allowed weight `MAXW`, maximum allowed capacity (number of containers on board) `MAXS`, and loaded cargo `CG` is defined by a term `< MAXW,MAXS | CG >` of sort `Ship`. A collection (multiset) of ships is a term of sort `Fleet` that is built by means of an ACU binary operator `__` whose identity element is the constant `none`. The considered sorts and their associated subsort relations are declared at the top of the `CONTAINER-TERMINAL` system module (lines 2–4), while the constructor op-

erators, which are required to define the system state structure, are specified by lines 5-11.

The Maude module also includes some equations that specify the auxiliary functions `size` (lines 15–17), `weight` (lines 18–20), and `isFull` (lines 21–23). The functions `size(CG)` and `weight(CG)` respectively return the number of containers and the total weight of the cargo list `CG`, whereas the Boolean function `isFull(CG)` checks whether each container in `CG` is completely filled, with `maxW_Container` (lines 24–25) being the maximum allowed weight of each container.

The system behavior is specified by means of four rewrite rules. The rule `stow` (lines 26–28) removes a container from the front of the terminal container list and loads it on an arbitrary ship provided that the total weight of its current cargo plus the weight of the considered container is lower than or equal to `MAXW`. Likewise, `unstow` (lines 29–30) removes a container from an arbitrary ship and adds it to the back of the container terminal cargo list. To simplify the model, the rule `load` (lines 31–33) increases by one unit the weight of an arbitrary container `c(W)` located at the cargo-terminal provided `c(W)` is not already full. Dually, the rule `unload` (lines 34–35) decreases the weight of `c(W)` by one unit.

Observe that the considered `CONTAINER-TERMINAL` system module can produce some awkward, certainly unwanted, system states. For instance, it is possible to reach a system configuration where the current number of loaded containers `CG` in a ship `< MAXW,MAXS | CG >` is greater than the allowable ship capacity `MAXS`. Moreover, there is no lower limit for the weight of a single container, which can lead to loading containers with a negative weight.

2.2. Rewriting in Conditional Rewrite Theories

In a rewrite theory (Σ, E, R), with $E = \Delta \cup B$, the concurrent system evolves by rewriting states using *equational rewriting*, i.e., rewriting with the rewrite rules in R *modulo the equations and axioms* in E [3].

The Maude interpreter implements equational rewriting by means of two simple relations, namely $\rightarrow_{\Delta,B}$ and $\rightarrow_{R,B}$. These allow rules and equations to be intermixed in the rewriting process by simply using both an algorithm of matching modulo B[4]. Roughly speaking, the relation $\rightarrow_{\Delta,B}$ uses the equa-

[4]Particularly important instances of B-matching occur when B specifies the following combinations of algebraic axioms for an operator *op*: associative axioms (A), associative

tions of Δ (oriented from left to right) as simplification rules. Thus, for any term t, by repeatedly applying the equations as simplification rules, we eventually reach a term $t{\downarrow_{\Delta,B}}$ to which no further equations can be applied. The term $t{\downarrow_{\Delta,B}}$ is called a *canonical form* (or irreducible form) of t w.r.t. Δ modulo B. On the other hand, the relation $\rightarrow_{R,B}$ implements rewriting with the rules of R, which might be non-terminating and non-confluent, whereas Δ is required to be terminating and Church-Rosser (i.e., confluent and sort-decreasing) modulo B [7] in order to guarantee the existence and unicity (modulo B) of a canonical form w.r.t. Δ for any term [9].

Formally, $\rightarrow_{R,B}$ and $\rightarrow_{\Delta,B}$ are defined as follows. Given a rewrite rule $[r] : (\lambda \Rightarrow \rho \; \mathit{if}\; C) \in R$ (resp., an equation $[e] : (\lambda = \rho \; \mathit{if}\; C) \in \Delta$), a substitution σ, a term t, and a position w of t, $t \stackrel{r,\sigma,w}{\rightarrow}_{R,B} t'$ (resp., $t \stackrel{e,\sigma,w}{\rightarrow}_{\Delta,B} t'$) iff $\lambda\sigma =_B t_{|w}$, $t' = t[\rho\sigma]_w$, and C *evaluates to true* w.r.t σ. When no confusion arises, we simply write $t \rightarrow_{R,B} t'$ (resp. $t{\rightarrow_{\Delta,B}}t'$) instead of $t \stackrel{r,\sigma,w}{\rightarrow}_{R,B} t'$ (resp. $t \stackrel{e,\sigma,w}{\rightarrow}_{\Delta,B} t'$).

Roughly speaking, a conditional rewrite step on the term t applies a rewrite rule/equation to t by replacing a *red*ucible (sub-)*ex*pression of t (namely $t_{|w}$), called the *redex*, by its contracted version $\rho\sigma$, called the *contractum*, whenever the condition C is fulfilled. Note that the evaluation of a condition C is typically a recursive process since it may involve further (conditional) rewrites in order to normalize C to *true*.

An essential executability condition for equational rewriting is coherence between rules in R and equations in Δ modulo the axioms in B, which basically states that the effect of rewriting modulo $\Delta \cup B$ can be achieved by intermingling rewriting with both R and Δ modulo B[5]. More precisely, under coherence conditions, an equational rewrite step on a term s can be implemented without loss of completeness by applying the following rewrite strategy:

1. **Equational simplification of s in Δ modulo B**, that is, reduce s using $\rightarrow_{\Delta,B}$ until the canonical form w.r.t. Δ modulo B ($s \downarrow_{\Delta,B}$) is reached;

---

and unity axioms (AU), associative and commutative axioms (AC), associative, commutative, and unity axioms (ACU). In the cases when *op* obeys A or AU (respectively, AC or ACU), any term rooted with *op* is implicitly handled as a list (respectively, a multiset).

[5] A precise characterization of the coherence property and a sufficient condition to guarantee it for a wide class of conditional rewrite theories can be found in [7].

2. **Rewrite** $(s \downarrow_{\Delta,B})$ **in** R **modulo** B to t' using $\rightarrow_{R,B}$, where $t' \in [t]_E$.

A *computation* (trace) $\mathcal{C}$ for s_0 in the conditional rewrite theory $(\Sigma, \Delta \cup B, R)$ is then deployed as the (possibly infinite) rewrite sequence

$$s_1 \rightarrow^*_{\Delta,B} s_1{\downarrow_{\Delta,B}} \rightarrow_{R,B} s_2 \rightarrow^*_{\Delta,B} s_2{\downarrow_{\Delta,B}} \rightarrow_{R,B} \ldots$$

that interleaves $\rightarrow_{\Delta,B}$ rewrite steps and $\rightarrow_{R,B}$ rewrite steps following the strategy mentioned above. Note that, after each rewrite step using $\rightarrow_{R,B}$, generally the resulting term s_i, $i = 1, \ldots, n$, is not in canonical normal form and is thus equationally simplified (or normalized) by using $\rightarrow_{\Delta,B}$ before the subsequent rewrite step using $\rightarrow_{R,B}$ is performed. Also, in the precise strategy adopted by Maude, the last term of a finite computation is finally normalized before the result is delivered. Therefore, any computation in $\mathcal{R}$ can be conveniently interpreted as a sequence of juxtaposed computation steps $s_1 \rightarrow_{\mathcal{R}} s_2 \rightarrow_{\mathcal{R}} s_3 \rightarrow_{\mathcal{R}}, \ldots$ where each s_i, with $i > 0$, is a canonical form. Terms s_i (and their canonical forms $s_i \downarrow_{\Delta,B}$) that appear in a computation are called *states*, and any sequence $s_i \rightarrow^*_{\Delta,B} s_i{\downarrow_{\Delta,B}} \rightarrow_{R,B} s_{i+i} \rightarrow^*_{\Delta,B} s_{i+i}{\downarrow_{\Delta,B}}$ from s_i to the canonical form of s_{i+i} is called a *computation step* for s_i.

Example 2.2

Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be the rewrite theory encoded in the Maude `CONTAINER-TERMINAL` module of Example 2.1. Then,

```
< 5,3 | nil > : c(1),c(3)  →R,B < 5,3 | nil > : c(1 + 1),c(3)
                           →*Δ,B < 5,3 | nil > : c(2),c(3)
```

is a computation step in $\mathcal{R}$ that first rewrites the initial state by applying the rule `load` and then simplifies the resulting state by using the built-in definition for natural addition.

In the following section, we briefly recall the assertion language of [5] that allows formal properties of software systems to be specified in rewriting logic.

3. The System Assertion Language

Given the rewrite theory $\mathcal{R} = (\Sigma, \Delta \cup B, R)$, we introduce an order-sorted assertion language $\mathcal{L}$ whose first-order formulas define properties of the system states of $\mathcal{R}$. To simplify our formulation, we assume that the formulas of $\mathcal{L}$ are built over a set of user-defined Boolean operators (predicates) of

Σ whose semantics is specified by some equations in Δ. The truth values of $\mathcal{L}$ are given by the formulas *true* and *false*, and the usual conjunction (*and*), disjunction (*or*), exclusive or (*xor*), negation (*not*), and implication (*implies*) logic operators are used to express composite properties. Variables in the formulas are not quantified.

Within this logic framework, we define system assertions (assertions for short) as *constrained terms* [10]. Formally, a *system assertion* is a pair $\Pi \mid \varphi$, where φ is a quantifier-free Boolean formula that is specialized to a (typically non-ground) term Π in $\mathcal{T}(\Sigma, \mathcal{V})$, with $\mathcal{V}ar(\varphi) \subseteq \mathcal{V}ar(\Pi)$.

Expressiveness and flexibility of our system assertion language rely on Maude's powerful capabilities for equational matching and reasoning that allow complex state templates Π and invariants φ to be specified in a concise and flexible way. System assertions can virtually characterize any set of system states by defining pure declarative representations of state templates and invariants that can predicate on the whole system structure or just on fragments of it. When several distinct, top-level state constructors are declared, we can define several assertions, one for each constructor in order to capture all of the possible states. Also, for the case when states are made of lists or sets, our assertion language is expressive enough to relate (e.g., count the number of) elements in the list/set. For instance, consider a Maude system module that represents states as lists/sets of elements of the form $\texttt{<a}_1, \texttt{a}_2, \texttt{a}_3, \ldots, \texttt{a}_n\texttt{>}$. We can easily define a system assertion that enforces a property `p(A)` (only) on states that contain more than 100 elements, as follows:

```
<A> | sz(A)>100 implies p(A)
```

where `A` is a variable of the (list/set) state sort, `sz` is a user-defined function that counts the elements of a list/set (that can be equationally specified by means of a simple recursion scheme), and `p(A)` is a user-defined property over the list/set `A`.

Operationally, a system assertion $\Pi \mid \varphi$ defines a generic safety property for a state t which specifies a logic invariant φ which must be enforced in any subterm of t that is an instance (modulo $\Delta \cup B$) of Π. More formally, a state t *satisfies* $\Pi \mid \varphi$ (in symbols, $t \models \Pi \mid \varphi$) iff for every $w \in \mathcal{P}os(t)$ and for every substitution σ, if $t_{|w} =_{\Delta \cup B} \Pi\sigma$ then $\varphi\sigma \rightarrow^*_{\Delta,B} \mathit{true}$. The notion of satisfaction can be naturally lifted to sets of assertions: given a set of

assertions $\mathcal{A}$, t *satisfies* $\mathcal{A}$ (in symbols, $t \models \mathcal{A}$), iff $t \models a$ for each assertion $a \in \mathcal{A}$. A *violation* of a system assertion a is detected in a term t, whenever $t \not\models a$, that is, when there exist a position $w \in \mathcal{P}os(t)$ and a substitution σ such that $t_{|w} =_{\Delta \cup B} \Pi\sigma$ and $\varphi\sigma \rightarrow^*_{\Delta,B} \mathit{false}$.

Verification of safety properties amounts to the problem of invariance checking. Hence, we say that a computation $\mathcal{C}$ is *safe* w.r.t. an assertion set $\mathcal{A}$, if for each state s in $\mathcal{C}$, $s \models \mathcal{A}$.

Example 3.1

Let us reinforce safety of the container terminal model of Example 2.1 by specifying some desired properties. Let $\mathcal{A}$ be the assertion set that includes:

```
(a1)  c(W) | W >= 0 and  W <= 5
(a2)  < MAXW,MAXS | CG > | weight(CG) <= MAXW and size(CG) <= MAXS
(a3)  CG1,c(W),CG2 | isFull(c(W)) implies isFull(CG1)
```

The assertion `a1` requires that every container has a weight `W` ranging from 0 tons to 5 tons independently of its location (ship or container terminal), while the assertion `a2` asserts that every ship configuration has a cargo weight and a total number of containers which must not exceed `MAXW` and `MAXS`, respectively.

Finally, `a3` specifies that, for any container list `CG1,c(W),CG2`, if the container `c(W)` is completely filled, then the containers in the list `CG1` are also filled. The goal of this last assertion is to promote container loading over container stowing. In fact, full containers are stowed into a ship before nonfull containers through the stowing fifo strategy encoded by the rule `stow`.

In the rest of the paper, we present a static correction technique for a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ w.r.t. the assertion set $\mathcal{A}$ that is formalized as a two-phase procedure. First, by using $\mathcal{A}$, the equational theory (Σ, E) in $\mathcal{R}$ is augmented with a new set of equations E' that identify all possible system assertion violations. Then, the rules of R are transformed into conditional rules by including guarding conditions that invoke the equations of E'. This guarantees that the application of the transformed rules always produces states that satisfy $\mathcal{A}$, and thus computations in the transformed theory are safe w.r.t. the assertion set $\mathcal{A}$. Clearly, truncating program behaviours of infinite computations could eventually violate liveness and introduce deadlocks. Nevertheless, satisfaction of liveness properties can be eventually analyzed after the correction transformation by means of Maude's model-checker [11].

4. Equational Encoding of System Assertion Violations

Let $\mathcal{E} = (\Sigma, \Delta \cup B)$ be an equational theory and $\mathcal{A}$ be a set of system assertions that are built using a set of predicates P. Without loss of generality, we assume that the equational definition of P is included in $\mathcal{E}$.

We aim to construct an extension of the equational theory $\mathcal{E}$ in which violations of the assertions in $\mathcal{A}$ can be automatically detected. The following definition provides a renaming procedure that is instrumental for the transformation. Roughly speaking, the goal of such a renaming is to neatly separate assertion checking from system computations so that they do not interfere: the overall system execution uses the original operators and equations in the equational theory, while assertion checking is performed on renamed terms. This is not only convenient but it is key to guarantee that the correction transformation does not break executability of the original theory by ensuring that confluence, termination, and coherence are preserved.

