Static Correction of Maude Programs with Assertions

M. Alpuente\textsuperscript{1}, D. Ballis\textsuperscript{2}, and J. Sapiña\textsuperscript{1}

\textsuperscript{1} DSIC-ELP, Universitat Politècnica de València
Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain
\{alpuente,jsapina\}@dsic.upv.es

\textsuperscript{2} DIMI, University of Udine,
Via delle Scienze, 206, 33100, Udine, Italy
demis.ballis@uniud.it

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 \( A \) of the program behavior that is given in the form of assertions together with an overly general program \( R \) whose execution might violate the assertions. Our technique translates \( R \) into a refined program \( R' \) in which every computation is also a computation in \( R \) that satisfies the assertions of \( A \). Our correction 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.

1 Introduction

Assertion checking is a practical means of validating programs. Program assertions often specify “can never happen” conditions that help detect program 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 is a high-level programming language and system that efficiently implements Rewriting Logic \[2\], 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

\textsuperscript{*} This work has been partially supported by the EU (FEDER) and the Spanish MINECO under grants TIN2015-69175-C4-1-R, and by Generalitat Valenciana ref. PROMETEOII/2015/013.
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) and as a compiler (up to 15 million rewrites per second on a standard computer). 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 [3]) or dynamically (Maude’s assertion checker [4]). 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.

Our correction transformation works with Maude programs that are equipped with system assertions, with each assertion consisting of a pair \( \Pi \mid \varphi \), where \( \Pi \) (the state template) is a term and \( \varphi \) (the state invariant) is a quantifier-free first-order formula with equality. Intuitively, system assertions specify those execution states such that, for every subterm of the state which fits the algebraic structure of \( \Pi \) with pattern matching substitution \( \sigma \), the constraints given by \( \varphi \sigma \) are satisfied.

Given a set of system assertions \( \mathcal{A} \), the notion of correction we seek is based on a binary relation \( \leq^A \) on rewrite theories, such that \( \mathcal{R}' \leq^A \mathcal{R} \) when \( \mathcal{R}' \) is a correction of \( \mathcal{R} \) w.r.t. \( \mathcal{A} \), meaning that \( \mathcal{R}' \) is a valid strengthening of \( \mathcal{R} \) that enforces \( \mathcal{A} \). This means that: 1) every execution of \( \mathcal{R}' \) is an execution of \( \mathcal{R} \) (i.e., no spurious computation states are produced); and 2) every assertion in \( \mathcal{A} \) is satisfied by all computation states in \( \mathcal{R}' \). Our correction technique works on the source code statically, so it does not depend on any concrete computations; this is in contrast with a previous, preliminary approach for dynamic program repair described in [4]. Another important feature of our technique is that it applies to Full Maude [3], 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.

Roughly speaking, our correction transformation works as follows. Starting from an overly general Maude program \( \mathcal{R} = (\Sigma, E, R) \) (that is, a program that contains all desired traces but may disprove some of the assertions), where \( E \) is a (confluent and terminating) set of \( \Sigma \)-equations and \( R \) is a set of rules, the program \( \mathcal{R} \) is coerced by inserting suitable conditions (abotted by the assertions of \( \mathcal{A} \)) in the rules of \( R \) and defining them by means of new equations that are added to \( E \) until a suitable correction is reached which satisfies all the assertions. Particular care is put in the transformation to ensure both that the new equations added to \( E \) do not break confluence or termination and that the transformed rules do not introduce spurious computations. The correction itself does not introduce new discrepancies or regression, that is, repairing an asser-
tion 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.

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 $\Pi \models \varphi$ to be defined expressing that any reachable system configuration that fits the pattern given by $\Pi$ has to invariably satisfy $\varphi$. Section 4 encodes a checking mechanism for system assertions as an expanded equational theory $E^A$ (extending the original theory $E$) that catches every possible assertion violation w.r.t. the assertion set $A$. 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 $E^A$. 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 ATAME, 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 Theories
Let us recall some important notions that are relevant to this work. We assume some basic knowledge of term rewriting [5] and Rewriting Logic [2]. Some familiarity with the Maude language [6,3] is also required.

Maude is a rewriting logic [2] 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 $\Sigma$ of operators (i.e., function symbols), with a finite poset of sorts $(S, \prec)$ that models the usual subsort relation [6]. The connected components of $(S, \prec)$ are the equivalence classes $[s]$ corresponding to the least equivalence relation $\equiv_\prec$ containing $\prec$. 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 $s$, $\text{top}(s)$ denotes the top sort of $s$.

An operator $f$ of $\Sigma$ in prefix notation is specified by notation $f : s_1 \ldots s_n \to s$, $n \geq 0$, where $s_1 \ldots s_n$ 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 $\Sigma$ in mixfix notation can be specified by using underscores as place holders for the input arguments (e.g. _\otimes_{...}:s_1s_2 \to s)$. A finite, possibly empty, sequence of sorts is denoted by $\Sigma$. We also consider an $S$-sorted family $\mathcal{V} = \{\mathcal{V}_s\}_{s \in S}$ of disjoint variable sets. $\tau(\Sigma, \mathcal{V})$ and $\tau(\Sigma)$ are the sets of terms and ground terms of sort $s$, respectively. We write $\tau(\Sigma, \mathcal{V})$ and $\tau(\Sigma)$ for the corresponding term algebras. The set of variables that occur in a term $t$ is denoted by $\text{Var}(t)$. A simple syntactic condition on $\Sigma$ and $(S, \prec)$, called preregularity \cite{dang}, ensures that each (well-formed) term $t$ always has a least-sort possible among all sorts in $S$, which is denoted by $bs(t)$.