Definition 4.1 (renaming extension of an equational theory) *Let $\mathcal{E} = (\Sigma, \Delta \cup B)$ be an equational theory where $(S, <)$ is the sort poset included in Σ. Let $M \subseteq S$ contain the top sorts of S, $M = \{s \in S \mid \nexists\ s' \in S \text{ such that } s < s'\}$.*

The renaming extension *of the equational theory $\mathcal{E}$ is the equational theory $\mathcal{E}' = (\Sigma \cup \Sigma', \Delta \cup \Delta' \cup B)$ such that*

1. *$(S, <) = (S', <')$ where $(S', <')$ is the sort poset included in Σ' (that is, the renaming extension $\mathcal{E}'$ preserves the sort structure of $\mathcal{E}$).*
2. *$\Sigma' \cap \Sigma = \emptyset$, and Σ' contains one* renamed *operator $f' \colon s_1 \dots s_n \to s$, $n \geq 0$, for each operator $f \colon s_1 \dots s_n \to s$ of Σ, and f' is given the same equational attributes as f. Furthermore, Σ' includes two universal operators $\mathsf{Ren} \colon \top \to \top$ and $\mathsf{Ren}^{-1} \colon \top \to \top$.*
3. *Δ' contains the following equations that define the operators Ren and Ren^{-1}, for every possible term in $\mathcal{T}(\Sigma, \mathcal{V})$ and $\mathcal{T}(\Sigma', \mathcal{V})$:*
   (a) *$\mathsf{Ren}(f(x_1, \dots, x_n)) = f'(\mathsf{Ren}(x_1), \dots, \mathsf{Ren}(x_n))$, with $x_i : s_i \in \mathcal{V}$, i=1,...,n, for every function symbol $f \colon s_1 \dots s_n \to s$ of Σ with arity n, $n \geq 0$, such that f obeys no unity axiom.*
   (b) *$\mathsf{Ren}^{-1}(f'(x_1, \dots, x_n)) = f(\mathsf{Ren}^{-1}(x_1), \dots, \mathsf{Ren}^{-1}(x_n))$, with $x_i : s_i \in \mathcal{V}$, i=1,...,n, for every $f' \colon s_1 \dots s_n \to s$ of Σ' such that $f' \in \Sigma'$ obeys no unity axiom.*

(c) *For the case of a binary symbol* $f\colon s_1\ s_2 \to s$ *(and their renamed version* f'*) with unity element* `id`*, the following conditional equations are also contained in* Δ'*:*

$$\mathsf{Ren}(f(x_1, x_2)) = f'(\mathsf{Ren}(x_1), \mathsf{Ren}(x_2)) \text{ if } x_1 =/= \mathsf{id} \wedge x_2 =/= \mathsf{id}$$
$$\mathsf{Ren}^{-1}(f'(x_1, x_2)) = f(\mathsf{Ren}^{-1}(x_1), \mathsf{Ren}^{-1}(x_2)) \text{ if } x_1 =/= \mathsf{id} \wedge x_2 =/= \mathsf{id}$$

where $x_1\colon s_1, x_2\colon s_2 \in \mathcal{V}$.

(d) $\mathsf{Ren}(x) = x$ [`owise`] *with* $x : s \in \mathcal{V}$*, for every* $s \in M$.
(e) $\mathsf{Ren}^{-1}(x) = x$ [`owise`] *with* $x : s \in \mathcal{V}$*, for every* $s \in M$.

Note that the extra constraint $x_1 =/= \mathsf{id} \wedge x_2 =/= \mathsf{id}$ of Case (c) is required to avoid non-termination of the renaming process. Indeed, since t is equivalent to $f(t, \mathsf{id})$ modulo the unity axiom for f, if the condition was omitted, the following non-terminating equational simplification could be delivered for any t, where $\mathsf{Ren}(t)$ is simplified infinitely often (and similarly for $\mathsf{Ren}^{-1}(t')$ for any t')

$$\mathsf{Ren}(t) =_B \mathsf{Ren}(f(t, \mathsf{id})) \to_{\Delta', B} f'(\mathsf{Ren}(t), \mathsf{Ren}(\mathsf{id})) \to_{\Delta', B} f'(f'(\mathsf{Ren}(t), \mathsf{Ren}(\mathsf{id})), \mathsf{Ren}(\mathsf{id})) \ldots$$

We enforce that terms of the form $\mathsf{Ren}(t)$, with $t \in \mathcal{T}(\Sigma, \mathcal{V})$, are evaluated using an eager[6] rewrite strategy that first simplifies the input argument t into the canonical form $t{\downarrow_{\Delta,B}}$ and then applies the equational definition of Ren to $t \downarrow_{\Delta,B}$. The eager strategy avoids potential interferences between the renaming and the simplification of a term t. In fact, it guarantees that renaming occurs only after term simplification within the original equational theory $(\Sigma, \Delta \cup B)$. More specifically, given a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, $\mathsf{Ren}(t) = t'{\downarrow_{\Delta,B}}$ where $t' \downarrow_{\Delta,B}$ is a term in $\mathcal{T}(\Sigma', \mathcal{V})$ that is computed by replacing each operator *op* of the canonical form $t{\downarrow_{\Delta,B}}$ with *op*$'$.

Dually, we enforce that $\mathsf{Ren}^{-1}(t')$ is evaluated by means of a lazy rewrite strategy that simplifies $\mathsf{Ren}^{-1}(t')$ by applying the equations for the operator Ren^{-1} at its root position, thereby undoing the renaming of t' and recursively restoring the (canonical form of the) original term t.

[6]Eager and lazy rewrite strategies can be straightforwardly defined in Maude by resorting to the `strat` operator attribute, which allows term evaluation orders to be precisely defined.

Example 4.2

Consider the `CONTAINER-TERMINAL` module of Example 2.1 together with the following container list

$$\texttt{c(X),c(3 + 1),c(Y)}$$

where `X` and `Y` are variables. Note that the list is not a canonical form; indeed, by applying the built-in sum of integers it can be simplified into the canonical form $\texttt{c(X),c(4),c(Y)}$.

Therefore, the renaming $\mathsf{Ren}(\texttt{c(X)}, \texttt{c(3 + 1)}, \texttt{c(Y)})$ yields the renamed canonical form

$$\texttt{c}'\texttt{(X)},'\ \texttt{c}'\texttt{(4}'\texttt{)},'\ \texttt{c}'\texttt{(Y)}.$$

Further, $\mathsf{Ren}^{-1}(\texttt{c}'\texttt{(X)},'\ \texttt{c}'\texttt{(4}'\texttt{)},'\ \texttt{c}'\texttt{(Y)})$ returns the term $\texttt{c(X),c(4),c(Y)}$.

Let us now provide an equational representation of the assertion set $\mathcal{A}$, in which we specify a checking equation e_a for each assertion $a \in \mathcal{A}$. The goal of e_a is to catch every possible violation of a inside any computation state in the rewrite theory $\mathcal{R}$.

Definition 4.3 (assertion-checking equations) *Let $(\Sigma \cup \Sigma', \Delta \cup \Delta' \cup B)$ be the renaming extension of the equational theory $\mathcal{E} = (\Sigma, \Delta \cup B)$. Given the system assertion $a = \Pi \mid \varphi$, the* assertion-checking *equation for a, in symbols e_a, is the conditional equation*

$$\Pi' = \mathsf{fail} \text{ if not}(\mathsf{Ren}^{-1}(\varphi)) \ .$$

where $\Pi' = \mathsf{Ren}(\Pi)$, and fail is a new, universal constant (not included in $\Sigma \cup \Sigma'$) of sort $\top$. Given the assertion set $\mathcal{A}$, we define $\mathsf{A} = \{e_a \mid a \in \mathcal{A}\}$.

The idea is now to expand the renaming extension $\mathcal{E}'$ of the equational theory $\mathcal{E}$ with A, and to use it to detect any assertion violations at runtime. More specifically, given an assertion $a \in \mathcal{A}$ and a system state st, a single application of e_a to a renamed version $\mathsf{Ren}(st)$ of st would reduce any subterm of $\mathsf{Ren}(st)$ that matches Π' to fail, hence signalling that st violates the assertion a.

The renaming operator Ren and its dual counterpart Ren^{-1} are key in producing a corrected theory whose computations are safe w.r.t. $\mathcal{A}$. Indeed, if we consider an assertion-checking equation $e_a = (\Pi' = \mathsf{fail} \text{ if not}(\mathsf{Ren}^{-1}(\varphi)))$, we can observe the two following facts.

- On the one hand, the application of Ren^{-1} to the logic formula φ ensures that any instance $\varphi\sigma$ of φ does not contain renamed terms, and thus $\varphi\sigma$ can be properly simplified to its truth value by using the predicates P that are specified in the original equational theory $\mathcal{E}$.
- On the other hand, the renaming Π' of Π is needed for e_a to be terminating as shown in the following example.

Example 4.4

Consider that the `CONTAINER-TERMINAL` module of Example 2.1 is augmented by defining a new predicate function `empty?` that returns `true` if there is no container in the considered ship, and `false` otherwise

```
empty?(< MAXW,MAXS | none >) = true .
empty?(< MAXW,MAXS | CG >) = false [owise] .
```

Let the assertion set $\mathcal{A}$ consist of the single assertion

$$a = \texttt{< MAXW,MAXS | CG > | not(empty?(< MAXW,MAXS | CG >))}$$

that enforces all possible ship configurations to contain at least one container, and whose associated assertion-checking equation e_a is[7]

$$\texttt{< MAXW,MAXS | CG >}' = \mathsf{fail} \quad \texttt{if not}(\mathsf{Ren}^{-1}(\texttt{not(empty?(< MAXW,MAXS | CG >))}))$$

Now, consider the following variant of the above equation which omits renaming of the state template `< MAXW,MAXS | CG >`:

$$[e_\infty] : \texttt{< MAXW,MAXS | CG >} = \mathsf{fail}\ \texttt{if not(not(empty?(< MAXW,MAXS | CG >)))}$$

Note that any attempt to use e_∞ enters an infinite loop when trying to evaluate the subterm `empty?(< MAXW,MAXS | CG >)` in its condition, since this requires using the equation e_∞ itself once again.

[7] Note that, in the case of a single mixfix operator such as the `Ship` constructor $< \_, \_|\_ >$, for simplicity, we just rename one operator symbol, $< \_, \_|\_ >'$.

On the contrary, the renamed state template `< MAXW,MAXS | CG >'` in the left-hand side of e_a prevents the equation from being recursively used to evaluate its condition $\texttt{not}(\mathsf{Ren}^{-1}(\texttt{not(empty?(< MAXW,MAXS | CG >))}))$ where, indeed, there are no renamed terms that can match the left-hand side of $e_{\texttt{a}}$.

It is immediate to see that incorporating the set A of assertion-checking equations into the renaming extension $\mathcal{E}'$ of the confluent and terminating theory $\mathcal{E}$ yields a terminating equational theory. However, the following example shows that this naïvely extended theory might be nonconfluent since more than one irreducible form might exist for a given (renamed) system state when multiple assertion violations are detected within the state.

Example 4.5

Consider again the `CONTAINER-TERMINAL` system module of Example 2.1 together with the assertion set $\mathcal{A}$ of Example 3.1, whose associated assertion-checking equations are

$$[e_{\texttt{a1}}] : \mathsf{c}'(\texttt{W}) = \mathsf{fail}\ \texttt{if not}(\mathsf{Ren}^{-1}(\texttt{W >= 0 and W <= 5}))$$

$$[e_{\texttt{a2}}] : \texttt{< MAXW,MAXS | CG >}' = \mathsf{fail}\ \texttt{if not}(\mathsf{Ren}^{-1}(\texttt{weight(CG) <= MAXW and size(CG) <= MAXS}))$$

$$[e_{\texttt{a3}}] : \texttt{CG1},'\, \mathsf{c}'(\texttt{W}),'\, \texttt{CG2} = \mathsf{fail}\ \texttt{if not}(\mathsf{Ren}^{-1}(\texttt{isFull(c(W)) implies isFull(CG1)}))$$

Now, observe that confluence can be broken by applying equations $e_{\texttt{a1}}$ and $e_{\texttt{a3}}$ to the renamed container list

$$\mathsf{c}'(6'),'\, \mathsf{c}'(5').$$

Indeed, both assertions `a1` and `a3` are violated in the given container list. The former is not satisfied because $\mathsf{c}'(6')$ exceeds the weight upper limit of 5 tons, while the latter is refuted since there is a full container $\mathsf{c}'(5')$ that is preceded in the container list by the overweighted container $\mathsf{c}'(6')$ (which actually should be full but not overweighted). Finally, note that the applications of the equations $e_{\texttt{a1}}$ and $e_{\texttt{a3}}$ to

$$\mathsf{c}'(6'),'\, \mathsf{c}'(5')$$

yield two distinct irreducible forms, which are, respectively,

$$\mathsf{fail},'\, \mathsf{c}'(\mathsf{5}') \quad \text{and} \quad \mathsf{fail}.$$

Nevertheless, we are able to recover confluence by providing the transformed theory with additional equations that reduce every (renamed) state that contains a fail subterm to the unique irreducible form fail. Formally,

Definition 4.6 (fail-detecting equations) *Let $(\Sigma \cup \Sigma', \Delta \cup \Delta' \cup B)$ be the renaming extension of the equational theory $(\Sigma, \Delta \cup B)$. For every renamed operator $f'\colon s_1 \dots s_n \to s \in \Sigma'$, $n \geq 0$, we define the set of fail-detecting equations $\boldsymbol{F}_{f'}$ such that*

$$\boldsymbol{F}_{f'} = \begin{cases} \{f'(x_1, \dots, x_{i-1}, \mathsf{fail}, x_{i+1}, \dots, x_n) = \mathsf{fail} \mid i = 1, \dots, n\} \\ \qquad\qquad \textit{for every } f' \in \Sigma' \textit{ without unity}, n \geq 0 \\ \{f'(x_1, \mathsf{fail}) \textit{ if } x_1 =/= \mathsf{fail}\} \cup \{f'(\mathsf{fail}, x_2) \textit{ if } x_2 =/= \mathsf{fail}\} \\ \qquad\qquad \textit{for every binary operator } f' \in \Sigma' \textit{ with unity } \mathsf{id} \end{cases}$$

Furthermore, we define $\mathsf{F} = \bigcup_{f' \in \Sigma'} \boldsymbol{F}_{f'}$.

Similarly to Definition 4.1, Definition 4.6 deals with binary operators with unity in a special way to avoid nontermination of the fail-detecting equations.

Now, we are ready to formalize the $\mathcal{A}$-extension of an equational theory $(\Sigma, \Delta \cup B)$.

Definition 4.7 ($\mathcal{A}$-extension of $\mathcal{E}$) *Let $\mathcal{E} = (\Sigma, \Delta \cup B)$ be an equational theory and $\mathcal{A}$ be an assertion set. Let $\mathcal{E}' = (\Sigma \cup \Sigma', \Delta \cup \Delta' \cup B)$ be the renaming extension of $\mathcal{E}$. Then, the* $\mathcal{A}$-extension *of $\mathcal{E}$ is the equational theory $\mathcal{E}^{\mathcal{A}} = (\Sigma^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B)$ such that*

- $\Sigma^{\mathcal{A}} = \Sigma \cup \Sigma' \cup \{\mathsf{fail} :\to \top\}$
- $\Delta^{\mathcal{A}} = \Delta \cup \Delta' \cup \mathsf{A} \cup \mathsf{F}$

Note that no confluence is lost by joining A with F because all of the equations in $\mathsf{A} \cup \mathsf{F}$ have the same right-hand side, fail.

Example 4.8

Let $\mathcal{E} = (\Sigma, \Delta \cup B)$ denote the equational theory that is encoded in the `CONTAINER-TERMINAL` module of Example 2.1 and consider again the assertion set $\mathcal{A}$ of Example 3.1. The $\mathcal{A}$-extension $(\Sigma^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B)$ of the equational theory $\mathcal{E}$ is obtained by expanding the renaming extension $\mathcal{E}' = (\Sigma', \Delta' \cup B)$ of $\mathcal{E}$ with the assertion-checking equations $e_{\mathtt{a1}}$, $e_{\mathtt{a2}}$, and $e_{\mathtt{a3}}$ of Example 4.5, and the set of the fail-detecting equations, which specifically includes

$$\mathsf{F}_{\_,'\_} = \{(\mathtt{X},'\mathsf{fail} = \mathsf{fail} \text{ if } \mathtt{X} =/= \mathtt{nil}), (\mathsf{fail},'\mathtt{X} = \mathsf{fail} \text{ if } \mathtt{X} =/= \mathtt{nil})\}$$

$$\mathsf{F}_{\mathtt{<}\_,\_\mid\_\mathtt{>}'} = \{(\mathtt{<}\,\mathsf{fail},\mathtt{X} \mid \mathtt{Y}\,\mathtt{>}' = \mathsf{fail}), (\mathtt{<}\,\mathtt{X},\mathsf{fail} \mid \mathtt{Y}\,\mathtt{>}' = \mathsf{fail}), (\mathtt{<}\,\mathtt{X},\mathtt{Y} \mid \mathsf{fail}\,\mathtt{>}' = \mathsf{fail})\}$$

Consider now a term $\mathtt{ship} = \mathtt{<\ 20,3\ \mid\ c(8),c(9)\ >}$ which raises two violations of the assertion $\mathtt{a1} \in \mathcal{A}$, since both `c(8)` and `c(9)` are overweighted. These violations are captured by the following equational simplifications that reduce $\mathsf{Ren}(\mathtt{ship})$ to the unique irreducible form fail by using the assertion-checking equation $e_{\mathtt{a1}}$ as well as the fail-detecting equations in $\mathsf{F}_{\_,'\_}$ and $\mathsf{F}_{\mathtt{<}\_,\_\mid\_\mathtt{>}'}$.

$$\begin{aligned}\mathsf{Ren}(\mathtt{ship}) &\rightarrow_{\Delta^{\mathcal{A}},B} \mathtt{<\ 20',3'\ \mid\ fail,'c'(9')\ >'} \\ &\rightarrow_{\Delta^{\mathcal{A}},B} \mathtt{<\ 20',3'\ \mid\ fail\ >'} \\ &\rightarrow_{\Delta^{\mathcal{A}},B} \mathsf{fail}\end{aligned}$$

$$\begin{aligned}\mathsf{Ren}(\mathtt{ship}) &\rightarrow_{\Delta^{\mathcal{A}},B} \mathtt{<\ 20',3'\ \mid\ c'(8'),'\ fail\ >'} \\ &\rightarrow_{\Delta^{\mathcal{A}},B} \mathtt{<\ 20',3'\ \mid\ fail\ >'} \\ &\rightarrow_{\Delta^{\mathcal{A}},B} \mathsf{fail}\end{aligned}$$

Given an equational theory $\mathcal{E}$ and an assertion set $\mathcal{A}$, the next result formally states that any violation of $\mathcal{A}$ within the canonical form of a term t can be captured by evaluating $\mathsf{Ren}(t)$ in the $\mathcal{A}$-extension of $\mathcal{E}$.

Proposition 4.9 (completeness) *Let $\mathcal{E}$ be an equational theory, $\mathcal{A}$ be an assertion set, and $\mathcal{E}^{\mathcal{A}} = (\Sigma^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B)$ be the $\mathcal{A}$-extension of $\mathcal{E}$. Let $a \in \mathcal{A}$ and $st \in \mathcal{T}(\Sigma, \mathcal{V})$. If $st{\downarrow_{\Delta,B}} \not\models a$, then $\mathsf{Ren}(st) \rightarrow^*_{\Delta^{\mathcal{A}},B} \mathsf{fail}$.*

5. Assertion-driven Correction of Topmost Rewrite Theories

In this section, we formalize an assertion-driven correction methodology for the class of *topmost* rewrite theories, that is, theories in which terms can only be rewritten at the root position. This class is of primary importance in the Rewriting Logic framework since a sound and complete procedure exists for goal reachability in topmost theories that has many practical applications (e.g., the analysis of security protocols [12]).

A topmost rewrite theory can be defined as follows [13].

Definition 5.1 *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory. Let S be the set of sorts of Σ. Then, $\mathcal{R}$ is* topmost *if, for some top sort State $\in S$,*

1. *for each rule $(\lambda \Rightarrow \rho$ if $C)$ in R, λ and ρ are terms of sort State;*
2. *there is no operator in Σ whose arity includes a sort s such that $top(s) = State$.*

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

Example 5.2

The `CONTAINER-TERMINAL` module of Example 2.1 encodes a topmost rewrite theory.

Essentially, our correction technique transforms the rewrite rules of a given topmost rewrite theory $\mathcal{R}$ into guarded, conditional rewrite rules that can only be fired if no system assertion is violated. The transformation builds on the notion of $\mathcal{A}$-extension of $\mathcal{R}$ that we defined in Section 4.

Before formalizing the theory correction methodology, let us also precisely characterize the notion of correction of a rewrite theory $\mathcal{R}$ w.r.t. an assertion set $\mathcal{A}$.

Definition 5.3 *Let $\mathcal{R}$ be a rewrite theory and $\mathcal{A}$ be an assertion set. The rewrite theory $\mathcal{R}'$ is a correction of $\mathcal{R}$ w.r.t. $\mathcal{A}$ (in symbols $\mathcal{R}' \leq^{\mathcal{A}} \mathcal{R}$) if the following requirements hold:*

1. *for every rewrite computation $(s_0 \rightarrow_{\mathcal{R}'} \ldots \rightarrow_{\mathcal{R}'} s_n)$ in $\mathcal{R}'$ s.t. $s_0 \models \mathcal{A}$, a rewrite computation $(s_0 \rightarrow_{\mathcal{R}} \ldots \rightarrow_{\mathcal{R}} s_n)$ exists in $\mathcal{R}$ and $s_i \models \mathcal{A}$, $i = 0, \ldots, n$*
2. *for every rewrite computation $(s_0 \rightarrow_{\mathcal{R}} \ldots \rightarrow_{\mathcal{R}} s_n)$ in $\mathcal{R}$ s.t. $s_i \models \mathcal{A}$, $i = 0, \ldots, n$, a rewrite computation $(s_0 \rightarrow_{\mathcal{R}'} \ldots \rightarrow_{\mathcal{R}'} s_n)$ exists in $\mathcal{R}'$.*

Roughly speaking, Definition 5.3 states that (1) every computation in $\mathcal{R}'$ whose initial state satisfies $\mathcal{A}$ is a safe computation w.r.t. $\mathcal{A}$ in $\mathcal{R}$, and (2) any safe computation w.r.t. $\mathcal{A}$ in $\mathcal{R}$ is also reproducible in the corrected theory $\mathcal{R}'$.

In topmost rewrite theories, all rewrite steps on system states happen at the top of the term. This implies that each rewrite step $s_1 \overset{r,\sigma,w}{\rightarrow}_{R,B} s_2$, with rule $r = (\lambda \Rightarrow \rho$ if $C)$, yields a state s_2 that is an *instance* of the right-hand side ρ of the applied rule r. Therefore, assertion violations in s_2 can only

occur in those terms that have been introduced by the instantiated right-hand side $\rho\sigma$. This suggests to us the idea that a correction refinement $\mathcal{R}'$ of the topmost rewrite theory $\mathcal{R}$ w.r.t. $\mathcal{A}$ can be synthesized by simply adding an extra constraint $\mathsf{Ren}(\rho) =/= \mathsf{fail}$ to the condition C of every rewrite rule $r = (\lambda \Rightarrow \rho \text{ if } C) \in \mathcal{R}$. Roughly speaking, the extra constraint guarantees that no assertion violation can occur in (any instance of) the right-hand side ρ of r by checking that (the corresponding renamed instances of) ρ cannot be equationally simplified to the constant fail in the $\mathcal{A}$-extension of the equational theory $\mathcal{E}$. This ensures that any state that can be obtained by applying the transformed rules of $\mathcal{R}'$ satisfies all of the assertions in $\mathcal{A}$.

Now, we are ready to formalize our correction transformation for topmost rewrite theories w.r.t. an assertional specification $\mathcal{A}$.

Definition 5.4 ($\mathcal{A}$-extension of $\mathcal{R}$) *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a topmost rewrite theory, $\mathcal{A}$ be an assertion set, and $(\Sigma^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B)$ be the $\mathcal{A}$-extension of the equational theory $(\Sigma, \Delta \cup B)$. The* $\mathcal{A}$-extension of $\mathcal{R}$ is defined as the rewrite theory $(\Sigma^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B, R^{\mathcal{A}})$, *where*

$$R^{\mathcal{A}} = \{\lambda \Rightarrow \rho \;\; \textit{if} \;\; C \wedge \mathsf{Ren}(\rho) =/= \mathsf{fail} \mid (\lambda \Rightarrow \rho \;\; \textit{if} \;\; C) \in R\}.$$

The following result states that, even if corrections can change the control flow of the program, they do not introduce new states because of their equational definition. Hence the traces $(s_0 \rightarrow_{\mathcal{R}} \ldots \rightarrow_{\mathcal{R}} s_n)$ in $\mathcal{R}$ and $(s_0 \rightarrow_{\mathcal{R}'} \ldots \rightarrow_{\mathcal{R}'} s_n)$ in $\mathcal{R}'$ specify the same state sequence.

Proposition 5.5 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a topmost rewrite theory, $\mathcal{A}$ be an assertion set, and $\mathcal{R}'$ be the $\mathcal{A}$-extension of $\mathcal{R}$ formalized in Definition 5.4. Then $\mathcal{R}' \leq^{\mathcal{A}} \mathcal{R}$.*

Example 5.6

Consider the topmost rewrite theory $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ encoded by the `CARGO` system module of Example 2.1, and the assertion set $\mathcal{A}$ of Example 3.1. Then the $\mathcal{A}$-extension of $\mathcal{R}$ w.r.t. $\mathcal{A}$ is the rewrite theory $\mathcal{R}' = (\Sigma^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B, R^{\mathcal{A}})$ where

- $(\Sigma^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B)$ is the $\mathcal{A}$-extension of $(\Sigma, \Delta \cup B)$ that has been computed in Example 4.8;
- $R^{\mathcal{A}}$ is the set that contains the following rewrite rules

```
crl [stow'] :      < MAXW,MAXS | CG > FL : c(W),CG1 =>
                   < MAXW,MAXS | CG,c(W) > FL : CG1
                   if weight(CG,c(W)) <= MAXW /\
                      Ren(< MAXW,MAXS | CG,c(W) > FL : CG1) =/= fail.

crl [unstow'] :    < MAXW,MAXS | c(W),CG > FL : CG1 =>
                   < MAXW,MAXS | CG > FL : CG1,c(W)
                   if Ren(< MAXW,MAXS | CG > FL : CG1,c(W)) =/= fail.

crl [load'] :      < MAXW,MAXS | CG > FL : CG1,c(W),CG2 =>
                   < MAXW,MAXS | CG > FL : CG1,c(W + 1),CG2
                   if not(isFull(c(W))/\
                      Ren(< MAXW,MAXS | CG > FL : CG1,c(W + 1),CG2) =/= fail.

crl [unload'] :    < MAXW,MAXS | CG > FL : CG1,c(W),CG2 =>
                   < MAXW,MAXS | CG > FL : CG1,c(W - 1),CG2
                   if Ren(< MAXW,MAXS | CG > FL : CG1,c(W - 1),CG2 ) =/= fail.
```

By Proposition 5.5, $\mathcal{R}' \leq^{\mathcal{A}} \mathcal{R}$, which implies that $\mathcal{R}'$ reproduces all and only the computations of $\mathcal{R}$ that are safe w.r.t. $\mathcal{A}$. For instance, the following computation step

$$\texttt{< 10,3 | c(5) > : c(0)} \rightarrow_{\mathcal{R}} \texttt{< 10,3 | c(5) > : c(-1)}$$

can be given in $\mathcal{R}$ by applying the `unload` rule, thereby yielding a resulting state $s_{err} =$ `< 10,3 | c(5) > : c(-1)` that violates the assertion `a1`.

Now, observe that there is no way to reach the state s_{err} from the initial system state $s_i =$ `< 10,3 | c(5) > : c(0) >` in $\mathcal{R}'$ because the transformed rule `unload'` cannot be applied to s_i. In fact, the instantiated guard

$$\texttt{ren(< 10,3 | c(5) > : c(0 - 1)) =/= fail}$$

in the conditional part of `unload'` evaluates to `false` and thus prevents the rule `unload'` from being fired.

Finally, we would like to point out that, in addition to preserve termination, confluence and sort-decreasingnes of E, our correction transformation preserves the coherence of R w.r.t. E when applied to topmost rewrite theories that do not contain (conditional) critical pairs, and additionally, meet the sufficient conditions of [7] for enforcing coherence in a conditional rewriting setting. Roughly speaking, the rewrite theory $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ is *coherent* if, for each term t to which both an equation of Δ and a rule of R can

be independently applied modulo B, the two independent derivations finally meet two terms, w and w', such that $w =_B w'$; formally, if $t \to_{\Delta,B} u$ and $t \to_{R,B} v$, then there exist u', u'', w and w' such that the diagram of Figure 1 commutes[8].

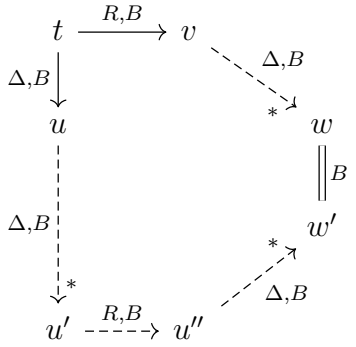

Figure 1: Commutative diagram for the coherence property.

Following [7], there are two scenarios in which coherence might be lost on a given term t because the two independent derivations stemming from t cannot be finally joined: the *overlap* scenario in which there exists at least a nonjoinable critical pair between R and Δ and the *nonoverlap* scenario that happens when a rewrite rule of R is applicable to a position of t that is under the position of a redex of t w.r.t. Δ. The class of rewrite theories that we consider excludes both scenarios.

Firstly, the renaming mechanism, encoded in the correction transformation, ensures that the new equations added to the fixed program $\mathcal{R}'$ cannot generate critical pairs —indeed, the left-hand sides of the rules in $\mathcal{R}'$ are built using the original symbols of $\mathcal{R}$, while the left-hand sides of the new, added equations are built using the primed version of the original symbols, hence no overlap is possible.

Secondly, the nonoverlap scenario is not a concern, since it cannot occur due to the topmost nature of the considered rewrite theories.

[8]We follow the usual diagrammatic convention, where dashed lines indicate existential quantification.

6. Assertion-driven Correction for more Complex Rewrite Theories

In this section, we enlarge the class of rewrite theories that can be automatically corrected so that no assertion in $\mathcal{A}$ is contravened. This is done by considering two classes of rewrite theories that can be transformed into (semantically equivalent) topmost rewrite theories, and then automatically corrected w.r.t. $\mathcal{A}$ by exploiting Proposition 5.5.

The first class we consider generalizes the topmost *modulo ACU* rewrite theories of [12] to topmost *modulo Ax* rewrite theories, where Ax consists of any of the combinations of axioms ACU, AC, AU, or A for a given binary symbol $\otimes$: *Config Config* $\rightarrow$ *Config* of the signature. The operator $\otimes$ is used to build *system configurations* that obey the *structural axioms* of $\otimes$ given by Ax. The second class we consider formalizes the so-called Russian doll rewrite theories, which allow one to deal with complex, recursively nested state configurations (e.g., multisets of multisets of elements) in which rewrites can happen at any nesting depth of a state.