A position $w$ in a term $t$ is represented by a sequence of natural numbers that addresses a subterm of $t$ (denotes the empty sequence, i.e., the root position). Given a term $t$, we let $\text{Pos}(t)$ denote the set of positions of $t$. By $t_{1w}$, we denote the subterm of $t$ at position $w$, and by $t[s]_w$, we denote the result of replacing the subterm $t_{1w}$ by the term $s$ in $t$. By $\text{root}(t)$, we denote the operator of $t$ that occurs at position $\lambda$.

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 $\tau(\Sigma, \mathcal{V})$, which is equal to the identity almost everywhere except for a set of variables $\{x_1, \ldots, x_n\}$. By $\varepsilon$, we denote the identity substitution. The application of a substitution $\sigma$ 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 $\sigma$ 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 $\sigma$ of $t$ in $s$. Given two substitutions $\theta$ and $\theta'$, 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 $\sim$, we define the usual transitive (resp., transitive and reflexive) closure of $\sim$ by $\sim^+$ (resp., $\sim^+\text{ref}$).

A labelled conditional equation, or simply (conditional) equation, is an expression of the form $[l]:\lambda \Rightarrow \rho$ if $C$, where $l$ is a label (i.e., a name that identifies the equation), $\lambda, \rho \in \tau(\Sigma, \mathcal{V})$ (with $ls(\lambda) \equiv ls(\rho)$), and $C$ is a (possibly empty) sequence $c_1 \land \ldots \land c_n$, where each $c_i$ is a Boolean expression\footnote{Actually, Maude supports different kinds of conditions 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.}. 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, $\lambda, \rho \in \tau(\Sigma, \mathcal{V})$ (with $ls(\lambda) \equiv ls(\rho)$), and $C$ is a (possibly empty) conjunction of Boolean expressions $c_1 \land \ldots \land c_n$. When the condition $C$ is empty, we simply write $[l]: \lambda \Rightarrow \rho$.

When no confusion can arise, rule and equation labels $[l]$ are often omitted. The term $\lambda$ (resp., $\rho$) 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 \([2]\) seamlessly combines a set \(R\) of conditional rewrite rules (or conditional term rewriting system, CTRS), with an equational theory \(E\) (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 \(E\) is a pair \((\Sigma, E)\), where \(\Sigma\) is an order-sorted signature, \(E = \Delta \cup B\) with \(\Delta\) 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 \(\Sigma\). The equational theory \(E\) induces a congruence relation on the term algebra \(\tau(\Sigma, V)\), which is denoted by \(=_{E}\).

A conditional rewrite theory (or simply, rewrite theory) is a triple \(R = (\Sigma, E, R)\), where \((\Sigma, E)\) is an order-sorted equational theory and \(R\) is a set of conditional 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 implicitly 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. Built-in operators are efficiently implemented in C++. Each syntactic element 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.

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

*Example 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.

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

sorts Container Cargo Ship Fleet State .
subsort Container < Cargo .
subsort Ship < Fleet .

op c : Int -> Container [ctor] .
op <_,_|_> : Int Int Cargo -> Ship [ctor] .
op _:_ : Fleet Cargo -> State .
ops weight size : Cargo -> Int .
op isFull : Cargo -> Bool .
op maxW : -> Int .

op none : -> Fleet .

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

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

eq weight(nil) = 0 .
eq weight(c(W),CG) = W + weight(CG) .
eq size(nil) = 0 .
eq size(c(W),CG) = 1 + size(CG) .
eq maxW = 5 .
eq isFull(nil) = true .
eq isFull(c(W),CG) = (W == maxW) and isFull(CG) .

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

crl [load] : < MAXW,MAXS | CG > FL : CG1 =>
< MAXW,MAXS | CG > FL : CG1,c(W) .
```
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 set 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 `c(W_1), c(W_2), ..., c(W_n)` 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 of ships is a term of sort `Fleet` that represents a multisets of ships and is built by means of an ACU binary operator `__,` whose identity element is the constant `none`.

The Maude module also includes some equations that specify the auxiliary functions `size` and `weight`, and `isFull`. 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` being the maximum allowed weight of each container.

The system behavior is specified by means of four rewrite rules. The rule `stow` 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` 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` 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` 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 \((\Sigma, E, R)\), 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 \(E\).
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 $\Delta$ \cite{10.1007/978-3-319-16346-6_5} Roughly speaking, the relation $\rightarrow_{\Delta,B}$ uses the equations of $\Delta$ (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 $t \downarrow_{\Delta,B}$ which might be non-terminating and non-confluent, whereas $\rightarrow_{\Delta,B}$ is called a canonical form (or irreducible form) of $t$ w.r.t. $\Delta$ 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 $\Delta$ is required to be terminating and Church-Rosser modulo $B$ in order to guarantee the existence and unicity (modulo $B$) of a canonical form w.r.t. $\Delta$ for any term $t$.

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

Roughly speaking, a conditional rewrite step on the term $t$ applies a rewrite rule/equation to $t$ by replacing a reducible (sub-)expression 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.

Under appropriate conditions on the rewrite theory, an equational rewrite step on a term $s$ can be implemented without loss of completeness by applying the following rewrite strategy \cite{2016MaudeUser}: 

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

$$s_{1} \overset{*}{\rightarrow}_{\Delta,B} s_{1} \downarrow_{\Delta,B} \rightarrow_{R,B} s_{2} \overset{*}{\rightarrow}_{\Delta,B} s_{2} \downarrow_{\Delta,B} \rightarrow_{R,B} \ldots$$

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 $\rightarrow_{\Delta,B}$, 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

\footnote{Particularly important instances of $B$-matching occur when $B$ specifies the following combinations of algebraic axioms for an operator $op$: associative axioms (A), associative 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).}
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 $R$ can be conveniently interpreted as a sequence of juxtaposed computation steps $s_1 \rightarrow_R s_2 \rightarrow_R s_3 \rightarrow_R \ldots$ 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 $s_i \rightarrow_{\Delta,B}^* s_{i+1} \downarrow_{\Delta,B}$ from $s_i$ to the canonical form of $s_{i+1}$ is called a computation step for $s_i$.

Example 2. Let $R = (\Sigma, \Delta \cup B, R)$ be the rewrite theory encoded in the Maude CONTAINER-TERMINAL module of Example 1. Then,

$$< 5,3 | \text{nil} > : c(1),c(3) \rightarrow_{R,B} < 5,3 | \text{nil} > : c(1 + 1),c(3)$$

$$\rightarrow_{\Delta,B}^{\ast} < 5,3 | \text{nil} > : c(2),c(3)$$

is a computation step in $R$ 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 [8] that allows formal properties of software systems to be specified in rewriting logic.

3 The System Assertion Language

Given the rewrite theory $R = (\Sigma, \Delta \cup B, R)$, we introduce an order-sorted assertion language $L$ whose first-order formulas define properties of the system states of $R$. To simplify our formulation, we assume that the formulas of $L$ are built over a set of user-defined Boolean operators (predicates) of $\Sigma$ whose semantics is specified by some equations in $\Delta$. The truth values of $L$ 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 [9]. Formally, a system assertion is a pair $\Pi \mid \varphi$, where $\varphi$ is a quantifier-free Boolean formula that is specialized to a (typically non-ground) term $\Pi$ in $\tau(\Sigma, V)$, with $\text{Var}(\varphi) \subseteq \text{Var}(\Pi)$.

Operationally, a system assertion $\Pi \mid \varphi$ defines a generic safety property for a state $t$ which specifies a logic invariant $\varphi$ which must be enforced in any subterm of $t$ that is an instance (modulo $\Delta \cup B$) of $\Pi$. More formally, a state $t$ satisfies $\Pi \mid \varphi$ (in symbols, $t \models \Pi \mid \varphi$) iff for every $w \in \text{Pos}(t)$ and for every substitution $\sigma$, if $t\sigma = \Delta_{\Delta,B} \Pi \sigma$ then $\varphi \sigma \rightarrow_{\Delta,B}^{\ast} \text{true}$. 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 $w \in \text{Pos}(t)$ and a substitution $\sigma$ such that $t\sigma = \Delta_{\Delta,B} \Pi \sigma$ and $\varphi \sigma \rightarrow_{\Delta,B}^{\ast} \text{false}$.