Topmost *modulo Ax* rewrite theories have many practical applications as they support system configurations that can consist of multisets (defined by the symbol $\otimes$ being a binary ACU/AC operator), or lists (defined by $\otimes$ being AU/A). This class of theories is particularly useful in the specification of object-oriented systems involving flat configurations in which the distributed state is a (multi-)set or a list of objects and messages [12]. Furthermore, different styles of Petri nets can also be modeled via topmost modulo Ax rewrite theories [14].

Note that our formalization of topmost *modulo Ax* rewrite theories excludes the cases when Ax contains the combination CU, or simply C. The reason is twofold. On the one hand, an operator $\otimes$ that obeys CU/C would model states as *recursively, nested* commutative pairs of terms. This state structure is tricky and generally of little use since it can be replaced by simpler state structures that exploit the more powerful ACU/AC/AU operators in most practical scenarios. On the other hand, if we just need to model states as *flat* commutative pairs $t_1 \otimes t_2$ (that is, pairs in which the operator $\otimes$ cannot occur either in t_1 or t_2), this can be done by defining a topmost rewrite theory in which $\otimes$ is a binary CU (or C) operator with sort $s\ s \rightarrow State$. This way, there is no need to apply any program transformation, since all states of the form $t_1 \otimes t_2$ are rewritten at the top level and the correction technique of Section 5 can be directly applied.

6.1. Topmost modulo Ax Rewrite Theories

Topmost *modulo Ax* rewrite theories can be formalized as follows. We denote by $\alpha(f)$ the set $\alpha(f) = \{ACU,\ AC,\ AU,\ A\}$ of combinations of associativity, commutativity, and/or unity axioms for the binary operator $f \in \Sigma$.

Definition 6.1 (Topmost *modulo* Ax rewrite theory) *Let* $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ *be a rewrite theory with a finite poset of sorts* $(S, <)$*. Let Config be a top sort in* S*, and* $\otimes$: *Config Config* $\rightarrow$ *Config* $\in \Sigma$ *be a binary operator that obeys a combination of associativity, commutativity, and/or identity axioms* $Ax \in \alpha(\otimes) \subseteq B$*.*

The theory $\mathcal{R}$ *is said to be topmost* modulo Ax *if*

1. *for each rule* $(\lambda \Rightarrow \rho$ *if* $C)$ *in* R*,* λ *and* ρ *are terms of sort Config;*
2. $\otimes$ *is the only operator in* Σ *whose arity includes a sort* s *such that* $top(s) = Config$*.*

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

Unlike topmost theories that globally rewrite a state at each rewrite step; topmost modulo Ax theories allow rewrite rules to be applied to *system configuration* fragments —thereby implementing local state changes and providing more flexibility and conciseness in theory specification.

Example 6.2

Let us consider the following Maude code fragment that specifies the `FlatCargo` data structure as a (flat) multiset of containers $\mathtt{c}(\mathtt{W}_1), \mathtt{c}(\mathtt{W}_2), \ldots, \mathtt{c}(\mathtt{W}_n)$.

```
sorts Container FlatCargo .
subsort Container < FlatCargo .
op c : Int -> Container .
op nil : -> FlatCargo .
op _,_ : FlatCargo FlatCargo -> FlatCargo [ctor assoc comm id: nil] .
var W : Int .

rl [load] :  c(W)  => c(W + 1) .
rl [unload] : c(W)  => c(W - 1) .
```

It is immediate to see that this code fragment encodes a topmost modulo ACU rewrite theory. It suffices to interpret the sort `FlatCargo` as the sort *Config* of Definition 6.1, and the ACU operator

```
op _,_ : FlatCargo FlatCargo -> FlatCargo [ctor assoc comm id: nil]
```

as the operator $\otimes : \mathit{Config}\ \mathit{Config} \to \mathit{Config}$. Note that the rule `load` (respectively, `unload`) allows the weight of an arbitrary container to be *locally* increased (respectively, decreased) within a cargo configuration $\mathtt{c}(\mathtt{W}_1), \ldots, \mathtt{c}(\mathtt{W}_n)$.

Unfortunately, our correction technique cannot be directly applied to this class of rewrite theories because the assertion checking mechanism, which is encoded in the corrected rewrite rules, could fail to catch some assertion violations when local state changes are performed. This is because such rules only check assertions within the rule contractum while ignoring the rest of the configuration (that is, the context at which the replacement takes place). Let us see an example.

Example 6.3

Consider again the Maude code fragment of Example 6.2 and an assertion set $\mathcal{A}$ that consists of a single assertion that enforces the containers in every system configuration to be pairwise distinct:

$$\mathtt{c(W1)},\mathtt{c(W2)}\ \mid\ \mathtt{W1}\ \mathtt{=/=}\ \mathtt{W2} \tag{1}$$

Now, if we apply the correction technique of Section 5, the following assertion-checking equation is synthesized

$$\mathtt{c}'(\mathtt{W1}),'\,\mathtt{c}'(\mathtt{W2}) = \mathsf{fail}\ \ \mathsf{if}\ \mathtt{not}(\mathsf{Ren}^{-1}(\mathtt{W1}\ \mathtt{=/=}\ \mathtt{W2})) \tag{2}$$

and the `load` rewrite rule of Example 6.2 is refined into the conditional rule

$$\mathtt{crl\ [load'] :\ \ c(W) => c(W + 1)\ if}\ \mathsf{Ren}(\mathtt{c(W + 1)})\ \mathtt{=/=}\ \mathsf{fail}. \tag{3}$$

that only increments the container weight provided the condition

$$\mathsf{Ren}(\mathtt{c(W + 1)})\ \mathtt{=/=}\ \mathsf{fail} \tag{4}$$

is satisfied.

Unfortunately, including Condition (4) into the definition of `load'` is not enough for the transformed theory to be correct w.r.t. $\mathcal{A}$. In fact, in such a naïvely transformed theory, the following undesired rewrite step can be given by applying `load'`

$$\mathtt{c(0)},\mathtt{c(1)},\mathtt{c(2)} \overset{\mathtt{load'}}{\to} \mathtt{c(0 + 1)},\mathtt{c(1)},\mathtt{c(2)}$$

that is further simplified into the canonical form `c(1),c(1),c(2)` which violates Assertion (1). This happens because Assertion (1) is only locally checked over (a renamed version of) the contractum `c(0 + 1)` and not against the whole system configuration `c(0 + 1),c(1),c(2)`.

The applicability problem revealed by Example 6.2 can be overcome by transforming a rewrite theory $\mathcal{R}$ that is topmost modulo Ax into an equivalent topmost theory $\hat{\mathcal{R}}$ to which our correction technique applies. Such a transformation was formerly studied in [12] for the case when only ACU operators are considered. Here we extend it with the combinations AC, AU and A.

Definition 6.4 (topmost extension of $\mathcal{R}$) *Let $\mathcal{R} = (\Sigma, E, R)$ be a topmost modulo Ax rewrite theory, where $E = \Delta \cup B$ and $Ax \in B$. Let X, X_1, and X_2 be variables of sort Config not occurring in either R or E. We define the topmost rewrite theory $\hat{\mathcal{R}} = (\hat{\Sigma}, E, \hat{R})$ where $\hat{\Sigma}$ extends Σ by adding a new top sort State, and a new operator $\{\_\}$: Config $\rightarrow$ State; and $\hat{R}$ is obtained by transforming R according to Ax as follows.*
For each $(\lambda \Rightarrow \rho \text{ if } C) \in R$

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

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

Proposition 6.5 *Let $\mathcal{R}$ be a topmost modulo Ax theory and $\hat{\mathcal{R}}$ be the topmost extension of $\mathcal{R}$. For any term t_i and t_f of sort Config, $t_i \rightarrow_{\mathcal{R}}^* t_f$ iff $\{t_i\} \rightarrow_{\hat{\mathcal{R}}}^* \{t_f\}$.*

Example 6.6

Consider the topmost modulo ACU rewrite theory $\mathcal{R}$ that is specified by the Maude code fragment of Example 6.2. By computing its topmost extension $\hat{\mathcal{R}}$, the rewrite rules `load` and `unload` are transformed into the rules

```
rl [load-ACU] :  { X,c(W) }  => { X,c(W + 1) } .
rl [unload-ACU] :  { X,c(W) }  => { X,c(W - 1) } .
```

Now, the undesired computation of Example 6.2 is mimicked in $\hat{\mathcal{R}}$ by the following computation

$$\texttt{\{c(0),c(1),c(2)\}} \rightarrow^*_{\hat{\mathcal{R}}} \texttt{\{c(1),c(1),c(2)\}}$$

in which the `load-ACU` rule is applied to the initial state `{c(0),c(1),c(2)}` to erroneously increase the weight of container `c(0)`.

By exploiting the program transformation for topmost rewrite theories of Section 5, the assertion-driven correction technique can also be applied to the class of topmost modulo Ax rewrite theories, and its correction follows from the correction result for the topmost theories (see Proposition 5.5).

Corollary 6.7 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a topmost modulo Ax rewrite theory, with $Ax \in B$. Let $\mathcal{A}$ be an assertion set, $\hat{\mathcal{R}} = (\hat{\Sigma}, \Delta \cup B, \hat{R})$ be the topmost extension of $\mathcal{R}$, and $(\hat{\Sigma}^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B)$ be the $\mathcal{A}$-extension of $(\hat{\Sigma}, \Delta \cup B)$.*

Let $\hat{\mathcal{R}}' = (\hat{\Sigma}^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B, \hat{R}^{\mathcal{A}})$ be a rewrite theory such that

$$\hat{R}^{\mathcal{A}} = \{\lambda \Rightarrow \rho \,\mathsf{if}\, C \wedge \mathsf{Ren}(\rho) =/= \mathsf{fail} \mid (\lambda \Rightarrow \rho \,\mathsf{if}\, C) \in \hat{R}\}.$$

Then $\hat{\mathcal{R}}' \leq^{\mathcal{A}} \hat{\mathcal{R}}$.

Example 6.8

Consider the assertion set $\mathcal{A} = \{\texttt{c(W1),c(W2) | W1 =/= W2}\}$ of Example 6.3 and the rewrite theory $\mathcal{R}$ of Example 6.2, together with the topmost extension $\hat{\mathcal{R}}$ of $\mathcal{R}$ given in Example 6.6. Then, $\hat{\mathcal{R}}' = (\hat{\Sigma}^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B, \hat{R}^{\mathcal{A}})$ includes the conditional rule

```
crl [loadC-ACU'] : { X,c(W) } => { X,c(W + 1) }
                                    if Ren({X,c(W + 1)}) =/= fail .
```

Note that the application of `[loadC-ACU']` completely replaces a multiset M of containers (that matches `{ X,c(W) }`) with a new one M' (that matches `{ X,c(W + 1) }`). The rule is fired only if the resulting multiset M' satisfies the condition `Ren( X,c(W + 1) ) =/= fail`, which is true if no violation of the assertion in $\mathcal{A}$ is detected (that is, the containers in M' are pairwise disjoint).

Now, as expected, the erroneous computation of Example 6.6, which violates $\mathcal{A}$, cannot be reproduced in $\hat{\mathcal{R}}'$.

It is worth noting that the program transformation above can be easily extended to those rewrite theories that include local as well as global state changes, which are respectively modeled by topmost and topmost modulo Ax rewrite rules. In this scenario, given a rewrite theory $\mathcal{R} = (\Sigma, \Delta \cup B, R)$, we are always able to partition the set of rewrite rules R into two disjoint sets R_Λ and $R_{>\Lambda}$ such that $R_\Lambda = \{\lambda \Rightarrow \rho \text{ if } C \mid \lambda, \rho \text{ are of sort } \mathit{State}\}$ and $R_{>\Lambda} = \{\lambda \Rightarrow \rho \text{ if } C \mid \lambda, \rho \text{ are of sort } \mathit{Config}\}$. The set R_Λ contains the topmost rewrite rules in R that globally rewrite a state at its root position, while $R_{>\Lambda}$ includes the topmost modulo Ax rules that locally rewrite an inner state fragment (that is, a configuration within the state).

Now, $\mathcal{R}$ can be turned into a topmost rewrite theory by reusing the sort *State* included in $\mathcal{R}$ and simply applying the program transformation of Definition 6.4 to $R_{>\Lambda}$, while leaving R_Λ unchanged.

Example 6.9

Let $\mathcal{R} = (\Sigma, \Delta \cup B, R_\Lambda \cup R_{>\Lambda})$ be a rewrite theory such that Σ includes the operators $\{\_\} : \texttt{Config} \to \texttt{State}$, $\texttt{a} :\to \texttt{Config}$, $\texttt{c} :\to \texttt{Config}$, as well as the ACU operator $\otimes : \texttt{Config}\ \texttt{Config} \to \texttt{Config}$, and

$$\begin{aligned} R_\Lambda &= \{[\texttt{r}_1] : \{\texttt{a} \otimes \texttt{X}\} \Rightarrow \{\texttt{X}\}\} \\ R_{>\Lambda} &= \{[\texttt{r}_2] : \texttt{a} \Rightarrow \texttt{c}\} \end{aligned}$$

where $\texttt{X}$ is a variable of sort $\texttt{Config}$.

Note that the rule $\texttt{r}_1 \in R_\Lambda$ globally rewrites terms of the form $\{t_1 \otimes \ldots \otimes t_n\}$ of sort $\texttt{State}$, while $\texttt{r}_2 \in R_\Lambda$ performs a local state change, that is, it allows a state fragment, namely, the configuration $\texttt{a}$, to be rewritten into the configuration $\texttt{c}$.

Now, by applying the transformation of Definition 6.4 to $\texttt{r}_2$, we get the topmost rule $[\texttt{r}_3] : \{\texttt{X} \otimes \texttt{a}\} \Rightarrow \{\texttt{X} \otimes \texttt{c}\}$, where $\texttt{X}$ is a variable of sort $\texttt{Config}$, and the rewrite theory $(\Sigma, \Delta \cup B, \{\texttt{r}_1, \texttt{r}_3\})$ is thus a topmost rewrite theory (equivalent to $\mathcal{R}$) to which the correction methodology can be applied.

6.2. Russian Doll Rewrite Theories

Many systems (e.g., distributed object-based systems) can have a complex state structure in which system configuration components (e.g. objects and messages) can themselves contain nested configurations of components (e.g., subobjects and submessages). Typically, these configurations are specified in Maude by means of a nested and recursive multiset structure.

Unfortunately, topmost modulo Ax rewrite theories of Section 6.1 can only deal with flat configurations, and thus cannot be used to model rewrite theories whose states have an inherently nested structure.

The class of *Russian doll* rewrite theories, originally introduced in [15], generalizes the class of topmost modulo ACU rewrite theories, and precisely captures the nature of recursively nested state structures. Roughly speaking, in a Russian doll rewrite theory, the nested state structuring is specified by a boundary operator[9] of the form $b\colon s_1 \dots s_n\ \mathit{Config} \to \mathit{Config}$, $n \geq 0$, which allows a configuration of sort *Config* to be encapsulated in a well-delimited structure. This structure may also include additional parameters of sorts $s_1, \dots, s_n$ that may be convenient to better describe system configurations. Then, a state *st* can contain several nested configurations, each of which is wrapped by means of the boundary operator b.

The assertion-driven program correction technique of previous sections cannot be directly applied to Russian doll rewrite theories since rewrites in a state can happen at any level of nesting. Nonetheless, analogously to the case of topmost modulo Ax theories, we can transform Russian doll theories into equivalent, topmost theories for which corrections w.r.t. $\mathcal{A}$ can be computed. This is essentially done by adapting the program transformation of [12], which is correct under the reasonable assumptions that equations do not change the depth of nesting of configurations and rewrite rules do not increase it.