Verification of safety properties amounts to the problem of invariance checking. Hence, we say that a computation $C$ is safe w.r.t. an assertion set $A$, if for each state $s$ in $C$, $s \models A$.

**Example 3.** Let us reinforce safety of the container terminal model of Example 1 by specifying some desired properties. Let $A$ be the assertion set that includes:

- (a1) $c(W) \mid W >= 0$ and $W <= 5$
- (a2) $< MAXW,MAXS \mid CG > \mid weight(CG) <= MAXW$ and $size(CG) <= MAXS$
- (a3) $CG1,c(W),CG2 \mid isFull(c(W))$ 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 $R = (\Sigma, E, R)$ w.r.t. the assertion set $A$ that is formalized as a two-phase procedure. First, by using $A$, the equational theory $(\Sigma, E)$ in $R$ 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 in the rules of $E$ that invoke the equations of $E'$. This guarantees that the application of the transformed rules always produces states that satisfy $A$, and thus computations in the transformed theory are safe w.r.t. the assertion set $A$.

### 4 Equational Encoding of System Assertion Violations

Let $E = (\Sigma, \Delta \cup B)$ be an equational theory and $A$ 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 $E$. Additionally, we assume that the signature $\Sigma$ includes a distinguished sort $\top$ that conceptually represents a universal supersort of all sorts in $S$; i.e., $\top$ types every term in $\tau(\Sigma, V)$.

The $\top$ sort is used to define auxiliary, universal operators that can be applied to every term independently of its specific sort.

We aim to construct an extension of the equational theory $E$ in which violations of the assertions in $A$ can be automatically detected. The following definition provides a renaming procedure that is instrumental for the transformation.

---

5 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.
Definition 1 (renaming extension of an equational theory). The renaming extension of the equational theory $\mathcal{E} = (\Sigma, \Delta \cup B)$ is the equational theory $\mathcal{E}' = (\Sigma \cup \Sigma', \Delta' \cup B)$ such that

1. $\Sigma' \cap \Sigma = \emptyset$, and $\Sigma'$ contains one renamed operator $f': s_1 \ldots s_n \rightarrow s$, $n \geq 0$, for each operator $f: s_1 \ldots s_n \rightarrow s$ of $\Sigma$, and $f'$ is given the same equational attributes as $f$. Furthermore, $\Sigma'$ includes two universal operators $\text{Ren}: \top \rightarrow \top$ and $\text{Ren}^{-1}: \top \rightarrow \top$.

2. $\Delta'$ contains the following equations that define the operators $\text{Ren}$ and $\text{Ren}^{-1}$ for every possible term in $\tau(\Sigma, \mathcal{V})$ and $\tau(\Sigma', \mathcal{V})$:
   
   - (a) $\text{Ren}(f(x_1, \ldots, x_n)) = f'(\text{Ren}(x_1), \ldots, \text{Ren}(x_n))$ for every function symbol $f$ in $\Sigma$ with arity $n$, $n \geq 0$, such that $f$ obeys no unity axiom, and $\text{Ren}(x) = x$ for $x \in \mathcal{V}$.
   
   - (b) $\text{Ren}^{-1}(f'(x_1, \ldots, x_n)) = f(\text{Ren}^{-1}(x_1), \ldots, \text{Ren}^{-1}(x_n))$ for every $f' \in \Sigma'$ such that $f'$ obeys no unity axiom, and $\text{Ren}^{-1}(x) = x$ for $x \in \mathcal{V}$.
   
   - (c) For the case of binary symbols $f$ (and their renamed version $f'$) with unity element $\text{id}$, the following conditional equations are also contained in $\Delta'$:
     
     $\text{Ren}(f(x_1, x_2)) = f'(\text{Ren}(x_1), \text{Ren}(x_2))$ if $x_1 =/= \text{id} \land x_2 =/= \text{id}$
     
     $\text{Ren}^{-1}(f'(x_1, x_2)) = f(\text{Ren}^{-1}(x_1), \text{Ren}^{-1}(x_2))$ if $x_1 =/= \text{id} \land x_2 =/= \text{id}$

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

\[
\text{Ren}(t) =_{B} \text{Ren}(f(t, \text{id})) \rightarrow_{\Delta', B} f'(\text{Ren}(t), \text{Ren}(\text{id})), \rightarrow_{\Delta', B} f'(f(\text{Ren}(t), \text{Ren}(\text{id})), \text{Ren}(\text{id})) \ldots
\]

We enforce that terms of the form $\text{Ren}(t)$, with $t \in \tau(\Sigma, \mathcal{V})$, are evaluated using an eager\footnote{Eager and lazy rewrite strategies can be straightforwardly defined in Maude by resorting to the \texttt{strat} operator attribute, which allows term evaluation orders to be precisely defined.} rewrite strategy that first simplifies the input argument $t$ into the canonical form $t \downarrow_{\Delta, B}$ and then applies the equational definition of $\text{Ren}$ 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 $t \in \tau(\Sigma, \mathcal{V})$, $\text{Ren}(t) = t' \downarrow_{\Delta, B}$ where $t' \downarrow_{\Delta, B}$ is a term in $\tau(\Sigma', \mathcal{V})$ that is computed by replacing each operator $op$ of the canonical form $t \downarrow_{\Delta, B}$ with $op'$. Dually, we enforce that $\text{Ren}^{-1}(t')$ is evaluated by means of a lazy rewrite strategy that simplifies $\text{Ren}^{-1}(t')$ by applying the equations for the operator $\text{Ren}^{-1}$ at its root position, thereby undoing the renaming of $t'$ and recursively restoring the (canonical form of the) original term $t$. 

\[
\text{Ren}(t) =_{B} \text{Ren}(f(t, \text{id})) \rightarrow_{\Delta', B} f'(\text{Ren}(t), \text{Ren}(\text{id})), \rightarrow_{\Delta', B} f'(f(\text{Ren}(t), \text{Ren}(\text{id})), \text{Ren}(\text{id})) \ldots
\]
Example 4. Consider the CONTAINER-TERMINAL module of Example 1 together with the following container list
\[ c(X), c(3 + 1), c(Y) \]
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 \( c(X), c(4), c(Y) \).

Therefore, the renaming \( \text{Ren}(c(X), c(3 + 1), c(Y)) \) yields the renamed canonical form
\[ c'(X), c'(4'), c'(Y) \]
Further, \( \text{Ren}^{-1}(c'(X), c'(4'), c'(Y)) \) returns the term \( c(X), c(4), c(Y) \).

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

**Definition 2 (assertion-checking equations).** Let \( (\Sigma \cup \Sigma', \Delta \cup \Delta' \cup B) \) be the renaming extension of the equational theory \( E = (\Sigma, \Delta \cup B) \). Given the system assertion \( a = \Pi \mid \varphi \), the assertion-checking equation for \( a \), in symbols \( e_a \), is the conditional equation
\[ \Pi' = \text{fail} \text{ if not } (\text{Ren}^{-1}(\varphi)) \]
where \( \Pi' = \text{Ren}(\Pi) \), and fail is a new, universal constant (not included in \( \Sigma \cup \Sigma' \)) of sort \( \top \). Given the assertion set \( A \), we define \( A = \{ e_a \mid e \in \Delta, a \in A \} \).

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

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

- On the one hand, the application of \( \text{Ren}^{-1} \) to the logic formula \( \varphi \) ensures that any instance \( \varphi_\sigma \) of \( \varphi \) 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 \( E \).
- On the other hand, the renaming \( \Pi' \) of \( \Pi \) is needed for \( e_a \) to be terminating as shown in the following example.

**Example 5.** Consider that the CONTAINER-TERMINAL module of Example 1 is augmented by defining a new predicate function \( \text{empty?} \) that returns \text{true} if there is no container in the considered ship, and \text{false} otherwise.
empty?(< MAXW, MAXS | none >) = true .
empty?(< MAXW, MAXS | CG >) = false [owise] .

Let the assertion set $A$ consist of the single assertion

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

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

$$< MAXW, MAXS | CG >' = \text{fail}$$

$$\text{if not}(\text{Ren}^{-1}(\text{not}(empty?(< MAXW, MAXS | CG >))))$$

Now, consider the following variant of the above equation which omits renam-ning of the state template $< MAXW, MAXS | CG >$:

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

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

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 $not(\text{Ren}^{-1}(\text{not}(empty?(< MAXW, MAXS | CG >))))$ where, indeed, there are no renamed terms that can match the left-hand side of $e_a$.

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 6.** Consider again the **CONTAINER-TERMINAL** system module of Example 4 together with the assertion set $A$ of Example 3 whose associated assertion-checking equations are

$$[e_{a_1}] : c'(W) = \text{fail} \text{ if not}(\text{Ren}^{-1}(W \geq 0 \text{ and } W \leq 5))$$

$$[e_{a_2}] : < MAXW, MAXS | CG >' = \text{fail}$$

$$\text{if not}(\text{Ren}^{-1}(\text{weight}(CG) \leq MAXW \text{ and } \text{size}(CG) \leq MAXS))$$

$$[e_{a_3}] : CG_1', c'(W)', CG_2 = \text{fail}$$

$$\text{if not}(\text{Ren}^{-1}(\text{isFull}(c(W)) \implies \text{isFull}(CG_1)))$$

---

7 Note that, in the case of a single mixfix operator such as the **Ship** constructor $< \_ | \_ >$, for simplicity, we just rename one operator symbol, $< \_ | \_ >'$.
Now, observe that confluence can be broken by applying equations \(e_{a_1}\) and \(e_{a_3}\) to the renamed container list

\[ c'(6'), c'(5'). \]

Indeed, both assertions \(a_1\) and \(a_3\) are violated in the given container list. The former is not satisfied because \(c'(6')\) exceeds the weight upper limit of 5 tons, while the latter is refuted since there is a full container \(c'(5')\) that is preceded in the container list by the overweighted container \(c'(6')\) (which actually should be full but not overweighted). Finally, note that the applications of the equations \(e_{a_1}\) and \(e_{a_3}\) to \(c'(6'), c'(5')\) yield two distinct irreducible forms, which are, respectively,