Following [12], the formalization of the class of the Russian doll theories requires the following auxiliary definitions.

Definition 6.10 *Let Σ be a signature, whose set of sorts is S. We say that Σ is a* Russian doll *signature if*

- *S includes the sorts Config and FlatConfig, with FlatConfig $<$ Config, and Config is a top sort in $S/\equiv_<$. Furthermore, for each sort $s \in S$, $s <$ Config implies $s <$ FlatConfig.*
- *The only operators in Σ whose arity includes a sort s such that $top(s) =$ Config are:*

$$\_ \otimes \_ : \mathit{FlatConfig}\ \ \mathit{FlatConfig} \to \mathit{FlatConfig}$$

[9] To keep the exposition simple, here we consider a single boundary operator b. However, as described in [15], multiple boundary operators of the form $b_j\colon \vec{s_j}\ \mathit{Config} \to \mathit{Config}, j = 1, \dots, m$ could be specified, each of which has distinct argument sorts $\vec{s_j}$.

$$\_ \otimes \_ : \mathit{Config}\ \mathit{Config} \to \mathit{Config}$$
$$b : \vec{s}\ \mathit{Config} \to \mathit{Config}$$

where $\otimes$ obeys the algebraic axioms ACU. Furthermore, for each operator $f : \vec{w} \to \mathit{Config} \in \Sigma$, f is either $\otimes$ or b.

Definition 6.11 *Given a Russian doll signature Σ, a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is of* bounded nesting *if, for all $x \in \mathcal{V}ar(t)$, x is of sort Config implies x is of sort FlatConfig.*

Given a term t of bounded nesting, we define the nesting depth *of t as follows:*

$$\mathit{depth}(t) = \begin{cases} 0 & t \notin \mathcal{T}(\Sigma, \mathcal{V})_{\mathit{Config}} \text{ or } t \in \mathcal{T}(\Sigma, \mathcal{V})_{\mathit{FlatConfig}} \\ \mathit{max}(\mathit{depth}(t_1), \mathit{depth}(t_2)) & t = t_1 \otimes t_2 \\ \mathit{depth}(t) + 1 & t = b(\vec{p}, t_1) \end{cases}$$

Now, a Russian doll rewrite theory is formally defined as follows.

Definition 6.12 (Russian doll rewrite theory) *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a rewrite theory. Then, $\mathcal{R}$ is a* Russian doll *rewrite theory if*

1. *Σ is a Russian doll signature;*
2. *for each equation $(\lambda = \rho \text{ if } C) \in \Delta$ and substitution σ, $\sigma(\lambda)$ is of bounded nesting iff $\sigma(\rho)$ is of bounded nesting, and if $\sigma(\lambda)$ and $\sigma(\rho)$ are of bounded nesting, then $\mathit{depth}(\sigma(\lambda)) = \mathit{depth}(\sigma(\rho))$;*
3. *for each rewrite rule $(\lambda \Rightarrow \rho \text{ if } C) \in R$, λ and ρ are of sort Config, and for each substitution σ such that $\sigma(\lambda)$ and $\sigma(\rho)$ are of bounded nesting, $\mathit{depth}(\sigma(\lambda)) \geq \mathit{depth}(\sigma(\rho))$.*

Roughly speaking, Condition 2 in Definition 6.12 ensures that equations do not change the nesting depth of terms, while Condition 3 enforces that rewrites do not increase the nesting depth of terms. Moreover, for Russian doll theories, we have that if t_1 is a term of bounded nesting, $t_1 \to_{\mathcal{R}} t_2$ implies that t_2 is of bounded nesting and $\mathit{depth}(t_1) \geq \mathit{depth}(t_2)$.

Example 6.13

The following fragment of Maude code extends the `Cargo` data structure of Example 2.1 that models flat lists of containers $\mathtt{c}(\mathtt{W}_1), \mathtt{c}(\mathtt{W}_2), \ldots, \mathtt{c}(\mathtt{W}_n)$ by considering multisets of elements that can be either simple containers $\mathtt{c}(\mathtt{W})$ or *compound containers*. Compound containers are in turn multisets of simple or compound containers.

```
sorts Container Cargo FlatCargo .
subsort Container < FlatCargo < Cargo .

op c : Int -> Container [ctor] .
op nil : -> FlatCargo .
op _,_ : FlatCargo FlatCargo -> FlatCargo [ctor assoc id: nil] .
op _,_ : Cargo Cargo -> Cargo [ctor assoc id: nil]
op [_] : Cargo -> Cargo .

vars X1 X2 : Cargo .
var w : Int .

rl [insert] : X1,[ X2 ] => [ X1,X2 ] .
rl [load-simple] : c(w) => c(w + 1) .
```

Note that the considered code fragment represents a Russian doll rewrite theory. It suffices to interpret sorts `FlatCargo` and `Cargo` as the sorts *FlatConfig* and *Config* of Definition 6.12, and the operators

```
_,_ : FlatCargo FlatCargo -> FlatCargo [ctor assoc id: nil] .
_,_ : Cargo Cargo -> Cargo [ctor assoc id: nil] .
[_] : Cargo -> Cargo .
```

as the operators $\otimes\colon \mathit{FlatConfig}\ \mathit{FlatConfig} \to \mathit{FlatConfig}$, $\otimes\colon \mathit{Config}\ \mathit{Config} \to \mathit{Config}$, and $b\colon \vec{s}\ \mathit{Config} \to \mathit{Config}$, where the sort list $\vec{s}$ is empty. The new `Cargo` data structure allows multisets of simple and compound containers at distinct nesting levels to be specified. For instance,

```
c(4),[c(3),c(1)],[[c(4),c(5)],c(3)]
```

is a depth-2 term of sort `Cargo`.

The rewrite rules `load-simple` and `insert` specify actions for simple and compound containers, respectively. The former increases the weight of a simple container, while the latter allows simple as well as compound containers to be inserted into another compound container.

The next definition is a natural extension of the program transformation of [12] that turns a Russian doll rewrite theory $\mathcal{R}$ into a topmost rewrite theory $\mathcal{R}_n$, which is able to deal with configurations whose rule structure has a fixed nesting depth.

Definition 6.14 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a Russian doll rewrite theory, and n be a natural number. We define the* topmost n-extension *of $\mathcal{R}$ as the topmost rewrite theory $\mathcal{R}_n = (\Sigma_n, \Delta \cup B, R_n)$ where*

- *Σ_n extends Σ by adding a new top sort State, and a new operator $\{\_\}$: Config $\rightarrow$ State;*
- *for each rewrite rule $(\lambda \Rightarrow \rho \text{ if } C) \in R$ and $0 \leq k \leq n$*

$$\{C_0 \otimes b(\vec{x}_1, C_1 \otimes b(\vec{x}_1, \ldots b(\vec{x}_k, C_k \otimes \lambda)\ldots))\} \Rightarrow \{C_0 \otimes b(\vec{x}_1, C_1 \otimes b(\vec{x}_2, \ldots b(\vec{x}_k, C_k \otimes \rho)\ldots))\} \text{ if } C \in R_n$$

where C_0,C_1,..., C_k are variables of sort Config, and $\vec{x}_1, \vec{x}_2 \ldots, \vec{x}_k$ are sequences of variables of the sorts required by the boundary operator b.

Example 6.15

Consider the Russian doll rewrite theory encoded by the Maude fragment of Example 6.13. Then, its topmost 1-extension is a topmost rewrite theory that contains the following rewrite rules:

```
rl [insert-0] : { C0,X1,[ X2 ] } => { C0,[ X1,X2 ] } .
rl [insert-1] : { C0,[ C1,X1,[ X2 ] ] } => { C0,[ C1,[ X1,X2 ] ] } .
rl [load-simple-0] : { C0,c(w) } => {C0,c(w + 1) } .
rl [load-simple-1] : { C0,[ C1,c(w) ] } => {C0,[ C1,c(w + 1) ] } .
```

It is worth noting that the rewrite rules in the computed 1-extension manage insert and load operations at a nesting depth smaller than or equal to 1. Specifically, the two variants of the original `load-simple` rule allow to loading simple containers located at depth 0 or 1 within a `Cargo` structure. Likewise, the rules `insert-0` and `insert-1` transfer a multiset of containers `X1` located at depth 0 and 1, respectively, to a deeper compound container.

Equivalence between a Russian doll rewrite theory and its *n-extension* $\mathcal{R}_n$ is given w.r.t. configurations of bounded nesting with a depth equal to n, for any natural number n. This means that if t_i is a configuration of bounded nesting with $depth(t_i) = n$ in $\mathcal{R}$, then t_i can be rewritten in t_f in $\mathcal{R}$ if and only if $\{t_i\}$ can be rewritten in $\{t_f\}$ in $\mathcal{R}_n$. More formally,

Proposition 6.16 ([12]) *Let $\mathcal{R}$ be a Russian doll rewrite theory. Let t_i be a term of bounded nesting of sort Config such that $depth(t_i) = n$. Let $\mathcal{R}_n$ be the topmost n-extension of $\mathcal{R}$. Then, $t_i \to^*_{\mathcal{R}} t_f$ iff $\{t_i\} \to^*_{\mathcal{R}^n} \{t_f\}$.*

Now, since $\mathcal{R}_n$ is a topmost rewrite theory, we can directly apply our correction technique and generate sound theory corrections (in the sense of Proposition 5.5) w.r.t. $\mathcal{A}$. Indeed, the following corollary holds.

Corollary 6.17 *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a Russian doll rewrite theory. Let $\mathcal{A}$ be an assertion set, $\mathcal{R}_n = (\Sigma_n, \Delta \cup B, R_n)$ be the topmost n-extension of $\mathcal{R}$, and $(\Sigma_n^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B)$ be the $\mathcal{A}$-extension of $(\Sigma_n, \Delta \cup B)$.*

Let $\mathcal{R}'_n = (\Sigma_n^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B, R_n^{\mathcal{A}})$ be a rewrite theory such that

$$R_n^{\mathcal{A}} = \{\lambda \Rightarrow \rho \,\mathsf{if}\, C \wedge \mathsf{Ren}(\rho) =/= \mathsf{fail} \;\mid\; (\lambda \Rightarrow \rho \,\mathsf{if}\, C) \in R_n\}.$$

Then $\mathcal{R}'_n \leq^{\mathcal{A}} \mathcal{R}_n$.

In Russian doll theories, configurations are specified by means of the ACU operator $\otimes$ that allows (nested) multisets of elements to be composed together. Actually, it would be possible to consider variants of $\otimes$ that obey distinct combinations of algebraic axioms such as AC, AU, or A, similarly to the case of topmost modulo Ax theories of Section 6.1. Nonetheless, this is typically not practical for correction purposes since the number of rewrite rules in the computed topmost n-extension $\mathcal{R}_n$ could become intractable even for small values of n. Indeed, a linear increment of the nesting depth n yields an exponential growth of the number of rewrite rules in $\mathcal{R}_n$. Let us see an example.

Example 6.18

Let $\mathcal{R}$ be a Russian doll rewrite theory that includes the rewrite rule $[\mathtt{r}] : \mathtt{a} \Rightarrow \mathtt{c}$. Further, for the sake of readability, we consider a simple boundary operator $[\_]$: *Config* $\to$ *Config* that encapsulates configurations without additional parameters.

Now, by Definition 6.14, the topmost 1-extension $\mathcal{R}_1$ of $\mathcal{R}$ contains the following two rules that mimick $\mathtt{r}$ within $\mathcal{R}_1$:

$$[\mathtt{r}_1] : \{\mathtt{C}_0 \otimes \mathtt{a}\} \Rightarrow \{\mathtt{C}_0 \otimes \mathtt{c}\}$$
$$[\mathtt{r}_2] : \{\mathtt{C}_0 \otimes [\mathtt{C}_1 \otimes \mathtt{a}]\} \Rightarrow \{\mathtt{C}_0 \otimes [\mathtt{C}_1 \otimes \mathtt{c}]\}$$

Now, suppose that $\otimes$ is AC instead of being ACU. In this case, the algebraic axiom U must be explicitly modeled in the 1-extension $\mathcal{R}_1$ by defining distinct rule patterns that consider the presence of both context variables C_0 and C_1, as well as the absence of one or both context variables. Thus, $\mathcal{R}_1$ must contain the following rules:

$$
\begin{aligned}
&[\mathtt{r}_1] : \{\mathtt{C}_0 \otimes \mathtt{a}\} \Rightarrow \{\mathtt{C}_0 \otimes \mathtt{c}\} \\
&[\mathtt{r}_2] : \{\mathtt{C}_0 \otimes [\mathtt{C}_1 \otimes \mathtt{a}]\} \Rightarrow \{\mathtt{C}_0 \otimes [\mathtt{C}_1 \otimes \mathtt{c}]\} \\
&[\mathtt{r}_3] : \{\mathtt{a}\} \Rightarrow \{\mathtt{c}\} \\
&[\mathtt{r}_4] : \{[\mathtt{a}]\} \Rightarrow \{[\mathtt{c}]\} \\
&[\mathtt{r}_5] : \{\mathtt{C}_0 \otimes [\mathtt{a}]\} \Rightarrow \{\mathtt{C}_0 \otimes [\mathtt{c}]\} \\
&[\mathtt{r}_6] : \{[\mathtt{C}_1 \otimes \mathtt{a}]\} \Rightarrow \{[\mathtt{C}_1 \otimes \mathtt{c}]\}
\end{aligned}
$$

Therefore, we need 6 rules in $\mathcal{R}_1$ to mimick the behavior of $\mathtt{r}$ in the case when $\otimes$ is AC.

---

7. Empirical evaluation

The program correction methodology defined in this paper has been efficiently implemented in a Maude tool called ÁTAME (*Assertion-based Theory Amendment in MaudE*). The tool has been implemented in Maude itself by using Maude's meta-level capabilities. ÁTAME integrates a RESTful Web service that is written in Java, and an intuitive Web user interface that is based on AJAX technology and is written in HTML5 canvas and Javascript. The implementation contains about 600 lines of Maude source code, 600 lines of C++ code, 750 lines of Java code, and 700 lines of HTML5 and JavaScript code. The correction tool ÁTAME is publicly available together with a number of examples at `http://safe-tools.dsic.upv.es/atame`.

In this section, we first illustrate the correction for the container terminal model of Section 2 that can be automatically synthesized by using ÁTAME. Then, we summarize the experimental results that we obtain on a set of representative benchmarks.

7.1. ÁTAME *to the Rescue: a Typical Repair Session*

Let us show how ÁTAME works in practice by showing a repair session for our container terminal specification of Example 2.1.

Maude programs can be either uploaded in ÁTAME as simple `.maude` module files or written from scratch inside a dedicated edit box. A collection of Maude programs, which includes the `CONTAINER-TERMINAL` module of Example 2.1, is provided with the tool for demonstration purposes.

To start the repair session, we can just select the `CONTAINER-TERMINAL` module under examination from the preloaded programs (see Figure 2) and proceed through the next steps.

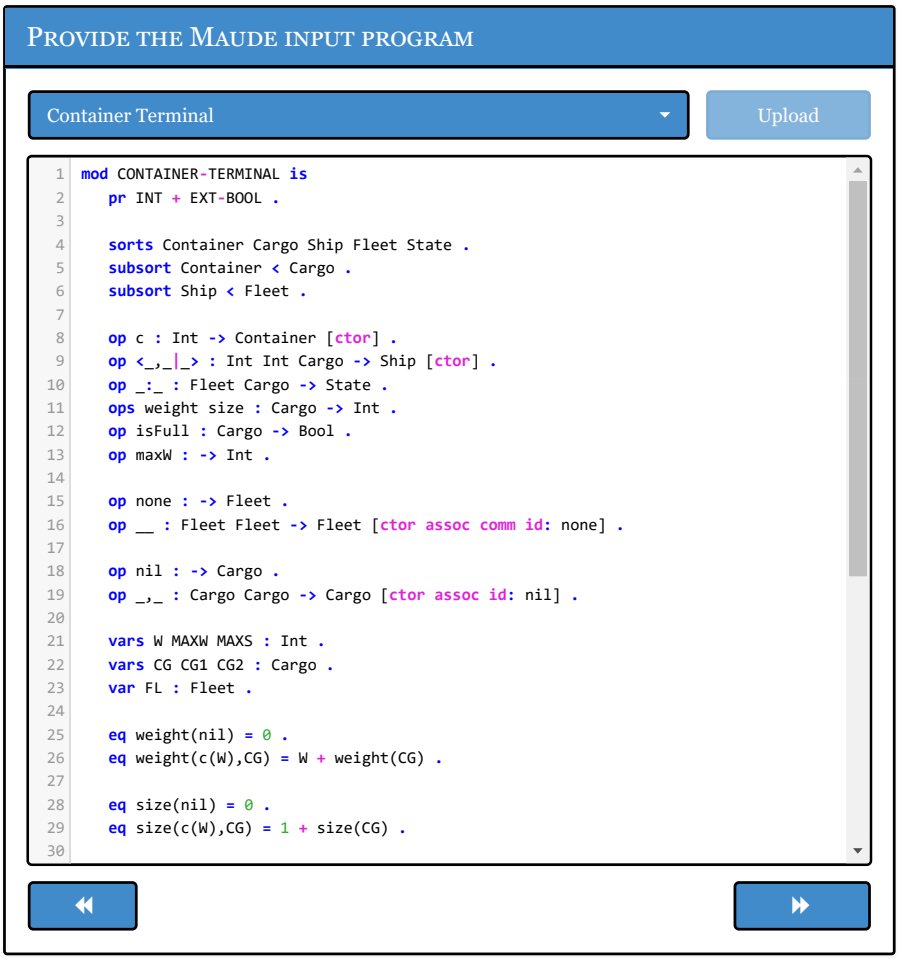

Figure 2: `CONTAINER-TERMINAL` module loaded in ÁTAME.

The next phase allows the user to specify safety properties to be enforced on the input program. These properties are modeled as system assertions

which may use logic predicates that are already defined in the program or new ones that are specified at this stage. Figure 3 illustrates the input phase of the system assertions of Example 3.1 in ÁTAME. Finally, by pressing

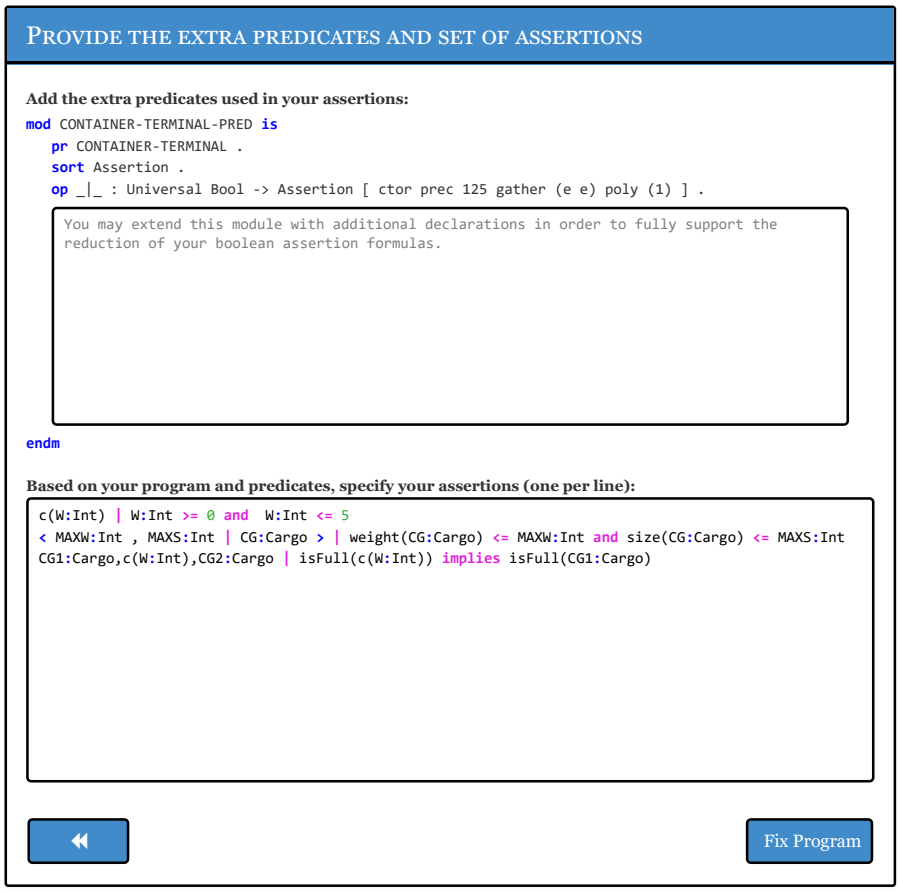

Figure 3: Input of the system assertions of Example 3.1 for the `CONTAINER-TERMINAL` module.

the Fix Program button, we run our static program repair procedure that automatically yields a corrected version of the input program in which all computations are safe w.r.t. the considered assertions. Figure 4 shows (a fragment of) the correction generated for the `CONTAINER-TERMINAL` module (i.e., the `CONTAINER-TERMINAL-FIX` module).

As an additional feature, ÁTAME provides the interconnection with the ANIMA Maude stepper [16], which integrates program animation capabilities

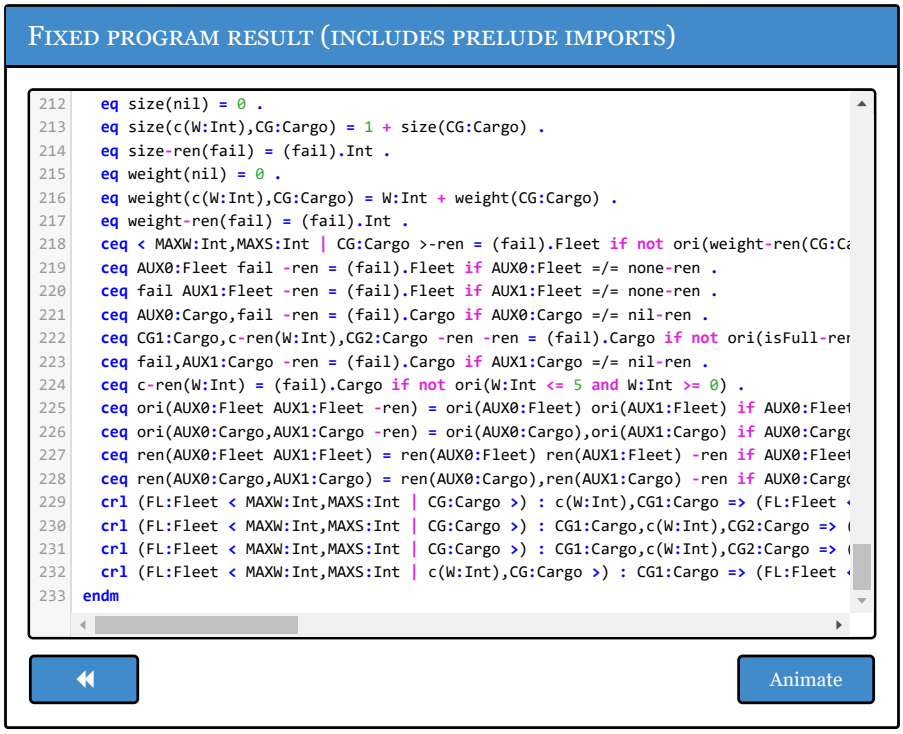

Figure 4: Correction for the `CONTAINER-TERMINAL` module.

[17] into ÁTAME. Specifically, by means of the Animate button, the user is allowed to interactively inspect an incrementally generated fragment of the computation tree[10] of the corrected program for a given initial state.

Figure 5 respectively shows a fragment of the computation tree of the `CONTAINER-TERMINAL` module (figure above) and its counterpart in the repaired module `CONTAINER-TERMINAL-FIX` (figure below). Note that state s_{39} in the computation tree of the original program does not belong to the computation tree of the repaired module. This is correct because s_{39} violates the specified assertion

```
CG1,c(W),CG2 | isFull(c(W)) implies isFull(CG1)
```

and thus it cannot appear in any computation of `CONTAINER-TERMINAL-FIX`.

[10]The computation tree of a Maude program M for a given initial state s subsumes all of the possible computations of M stemming from s.

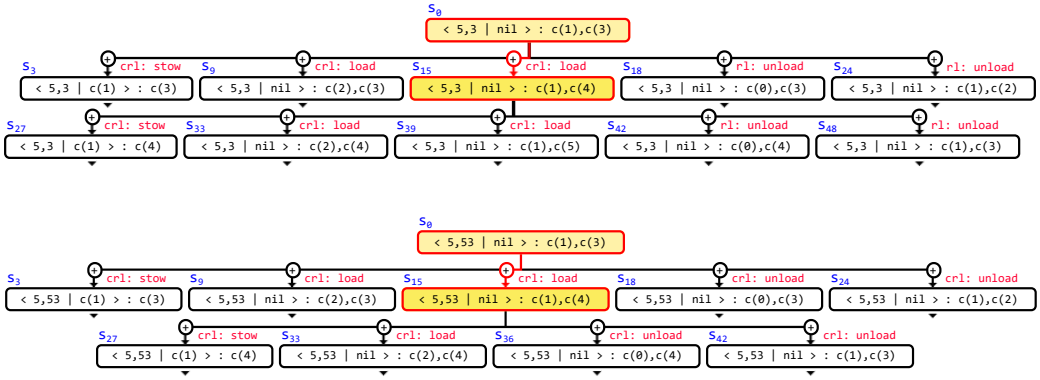

Figure 5: (Above) Fragment of the computation tree of the `CONTAINER-TERMINAL` module. (Below) Fragment of the computation tree of the `CONTAINER-TERMINAL-FIX` module.

7.2. Experimental Results

In order to evaluate the performance of the ÁTAME system, we endowed several Maude programs with system assertions, and we used the system to correct the programs w.r.t. the assertions. In all cases, the system assertions and program faults chosen are representative of typical deviations found in Maude programs. We benchmarked ÁTAME on the following collection of Maude programs, which are all available and fully described within the ÁTAME Web platform: *Bank model*, a conditional Maude specification that models a distributed banking system; *Blocks World*, a Maude encoding of the classical AI planning problem that consists of setting one or more vertical stacks of blocks on a table using a robotic arm; *BRP*, a Maude implementation of the Bounded Retransmission Protocol; *Container*, the Maude specification that models the cargo manipulation in a container terminal of the running Example 2.1; *Crossing River*, a Maude program that solves the well-known crossing river puzzle; *Dekker*, a Maude specification of Dekker's mutual exclusion algorithm; *Maude NPA*, an analysis tool for cryptographic protocols that takes into account the algebraic properties of cryptosystems; *Philosophers*, a Maude specification of the classical Dijkstra concurrency example; *Semaphore*, a classical mutual exclusion protocol with semaphores written in Maude; *Stock Exchange*, a simplified stock exchange concurrent system in which traders operate on stocks via limit orders; *Webmail app*, a

Maude specification of a rich webmail application that provides typical email management, system administration capabilities, login/logout functionality, etc; *Wolfram's Rule 30*, a one-dimensional binary cellular automaton rule introduced by Stephen Wolfram.

	$\#Ops$		$\#Eqs$								
	$\mathcal{R}$	$\mathcal{R}_{fix}$	$\mathcal{R}$	$\mathcal{R}_{fix}$	TT	$T_{\mathcal{R}}$	$T_{\mathcal{R}}^{Chk}$	$T_{\mathcal{R}_{fix}}$	S	O_{Chk}	O_{fix}
Bank Model	112	120	38	246	5	17	101	26	0.74	4.94	0.53
Blocks World	89	104	12	194	3	19	37	31	0.16	0.95	0.63
BRP	23	26	12	27	2	5	23	7	0.70	3.6	0.4
Container	90	104	20	208	5	14	80	19	0.76	4.71	0.36
Crossing River	20	33	14	58	1	6	20	7	0.65	2.33	0.17
Dekker	126	161	25	307	7	40	98	51	0.48	1.45	0.28
Maude NPA	46	75	11	128	3	33	71	36	0.50	1.15	0.09
Philosophers	51	64	12	122	3	12	36	15	0.59	2	0.25
Semaphore Problem	49	60	10	109	2	7	16	9	0.44	1.29	0.29
Stock Exchange	179	192	106	473	13	36	103	46	0.55	1.86	0.28
Webmail app	317	409	191	1044	51	138	271	178	0.34	0.96	0.29
Wolfram's Rule 30	49	60	13	117	3	8	20	10	0.5	1.5	0.25

Table 1: Experimental results of the correction technique.

All of the experiments were conducted on an Intel Xeon E5-1660 3.3GHz CPU with 64GB RAM. Table 1 summarizes our results. We have considered five assertions per benchmark (except for the case of the *Container* program which includes the three assertions of Example 3.1). The $\#Ops$ column (resp. the $\#Eqs$ column) records the number of operator declarations (resp. the number of equations) in both the original program $\mathcal{R}$ and the statically repaired program $\mathcal{R}_{fix}$. Column TT measures the transformation time (in ms) that is required to compute the program corrections for the programs in the considered benchmark set. We also measure the average execution time (in ms) of 10 computations in the original program $\mathcal{R}$ with and without assertion checking (columns $T_{\mathcal{R}}$ and $T_{\mathcal{R}}^{Chk}$, respectively) and in the repaired program $\mathcal{R}_{fix}$ (column $T_{\mathcal{R}_{fix}}$). More specifically, execution times in $T_{\mathcal{R}}^{Chk}$ have been computed in the extended Maude runtime environment of [6] that adds assertion-checking capabilities to the standard Maude rewrite engine. This way, any state transition (i.e. rewrite rule application) $s \to s'$ is enabled in the extended runtime environment if an only if the state s' and all of its subterms meet the system assertions under examination. Note that the advantage of our correction transformation w.r.t. [6] is precisely in statically encoding the necessary checks to prevent violations into Maude programs, instead of doing unnecessary checks at runtime.

All of the executions consist of about 500 Maude computation steps, which amounts to 5,000 rewrite steps on average including equational simplification steps. In column S, we report the total speedups $(1 - T_{\mathcal{R}_{fix}}/T_{\mathcal{R}}^{Chk})$ that we achieve w.r.t. the highly optimized executions with runtime assertion checking of [6]. Finally, we record the overheads of the execution times for the original program $\mathcal{R}$ with respect to: 1) the monitorized execution times for $\mathcal{R}$, i.e., the ratio $(T_{\mathcal{R}}^{Chk} - T_{\mathcal{R}})/T_{\mathcal{R}}$ (in column O_{Chk}, taken from [6]); and 2) the execution times of the repaired program, i.e., the ratio $(T_{\mathcal{R}_{fix}} - T_{\mathcal{R}})/T_{\mathcal{R}}$ (in column O_{fix}). These overheads indicate the relative slowdown due to runtime assertion checking (column O_{Chk}) and to the evaluation of the extra conditions inserted by the correction transformation (column O_{fix}).