\[ \text{fail}', c'(5') \quad \text{and} \quad \text{fail}. \]

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 3 (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 \(f': s_1 \ldots s_n \rightarrow s \in \Sigma', n \geq 0\), we define the set of fail-detecting equations \(F_{f'}\) such that

\[
F_{f'} = \begin{cases} 
\{ f'(x_1, \ldots, x_i, \text{fail}, x_{i+1}, \ldots, x_n) = \text{fail} \mid i = 1, \ldots, n \} & \text{for every } f' \in \Sigma' \text{ without unity, } n \geq 0 \\
\{ f'(x_1, \text{fail}) \text{ if } x_1 \neq \text{fail} \} \cup \{ f'(\text{fail}, x_2) \text{ if } x_2 = \neq \text{fail} \} & \text{for every binary operator } f' \in \Sigma' \text{ with unity id}
\end{cases}
\]

Similarly to Definition 1, Definition 3 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 \(A\)-extension of an equational theory \((\Sigma, \Delta \cup B)\).

**Definition 4 (\(A\)-extension of \(E\)).** Let \(E = (\Sigma, \Delta \cup B)\) denote the equational theory that is encoded in the CONTAINER-TERMINAL module of Example 1 and consider again the assertion set \(A\) of Example 3. The \(A\)-extension \((\Sigma^A, \Delta^A \cup B)\) of the equational theory \(E\) such that

- \(\Sigma^A = \Sigma \cup \Sigma' \cup \{ \text{fail} \rightarrow \top \}\)
- \(\Delta^A = \Delta \cup \Delta' \cup A \cup \text{fail}\)

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

**Example 7.** Let \(E = (\Sigma, \Delta \cup B)\) denote the equational theory that is encoded in the CONTAINER-TERMINAL module of Example 1 and consider again the assertion set \(A\) of Example 3. The \(A\)-extension \((\Sigma^A, \Delta^A \cup B)\) of the equational theory \(E\)
is obtained by expanding the renaming extension $E' = (\Sigma', \Delta' \cup B)$ of $E$ with the assertion-checking equations $e_{a1}$, $e_{a2}$, and $e_{a3}$ of Example 6, and the fail-detecting equations from $F = \bigcup_{f' \in \Delta'} F_{f'}$, which specifically includes

$$F_{f' \in \Delta' \cup B} = \{(X, \text{fail} = \text{fail if } X =/= \text{nil})\}$$

$$F_{\text{fail} \in \Delta' \cup B} = \{\langle <\text{fail, } X | Y >', \text{fail} = \text{fail if } X =/= \text{nil} \rangle, \langle \text{fail, } X | Y >', \text{fail} = \text{fail if } X =/= \text{nil} \rangle, \langle \text{fail, } X | Y >', \text{fail} = \text{fail if } X =/= \text{nil} \rangle\}$$

Consider now a term $\text{ship} = < 20, 3 | c(8), c(9) >$ which raises two violations of the assertion $a1 \in A$, since both $c(8)$ and $c(9)$ are overweighted. These violations are captured by the following equational simplifications that reduce $\text{Ren}(\text{ship})$ to the unique irreducible form fail by using the assertion-checking equation $e_{a1}$ as well as the fail-detecting equations in $F_{\text{fail}, \text{fail} | X}$. For $F_{\text{fail}, \text{fail} | X}$, we have:

$$\text{Ren}(\text{ship}) \rightarrow_{\Delta A, B} < 20', 3' | \text{fail,'c'}'(9') >' \rightarrow_{\Delta A, B} < 20', 3' | \text{fail>}'

$$\text{Ren}(\text{ship}) \rightarrow_{\Delta A, B} < 20', 3' | \text{fail,'c'}'(8') >' \rightarrow_{\Delta A, B} < 20', 3' | \text{fail>}'

$$\text{Ren}(\text{ship}) \rightarrow_{\Delta A, B} < 20', 3' | \text{fail>}' \rightarrow_{\Delta A, B} \text{fail}$$

$$\text{Ren}(\text{ship}) \rightarrow_{\Delta A, B} < 20', 3' | \text{fail>}' \rightarrow_{\Delta A, B} \text{fail}$$

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

**Proposition 1 (completeness).** Let $E$ be an equational theory, $A$ be an assertion set, and $E^A = (\Sigma^A, \Delta^A \cup B)$ be the $A$-extension of $E$. Let $a \in A$ and $st \in \tau(\Sigma, V)$. If $st \not\rightarrow_{\Delta A, B} a$, then $\text{Ren}(st) \rightarrow_{\Delta A, B} \text{fail}$.

### 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 [10]).

A topmost rewrite theory can be defined as follows [11].

**Definition 5.** Let $R = (\Sigma, E, R)$ be a rewrite theory. Let $S$ be the set of sorts of $\Sigma$. Then, $R$ is topmost if, for some top sort $\text{State} \in S$,

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

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

**Example 8.** The CONTAINER-TERMINAL module of Example 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 6.** 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 \( \mathcal{R}' \leq^A \mathcal{R} \)) 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 6 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 \( s_1 \xrightarrow{r,\sigma,w}_{\mathcal{R},B} s_2 \), with rule \( r = (\lambda \Rightarrow \rho \text{ if } C) \), yields a state \( s_2 \) that is an instance of the right-hand side \( \rho \) 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 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 \( \text{Ren}(\rho) =/= \text{fail} \) 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 \( \rho \) of \( r \) by checking that (the corresponding renamed instances of) \( \rho \) cannot be equationally simplified to the constant \( \text{fail} \) 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 7 (\( \mathcal{A} \)-extension of \( \mathcal{R} \)).** Let \( \mathcal{R} = (\Sigma, \Delta \cup B, R) \) be a topmost rewrite theory, \( \mathcal{A} \) be an assertion set, and \( (\Sigma^A, \Delta^A \cup B) \) 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 \( (\Sigma^A, \Delta^A \cup B, R^A) \), where

\[