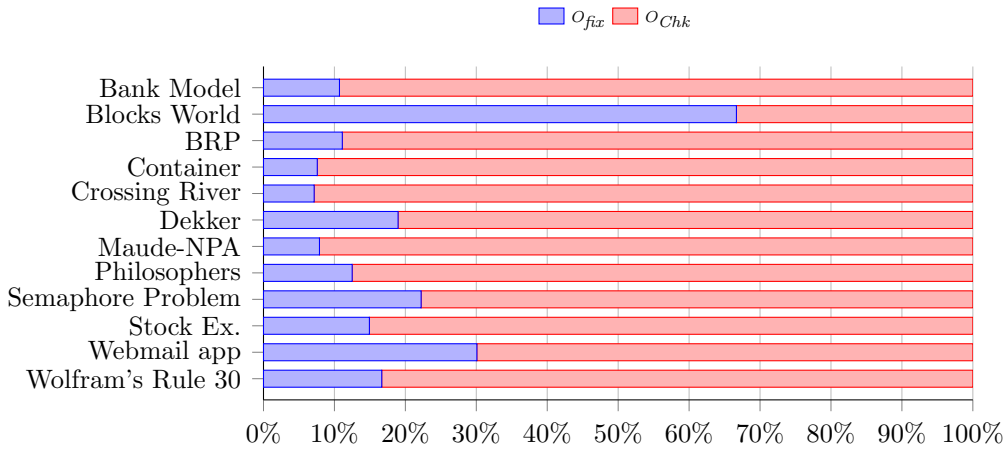

Figure 6: Overhead comparison O_{fix} vs. O_{Chk}.

Our figures show that, on average, the increasing in the number of equations grows linearly to the number of newly declared operators and the size of the corrected code is 2.8 times the size of the original code. To fairly calculate this ratio, we compared the actual size of the original code —which is the size of the user-defined specification plus the size of the extra definitions that are implicitly included in each imported module (e.g. `INT`, `RAT`, `BOOL`)— with the size of its corrected version in which extra definitions of the imported modules are explicit. In exchange for that, the correction transformation has a positive impact on the execution times w.r.t. the monitorized execution times. As expected, the corrected program $\mathcal{R}_{fix}$ is typically slower than the original program $\mathcal{R}$; nevertheless, it exhibits a better performance than $\mathcal{R}$ when run with the assertion-checking enabled. Indeed, in all cases

$T_{\mathcal{R}} \leq T_{\mathcal{R}_{fix}} \leq T_{\mathcal{R}}^{Chk}$. Moreover, the average value of the speedups in column S is 0.53, which means that running the corrected program $\mathcal{R}_{fix}$ is 53% faster than the monitorized execution of $\mathcal{R}$.

The overheads O_{fix} and O_{Chk} are graphically compared in Figure 6. The results obtained are quite satisfactory with an average value of the overheads in column O_{fix} of 0.38, which is 14.23% of the average value (2.67) of the overheads in column O_{Chk} for the same benchmark programs.

As for the time necessary for computing the program corrections, it is almost negligible (a few milliseconds) as witnessed by the data in column TT. Indeed, the worst case is 51ms for the *Webmail* specification. The time for inferring the repairs is in any case a small portion of the total execution time.

If assertions are complex (e.g., they involve recursive conditional computations), as in the *Bank Model*, our transformation can improve the execution time significantly w.r.t. the execution with dynamic assertion checking. And even when the improvement is small or not measurable (e.g., *Blocks World*), the correction transformation is useful since the transformed program is demonstrably safe and corrections are generated fast enough that they could be computed during active development, thus reducing the debugging burden.

8. Related Work and Conclusion

Automated program correction and related problems are not new, with proposed techniques ranging from semantic analysis to stochastic search [18]. Research in this area holds promise for reducing software maintenance costs due to buggy code. A number of techniques have been developed for the *code repair* problem, i.e., the general problem of computing modifications to a buggy program in order to obtain a new program that satisfies a suitable specification of the expected program behavior, or user intent. Such a specification can either be expressed as sets of passing and failing test cases, functional specifications, reference implementations, program models, sets of logic properties, examples, traces, assertions, summaries or code contracts. For instance, Gopinath et al. [19] use behavioral specifications to fix buggy Java programs, and they use the SAT-based Alloy tool-set to prune any nondeterminism that could be introduced by the repair actions. Autofix [20] bridges the gap between specification-based and test-based repairs by using Eiffel contracts to correct violations of simple assertions that are formulated

using boolean methods that are already present in the program. As for the repair synthesis itself, it can be based on different search techniques, such as enumerative and heuristic search, deduction, constraint-solving, symbolic methods, or some combination of these [18]. Other approaches for fixing code are based on statistical fault localization, evolutionary computation, or game theory. Dynamic patch generation can also be achieved by runtime monitoring and instrumentation. For a detailed discussion, see [18, 21, 22] and references therein.

More closely related to our work is the concept of *automated program repair* of [23], a change to a program source that removes bad execution traces while increasing the number of good traces by applying abstract interpretation. A bad run is one that violates a given specification either provided by the programmer (e.g., as contracts), or provided by language semantics (e.g., division by zero, null pointer, etc). A good run is one that meets all specifications of the original program. Beyond the technical differences with our work, a more basic difference is in the intention. While [23] aims to automatically produce a collection of fixes for faulty programs that are not necessarily applied in an automatic way[11], we are looking to reinforce software quality by automatically generating program corrections from a set of safety assertions so that runtime checking can be safely omitted because no crashes can occur at runtime due to assertion violation. For a recent survey on automatic software repair, see [24].

In [25] a generic strategy is defined to ensure that a Maude program satisfies a set of state invariants that can be expressed in different logics. This is achieved by imposing (on top of Maude) a programmed strategy that dynamically drives the system's execution in such a way that some state transitions are avoided so that every system state complies with the constraints. In contrast, our methodology is static and enforces the assertions by transforming the program code so that the system constraints are verified by construction.

The correction methodology that we propose can be very useful for a programmer who wants to correct a program w.r.t. a preliminary version which was written with no safety concerns. Our approach can also be applied to automatically finding fixes for programs with incomplete specifications

[11] In [23], repairs cannot be applied automatically since, in general, there is more than one repair possible for a given bug or warning.

given by system assertions while keeping the transformed programs as close as possible to the original ones.

Also, it is worth noting that the core idea of our correction transformation can be applied to virtually any rewriting-based, programming language: from simple term rewriting systems to the most widespread functional languages such as Haskell and Erlang. Indeed, the proposed correction method transforms program rules into guarded program rules whose conditions supersede the system assertion checks and are simply evaluated by using the language rewriting infrastructure. Therefore, this checking mechanism can be embedded into any setting that supports rewriting with an effort that depends on the complexity of the chosen formal framework.

As future work, we plan to improve the expressiveness of our assertion language that currently supports only safety constraints (e.g. state invariants). More concretely, the idea is to extend the language with new constructs that allow one to predicate not only over single states but also over sequences of states. This way we could, i.e., specify and enforce liveness and other more convoluted, temporal properties.

Acknowledgements

We are very grateful to Santiago Escobar for useful discussions concerning the optimization of Maude programs.

References

[1] Y. Pei, C. Furia, M. Nordio, Y. Wei, B. Meyer, A. Zeller, Automated Fixing of Programs with Contracts, IEEE Transactions on Software Engineering 40 (5) (2014) 427–449.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude: A High-Performance Logical Framework, Springer, 2007.

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

[4] F. Durán, S. Eker, P. Lincoln, J. Meseguer, Principles of Mobile Maude, in: Agent Systems, Mobile Agents, and Applications, Second International Symposium on Agent Systems and Applications and Fourth In-

ternational Symposium on Mobile Agents, ASA/MA 2000, Proceedings, 2000, pp. 73–85.

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

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

[7] F. Durán, J. Meseguer, On the Church-Rosser and Coherence Properties of Conditional Order-sorted Rewrite Theories, The Journal of Logic and Algebraic Programming 81 (7–8) (2012) 816–850.

[8] TeReSe, Term Rewriting Systems, Cambridge University Press, 2003.

[9] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, Maude Manual (Version 2.7.1), Tech. rep., SRI International Computer Science Laboratory, available at: `http://maude.cs.uiuc.edu/maude2-manual/` (2016).

[10] C. Rocha, J. Meseguer, C. Muñoz, Rewriting Modulo SMT and Open System Analysis, in: Proceedings of the 10th International Workshop on Rewriting Logic and its Applications (WRLA 2014), Vol. 8663 of Lecture Notes in Computer Science, Springer, 2014, pp. 247–262.

[11] K. Bae, J. Meseguer, A Rewriting-Based Model Checker for the Linear Temporal Logic of Rewriting, Electr. Notes Theor. Comput. Sci. 290 (2012) 19–36.

[12] J. Meseguer, P. Thati, Symbolic Reachability Analysis Using Narrowing and its Application to Verification of Cryptographic Protocols, Electronic Notes in Theoretical Computer Science 117 (2005) 153–182.

[13] C. Rocha, J. Meseguer, C. A. Muñoz, Rewriting Modulo SMT and Open System Analysis, Journal of Logical and Algebraic Methods in Programming 86 (2017) 269–297.

[14] M. -O. Stehr, J. Meseguer, P. C. Ölveczky, Rewriting Logic as a Unifying Framework for Petri Nets, in: Unifying Petri Nets, Advances in Petri

Nets, Vol. 2128 of Lecture Notes in Computer Science, Springer, 2001, pp. 250–303.

[15] J. Meseguer, C. Talcott, Semantic Models for Distributed Object Reflection, in: Proceedings of 16th European Conference on Object-Oriented Programming (ECOOP 2002), Vol. 2374 of Lecture Notes in Computer Science, Springer, 2002, pp. 1–36.

[16] The Anima Website, Available at: `http://safe-tools.dsic.upv.es/anima` (2015).

[17] M. Alpuente, D. Ballis, F. Frechina, J. Sapiña, Exploring Conditional Rewriting Logic Computations, Journal of Symbolic Computation 69 (2015) 3–39.

[18] S. Gulwani, O. Polozov, R. Singh, Program Synthesis, Foundations and Trends in Programming Languages 4 (2017) 1–119.

[19] D. Gopinath, M. Z. Malik, S. Khurshid, Specification-Based Program Repair Using SAT, in: Proceedings of the 17th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2011), Vol. 6605 of Lecture Notes in Computer Science, Springer, 2011, pp. 173–188.

[20] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, A. Zeller, Automated Fixing of Programs with Contracts, in: Proceedings of 19th International Symposium on Software Testing and Analysis (ISSTA 2010), Association for Computing Machinery, 2010, pp. 61–72.

[21] C. Goues, S. Forrest, W. Weimer, Current Challenges in Automatic Software Repair, Software Quality Journal 21 (2013) 421–443.

[22] S. Kim, C. L. Goues, M. Pradel, A. Roychoudhury, Automated Program Repair (Dagstuhl Seminar 17022), Dagstuhl Reports 7 (2017) 19–31.

[23] F. Logozzo, T. Ball, Modular and Verified Automatic Program Repair, in: Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2012), Association for Computing Machinery, 2012, pp. 133–146.

[24] L. Gazzola, D. Micucci, L. Mariani, Automatic software repair: a survey, in: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, 2018, p. 1219.

[25] M. Roldán, F. Durán, A. Vallecillo, Invariant-driven Specifications in Maude, Science of Computer Programming 74 (10) (2009) 812–835.

Appendix A. Proofs of Technical Results

Proof of Proposition 4.9. Let $\mathcal{E} = (\Sigma, \Delta \cup B)$ be an equational theory. Let $a \in \mathcal{A}$ be a system assertion of the form $\Pi \mid \varphi$ and $st \in \mathcal{T}(\Sigma, \mathcal{V})$ be a system state such that $(st)\!\downarrow_{\Delta,B} \not\models \Pi \mid \varphi$. Hence, there exist a position $w \in \mathcal{P}os(st)$ and a substitution σ such that $((st)\!\downarrow_{\Delta,B})_{|w} =_B \Pi\sigma$ and $\varphi\sigma \to^*_{\Delta,B} \mathit{false}$. By Definition 4.7, there exists the assertion checking equation

$$e_{\Pi|\varphi} = (\Pi' = \mathsf{fail}\ \mathtt{if}\ \mathtt{not}(\mathsf{Ren}^{-1}(\varphi))) \in \Delta^{\mathcal{A}}$$

Now, it is immediate to see that

$$(\mathsf{Ren}(st))_{|w} \overset{e_{\Pi|\varphi},\sigma',w}{\to}_{\Delta^{\mathcal{A}}\cup B} \mathsf{fail} \tag{A.1}$$

with the substitution $\sigma' = \{x/t' \mid x/t \in \sigma\}$. Indeed,

- $\mathsf{Ren}(st)_{|w} =_B (\mathsf{Ren}(st)_{|w})\!\downarrow_{\Delta,B} =_B \Pi'\sigma'$, since $st_{|w} =_B (st_{|w})\!\downarrow_{\Delta,B} =_B \Pi\sigma$, and $\mathsf{Ren}(st_{|w})$ and Π' are respectively the corresponding renamed versions of the canonical form $(st_{|w})\!\downarrow_{\Delta,B}$ of $st_{|w}$ and Π in $(\Sigma^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B)$;
- $\mathtt{not}(\mathsf{Ren}^{-1}(\varphi))\sigma' = \mathtt{not}(\mathsf{Ren}^{-1}(\varphi\sigma')) = \mathtt{not}(\varphi\sigma) \to^*_{\Delta^{\mathcal{A}}\cup B} \mathit{true}$ because $\varphi\sigma \to^*_{\Delta,B} \mathit{false}$ and σ' only introduces in φ renamed bindings of σ (in fact, $\sigma' = \{x/t' \mid x/t \in \sigma\}$).

Now, let us prove that if $(st)\!\downarrow_{\Delta,B} \not\models \Pi|\varphi$, then $\mathsf{Ren}(st) \to^*_{\Delta^{\mathcal{A}},B} \mathsf{fail}$. The proof is done by induction on the position $w \in \mathcal{P}os(\mathsf{Ren}(st))$ used in step (A.1) above, i.e., $\mathsf{Ren}(st)_{|w} \overset{e_{\Pi|\varphi},\sigma',w}{\to}_{\Delta^{\mathcal{A}}\cup B} \mathsf{fail}$.

($w = \Lambda$) Immediate, since $\mathsf{Ren}(st) = \mathsf{Ren}(st)_{|\Lambda}$ and $\mathsf{Ren}(st)_{|\Lambda} \overset{e_{\Pi|\varphi},\sigma',w}{\to}_{\Delta^{\mathcal{A}}\cup B} \mathsf{fail}$ by Equation A.1.