R^A = \{ \lambda \Rightarrow \rho \text{ if } C \land \text{Ren}(\rho) =/= \text{fail} \mid (\lambda \Rightarrow \rho \text{ if } C) \in R \}.
\]

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 2. Let $R = (\Sigma, \Delta \cup B, R)$ be a topmost rewrite theory, $A$ be an assertion set, and $R'$ be the $A$-extension of $R$ formalized in Definition 7. Then $R' \leq^A R$.

Example 9. Consider the topmost rewrite theory $R = (\Sigma, \Delta \cup B, R)$ encoded by the CARGO system module of Example 1 and the assertion set $A$ of Example 3. Then the $A$-extension of $R$ w.r.t. $A$ is the rewrite theory $R' = (\Sigma^A, \Delta^A \cup B, R^A)$ where

- $(\Sigma^A, \Delta^A \cup B)$ is the $A$-extension of $(\Sigma, \Delta \cup B)$ that has been computed in Example 7;
- $R^A$ is the set that contains the following rewrite rules

\[
\begin{align*}
\text{crl [stow']} : & < \text{MAXW,MAXS | CG > FL : c(W),CG} = c(0),CG1 => \\
& < \text{MAXW,MAXS | CG,c(W) > FL : CG1} \\
& \text{if weight(CG,c(W)) <= MAXW /} \\
& \text{Ren(< MAXW,MAXS | CG,c(W) > FL : CG1) /= fail.}
\end{align*}
\]

\[
\begin{align*}
\text{crl [unstow']} : & < \text{MAXW,MAXS | c(W),CG > FL : CG1 =>} \\
& < \text{MAXW,MAXS | c(W) > FL : CG1,c(W)} \\
& \text{if Ren(< MAXW,MAXS | c(W) > FL : CG1,c(W)) /= fail.}
\end{align*}
\]

\[
\begin{align*}
\text{crl [load']} : & < \text{MAXW,MAXS | CG > FL : CG1,c(W),CG2 =>} \\
& < \text{MAXW,MAXS | CG > FL : CG1,c(W + 1),CG2} \\
& \text{if not(isFull(c(W))/} \\
& \text{Ren(< MAXW,MAXS | CG > FL : CG1,c(W + 1),CG2) /= fail.}
\end{align*}
\]

\[
\begin{align*}
\text{crl [unload']} : & < \text{MAXW,MAXS | CG > FL : CG1,c(W),CG2 =>} \\
& < \text{MAXW,MAXS | CG > FL : CG1,c(W - 1),CG2} \\
& \text{if Ren(< MAXW,MAXS | CG > FL : CG1,c(W - 1),CG2 ) /= fail.}
\end{align*}
\]

By Proposition 2, $R' \leq^A R$, which implies that $R'$ reproduces all and only the computations of $R$ that are safe w.r.t. $A$. For instance, the following computation step

\[
< 10,3 | c(5) > : c(0) \rightarrow_R < 10,3 | c(5) > : c(-1)
\]

can be given in $R$ 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 $R'$ because the transformed rule unload' cannot be applied to $s_i$. In fact, the instantiated guard

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

in the conditional part of unload' evaluates to false and thus prevents the rule unload' from being fired.
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 2.

The first class we consider generalizes the topmost modulo ACU rewrite theories of [10] 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 $\otimes : Config \times Config \rightarrow Config$ of the signature. The operator $\otimes$ is used to build system configuration 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 [10]. Furthermore, different styles of Petri nets can also be modeled via topmost modulo $Ax$ rewrite theories [12].

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 $s \times s \rightarrow State$. 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 8 (Topmost modulo $Ax$ rewrite theory). Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a rewrite theory with a finite poset of sorts $(S, \prec)$. Let Config be a top sort in
Let \( \Sigma \), and \( \otimes : \text{Config} \times \text{Config} \rightarrow \text{Config} \in \Sigma \) be a binary operator that obeys a combination of associativity, commutativity, and/or identity axioms \( Ax \in \alpha(\otimes) \subseteq B \).

The theory \( R \) is said to be topmost modulo \( Ax \) if

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

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 10. Let us consider the following Maude code fragment that specifies the FlatCargo data structure as a (flat) multiset of containers \( c(W_1), c(W_2), \ldots, c(W_n) \).

\[
\begin{align*}
sorts & \text{Container FlatCargo} . \\
\text{subsort} & \text{Container} < \text{FlatCargo} . \\
\text{op} & c : \text{Int} \rightarrow \text{Container} . \\
\text{op} & \text{nil} : \rightarrow \text{FlatCargo} . \\
\text{op} & \_ , \_ : \text{FlatCargo} \times \text{FlatCargo} \rightarrow \text{FlatCargo} [\text{ctor assoc comm id: nil}] . \\
\text{var} & W : \text{Int} . \\
\text{rl} \ [\text{load}] : & c(W) \Rightarrow c(W + 1) . \\
\text{rl} \ [\text{unload}] : & c(W) \Rightarrow c(W - 1) .