($w = i.p$) In this case, $w = i.p \in \mathcal{P}os(\mathsf{Ren}(st))$ for some natural number i that ranges from 1 to the arity of $root(\mathsf{Ren}(st))$ and some position p denoting the rest of the accessing path to $\mathsf{Ren}(st)_{|w}$. This implies that $\mathsf{Ren}(st)$ is a renamed version of the canonical form $(st_{|w})\!\downarrow_{\Delta,B}$ of st of the form $f'(t'_1, \ldots, t'_n)$ with $\mathsf{Ren}(st)_{|w} = t'_{i|p}$ (and, hence$(st_{|w})\!\downarrow_{\Delta,B} = t_{i|p}$). Because $(st_{|w})\!\downarrow_{\Delta,B} = t_{i|p}$ and $st \not\models \Pi \mid \varphi$, we also have that $t_i \not\models \Pi \mid \varphi$. By applying the induction hypothesis, we get $t'_i \to^*_{\Delta^{\mathcal{A}}\cup B} \mathsf{fail}$.

Furthermore, by Definition 4.7, there exists the equation

$$f'(x_1, \ldots, x_{i-1}, \mathtt{fail}, x_{i+1}, \ldots, x_n) = \mathsf{fail} \in \Delta^{\mathcal{A}}$$

Therefore, we can finally build the following rewrite sequence

$$\begin{aligned}\mathsf{Ren}(st) = f'(t'_1,\ldots,t'_{i-1},t'_i,t'_{i+1},\ldots,t'_n) &\rightarrow^*_{\Delta^{\mathcal{A}}\cup B} f'(t'_1,\ldots,t'_{i-1},\mathtt{fail},t'_{i+1},\ldots,t'_n)\\ &\rightarrow^*_{\Delta^{\mathcal{A}}\cup B} \mathsf{fail}.\end{aligned}$$

■

Proof of Proposition 5.5. Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a topmost rewrite theory, $\mathcal{A}$ be an assertion set, and $\mathcal{R}' = (\Sigma^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B, R^{\mathcal{A}})$ be the $\mathcal{A}$-extension of $\mathcal{R}$. By Definition 5.3, in order to show that $\mathcal{R}' \leq^{\mathcal{A}} \mathcal{R}$ (i.e., $\mathcal{R}'$ is a correction of $\mathcal{R}$ w.r.t. $\mathcal{A}$), we have to prove that

i. for all $s_0 \rightarrow_{\mathcal{R}'} \ldots \rightarrow_{\mathcal{R}'} s_n$ s.t. $s_0 \models \mathcal{A}$, there is $s_0 \rightarrow_{\mathcal{R}} \ldots \rightarrow_{\mathcal{R}} s_n$, s.t. $s_i \models \mathcal{A},\ i = 0,\ldots,n$

ii. for all $s_0 \rightarrow_{\mathcal{R}} \ldots \rightarrow_{\mathcal{R}} s_n$, s.t. $s_i \models \mathcal{A},\ i = 0,\ldots,n$, there is $s_0 \rightarrow_{\mathcal{R}'} \ldots \rightarrow_{\mathcal{R}'} s_n$.

Let us first prove Claim *i*.

[**Claim** *i*] Let us consider an arbitrary computation $\mathcal{C}^{\mathcal{A}} = (s_0 \rightarrow_{\mathcal{R}'} \ldots \rightarrow_{\mathcal{R}'} s_n)$ in $\mathcal{R}' = (\Sigma^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B, R^{\mathcal{A}})$ such that $s_0 \models \mathcal{A}$. We proceed by induction on the length n of the computation $\mathcal{C}^{\mathcal{A}}$.

$n = 0$. Immediate, since $\mathcal{C}^{\mathcal{A}}$ does not contain any rewrite step and $s_0 \models \mathcal{A}$.

$n > 0$. The computation $\mathcal{C}^{\mathcal{A}}$ can be decomposed as follows:

$$\mathcal{C}^{\mathcal{A}} = (s_0 \rightarrow_{\mathcal{R}'} \ldots \rightarrow_{\mathcal{R}'} s_{n-1} \rightarrow_{\mathcal{R}'} s_n).$$

Now, by applying the induction hypothesis to $s_0 \rightarrow_{\mathcal{R}'} \ldots \rightarrow_{\mathcal{R}'} s_{n-1}$, we know that there exists

$$\mathcal{C}^{\mathcal{A}}_{n-1} = (s_0 \rightarrow_{\mathcal{R}} \ldots \rightarrow_{\mathcal{R}} s_{n-1}),\ \text{s.t. } s_i \models \mathcal{A},\ i = 0,\ldots,(n-1)$$

Therefore, to prove Claim *i*, we just need to show that

(a) $s_{n-1} \rightarrow_{\mathcal{R}} s_n$

(b) $s_n \models \mathcal{A}$.

The proof of *(a)* is as follows: since $\mathcal{R}$ is topmost, we can expand $s_{n-1} \to_{\mathcal{R}'} s_n$ as follows

$$s_{n-1} \overset{r^{\mathcal{A}},\sigma,w}{\to}_{R^{\mathcal{A}},B} \rho\sigma \to^*_{\Delta^{\mathcal{A}},B} (\rho\sigma{\downarrow_{\Delta^{\mathcal{A}},B}}) = s_n$$

where $r^{\mathcal{A}} = (\lambda \Rightarrow \rho \text{ if } C \wedge \mathsf{Ren}(\rho) =/= \mathsf{fail}) \in R^{\mathcal{A}}$ is just a more restrictive variant of $r = (\lambda \Rightarrow \rho \text{ if } C) \in R$, thus we also have $s_{n-1} \overset{r,\sigma,w}{\to}_{R,B} \rho\sigma$. Furthermore, $\rho\sigma$ does not contain any renamed operator. This implies that all the equations of $\Delta^{\mathcal{A}}$, which are used to simplify $\rho\sigma$ into its canonical form $(\rho\sigma{\downarrow_{\Delta^{\mathcal{A}},B}})$, are also included in Δ; hence $\rho\sigma \to^*_{\Delta,B} (\rho\sigma{\downarrow_{\Delta,B}})$ as well. Therefore,

$$s_{n-1} \overset{r,\sigma,w}{\to}_{R,B} \rho\sigma \to^*_{\Delta,B} (\rho\sigma{\downarrow_{\Delta,B}}) = s_n.$$

To prove *(b)*, observe that in order to enable the rewrite step

$$s_{n-1} \overset{r^{\mathcal{A}},\sigma,w}{\to}_{R^{\mathcal{A}},B} \rho\sigma$$

included in $s_{n-1} \to_{\mathcal{R}'} s_n$, the instantiated condition $\mathsf{Ren}(\rho\sigma) =/= \mathsf{fail}$ of $r^{\mathcal{A}}$ must hold. Thus,

$$\mathsf{Ren}(\rho\sigma) \not\to^*_{\Delta^{\mathcal{A}},B} \mathsf{fail}$$

which implies that $(\rho\sigma){\downarrow_{\Delta,B}} \models \mathcal{A}$ by Proposition 4.9.

[**Claim** *ii*] We consider an arbitrary computation $\mathcal{C} = (s_0 \to_{\mathcal{R}} \ldots \to_{\mathcal{R}} s_n)$ s.t. $s_i \models \mathcal{A}$, $i = 0, \ldots, n$. Similarly to the proof of Claim i, we just proceed by induction on the length n of the computation $\mathcal{C}$ to show that there exists $s_0 \to_{\mathcal{R}'} \ldots \to_{\mathcal{R}'} s_n$. Again, the base case is trivial, while the inductive case is similar to the inductive case of [Claim i] by making use of Proposition 4.9. ■

Proof of Proposition 6.5. The case when Ax=ACU has been stated in Lemma 5.3 of [12]. Here, we prove the case when Ax=AC, which involves two extension rewrite rules, namely, $(\{X \otimes \lambda\} \Rightarrow \{X \otimes \rho\} \text{ if } C)$ and $(\{\lambda\} \Rightarrow \{\rho\} \text{ if } C)$. The remaining cases are straightforward adaptations of the following proof scheme.
We have to prove the following two implications for the case when Ax=AC:

$(\to)$ for any term t_i and t_f of sort *Config*, $t_i \to^*_{\mathcal{R}} t_f$ implies $\{t_i\} \to^*_{\hat{\mathcal{R}}} \{t_f\}$;

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

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

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

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

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

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

$$t_i = t_0 \rightarrow^*_{\mathcal{R}} t_{n-1} \text{ implies } \{t_i\} = \{t_0\} \rightarrow^*_{\hat{\mathcal{R}}} \{t_{n-1}\}. \tag{A.2}$$

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

$$\{t_{n-1}\} \rightarrow_{\hat{\mathcal{R}}} \{t_n\}, \tag{A.3}$$

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

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

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

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

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

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

($w \neq \Lambda$) Since $\mathcal{R}$ is topmost modulo Ax and $w \neq \Lambda$, there exist $s_i \in \mathcal{T}(\Sigma, \mathcal{V})_{Config}, i = 1, \ldots, k$, with $k > 1$ such that

$$t_{n-1} = s_1 \otimes \ldots \otimes s_k$$

and $t_{n-1} \in \mathcal{T}(\Sigma, \mathcal{V})_{Config}$. Now, since $t_{n-1} \overset{r,\sigma,w}{\to}_{R,B} \tilde{t}_{n-1} \to^*_{\Delta,B} t_n$, with $Ax = AC$ and $r = (\lambda \Rightarrow \rho \text{ if } C)$,

$$t_{n-1} = s_1 \otimes \ldots \otimes s_k =_{AC} s_{\pi(1)} \otimes \ldots \otimes s_{\pi(m)} \otimes \lambda\sigma \overset{r,\sigma,w}{\to}_{R,B} s_{\pi(1)} \otimes \ldots \otimes s_{\pi(m)} \otimes \rho\sigma \to^*_{\Delta,B} t_n$$

where $1 \leq m \leq k-1$, and $\pi\colon \{1, ..., m\} \to \{1, \ldots, k\}$ is an injective function that selects a permutation of m s_i's within $s_1 \otimes \ldots \otimes s_k$. Hence, we can build the following rewrite sequence

$$\{t_{n-1}\} = \{s_1 \otimes \ldots \otimes s_k\} =_{AC} \{s_{\pi(1)} \otimes \ldots \otimes s_{\pi(m)} \otimes \lambda\sigma\} \overset{\hat{r},\hat{\sigma},\Lambda}{\to}_{\hat{R},B} \{s_{\pi(1)} \otimes \ldots \otimes s_{\pi(m)} \otimes \rho\sigma\} \to^*_{\Delta,B} \{t_n\}$$

with $\hat{\sigma} = \sigma \cup \{X/s_{\pi(1)} \otimes \ldots \otimes s_{\pi(m)}\}$, and $\hat{r} = (\{X \otimes \lambda\} \Rightarrow \{X \otimes \rho\} \text{ if } C) \in \hat{R}$. This proves that $\{t_{n-1}\} \to_{\hat{\mathcal{R}}} \{t_n\}$ also in the case when $w \neq \Lambda$.

Finally, by using the induction hypothesis A.2 and the rewrite step A.3, we easily derive the implication $(\to)$.

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

$$\{t_i\} = \{t_0\} \to_{\hat{\mathcal{R}}} \ldots \to_{\hat{\mathcal{R}}} \{t_{n-1}\} \to_{\hat{\mathcal{R}}} \{t_n\} = \{t_f\}$$

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

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

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

$$\{t_i\} = \{t_0\} \to^*_{\hat{\mathcal{R}}} \{t_{n-1}\} \text{ implies } t_i = t_0 \to^*_{\mathcal{R}} t_{n-1}. \tag{A.4}$$

Therefore, it suffices to show that $t_{n-1} \to^*_{\mathcal{R}} t_n$ and combine this result with the induction hypothesis to finally prove Case $(\leftarrow)$.

By hypothesis, $\{t_{n-1}\} \to_{\hat{\mathcal{R}}} \{t_n\}$, which can be expanded into the following rewrite sequence

$$\{t_{n-1}\} \overset{\hat{r},\hat{\sigma},\Lambda}{\to}_{\hat{R},B} \{\tilde{t}_{n-1}\} \to^*_{\Delta,B} \{\tilde{t}_{n-1}\}{\downarrow_{\Delta,B}} = \{t_n\} \tag{A.5}$$

where $\hat{r} \in \hat{R}$, and $\{t_{n-1}\}, \{\tilde{t}_{n-1}\}, \{t_n\} \in \mathcal{T}(\hat{\Sigma}, \mathcal{V})_{State}$. Observe that the first rewrite step of the rewrite sequence (A.5) must occur at position Λ, since the rewrite theory $\hat{\mathcal{R}}$ is topmost.

Here, we distinguish two cases according to the form of the rewrite rule $\hat{r} \in \hat{R}$ applied in $\{t_{n-1}\} \overset{\hat{r},\hat{\sigma},\Lambda}{\rightarrow}_{\hat{R},B} \{\tilde{t}_{n-1}\}$.

By Definition 6.4, $\hat{r}$ is either $\{\lambda\} \Rightarrow \{\rho\}$ if C or $\{X \otimes \lambda\} \Rightarrow \{X \otimes \rho\}$ if C, as $Ax = AC$ and $\{t_{n-1}\}, \{\tilde{t}_{n-1}\} \in \mathcal{T}(\hat{\Sigma}, \mathcal{V})_{State}$.

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

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

Now, by Definition 6.4, $\lambda\sigma$ and $\rho\sigma$ are terms of sort *Config*; thus, we can also apply $r = (\lambda \Rightarrow \rho$ if $C) \in R$ to $\lambda\sigma$, thereby obtaining the following computation

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

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

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

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

where $c, \lambda\hat{\sigma}, \rho\hat{\sigma} \in \mathcal{T}(\Sigma, \mathcal{V})_{Config}$, and $\hat{\sigma} = \{X/c\} \cup \sigma$, for some substitution σ.

Now, by Definition 6.4, variable X does not occur in either λ or ρ; this implies that $\lambda\hat{\sigma} = \lambda\sigma$ and $\rho\hat{\sigma} = \rho\sigma$. Therefore, we can construct the following computation:

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

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

■

Proof of Corollary 6.7. Immediate by applying Proposition 5.5 to the topmost rewrite theory $\hat{\mathcal{R}}' = (\hat{\Sigma}^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B, \hat{R}^{\mathcal{A}})$. ■

Proof of Corollary 6.17. The rewrite theory $\mathcal{R}'_n = (\Sigma_n^{\mathcal{A}}, \Delta^{\mathcal{A}} \cup B, R_n^{\mathcal{A}})$ is topmost, so the result is immediate by Proposition 5.5. ■

María Alpuente is a full professor of industrial formal methods at the Technical University Valencia (UPV) in Spain where she heads the programming languages research group ELP since its creation in 1989. She received a Ph.D. degree in Computer Science from UPV in 1991. Prior to her PhD, she graduated with a B.Sc. / M.Sc. degree in Physics in 1985 at the U. of Valencia in Spain. Her research interests include automated software engineering, industrial formal methods, multi-paradigm programming, program transformation, and their application to the modeling, analysis and verification of web systems and security applications.

Demis Ballis is an assistant professor of Computer Science at the University of Udine (UniUd). He received a Ph.D. degree in Computer Science from UPV and UniUD in 2005. His research activity is mainly centered around the development of practical, cost-effective, formal methodologies for the analysis, verification, and optimization of complex software systems. In this respect, he has contributed to the development of techniques and tools that have been successfully applied to the analysis of various application domains (notably, web applications, biological systems, and UML and XML models).

Julia Sapiña is a PostDoc researcher at the Technical University Valencia (UPV) in Spain. Her areas of expertise include the dynamic analysis, debugging and optimization of complex software systems. She is currently involved in the project called *Advanced symbolic methods for the cryptographic protocol analyzer Maude-NPA*, funded by Air Force Office of Scientific Research.