\end{align*}
\]

It is immediate to see that this code fragment encodes a topmost modulo ACU rewrite theory. It suffices to interpret the sort \( \text{FlatCargo} \) as the sort \( \text{Config} \) of Definition 8, and the ACU operator \( \text{op} \_ , \_ : \text{FlatCargo} \times \text{FlatCargo} \rightarrow \text{FlatCargo} [\text{ctor assoc comm id: nil}] \) as the operator \( \otimes : \text{Config} \times \text{Config} \rightarrow \text{Config} \). Note that the rule \text{load} (respectively, \text{unload}) allows the weight of an arbitrary container to be locally increased (respectively, decreased) within a cargo configuration \( c(W_1), \ldots, c(W_n) \).

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 11. Consider again the Maude code fragment of Example 10 and an assertion set \( \mathcal{A} \) that consists of a single assertion that enforces the containers in every system configuration to be pairwise distinct:

\[
c(W_1), c(W_2) \mid W_1 \neq W_2
\]
Now, if we apply the correction technique of Section 3, the following assertion-checking equation is synthesized
\[
c'(W1), c'(W2) = \text{fail} \text{ if } \neg (\text{Ren}^{-1}(W1 =/= W2)) \tag{2}
\]
and the load rewrite rule of Example 10 is refined into the conditional rule
\[
crl [\text{load}'] : c(W) \Rightarrow c(W + 1) \text{ if } \text{Ren}(c(W + 1)) =/= \text{fail}. \tag{3}
\]
that only increments the container weight provided the condition
\[
\text{Ren}(c(W + 1)) =/= \text{fail} \tag{4}
\]
is satisfied.

Unfortunately, including Condition (4) into the definition of \text{load}' is not enough for the transformed theory to be correct w.r.t. \( \mathcal{A} \). In fact, in such a naively transformed theory, the following undesired rewrite step can be given by applying \text{load}'
\[
c(0), c(1), c(2) \xrightarrow{\text{load}'} c(0 + 1), c(1), c(2)
\]
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 10 can be overcome by transforming a rewrite theory \( \mathcal{R} \) that is topmost modulo \( \mathcal{A} \) into an equivalent topmost theory \( \hat{\mathcal{R}} \) to which our correction technique applies. Such a transformation was formerly studied in [10] for the case when only ACU operators are considered. Here we extend it with the combinations AC, AU and A.

**Definition 9 (topmost extension of \( \mathcal{R} \)).** Let \( \mathcal{R} = (\Sigma, E, R) \) be a topmost modulo \( \mathcal{A} \) rewrite theory, where \( E = \Delta \cup B \) and \( \mathcal{A} \in B \). Let \( X, X_1, \) and \( X_2 \) be variables of sort \( \text{Config} \) not occurring in either \( R \) or \( E \). We define the topmost rewrite theory \( \hat{\mathcal{R}} = (\hat{\Sigma}, E') \) where \( \hat{\Sigma} \) extends \( \Sigma \) by adding a new top sort \( \text{State} \), and a new operator \( \{.;\} : \text{Config} \rightarrow \text{State} \) and \( \hat{\mathcal{R}} \) is obtained by transforming \( R \) according to \( \mathcal{A} \) as follows.

For each \( (\lambda \Rightarrow \rho \text{ if } C) \in R \)

<table>
<thead>
<tr>
<th>Case ( Ax = ACU )</th>
<th>( {X \odot \lambda } \Rightarrow {X \odot \rho \text{ if } C} \in \hat{R}; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case ( Ax = AC )</td>
<td>( {X \odot \lambda } \Rightarrow {X \odot \rho \text{ if } C} \in \hat{R}; )</td>
</tr>
<tr>
<td></td>
<td>( {\lambda} \Rightarrow {\rho \text{ if } C} \in \hat{R}; )</td>
</tr>
<tr>
<td>Case ( Ax = AU )</td>
<td>( {X_1 \odot \lambda \odot X_2} \Rightarrow {X_1 \odot \rho \odot X_2 \text{ if } C} \in \hat{R}; )</td>
</tr>
<tr>
<td>Case ( Ax = A )</td>
<td>( {X_1 \odot \lambda \odot X_2} \Rightarrow {X_1 \odot \rho \odot X_2 \text{ if } C} \in \hat{R}; )</td>
</tr>
<tr>
<td></td>
<td>( {X_1 \odot \lambda} \Rightarrow {X_1 \odot \rho \text{ if } C} \in \hat{R}; )</td>
</tr>
<tr>
<td></td>
<td>( {\lambda \odot X_1} \Rightarrow {\rho \odot X_1 \text{ if } C} \in \hat{R}; )</td>
</tr>
<tr>
<td></td>
<td>( {\lambda} \Rightarrow {\rho \text{ if } C} \in \hat{R}; )</td>
</tr>
</tbody>
</table>
We call \( \hat{R} \) the topmost extension of \( R \).

**Proposition 3.** Let \( R \) be a topmost modulo \( Ax \) theory and \( \hat{R} \) be the topmost extension of \( R \). For any term \( t_i \) and \( t_f \) of sort Config, \( t_i \rightarrow^* R t_f \) iff \( \{t_i\} \rightarrow^*_\hat{R} \{t_f\} \).

**Example 12.** Consider the topmost modulo ACU rewrite theory \( R \) that is specified by the Maude code fragment of Example 10. By computing its topmost extension \( \hat{R} \), the rewrite rules \texttt{load} and \texttt{unload} are transformed into the rules

\[
\begin{align*}
\text{rl [load-ACU]} &: \{ X,c(W) \} \Rightarrow \{ X,c(W + 1) \} . \\
\text{rl [unload-ACU]} &: \{ X,c(W) \} \Rightarrow \{ X,c(W - 1) \} .
\end{align*}
\]

Now, the undesired computation of Example 10 is mimicked in \( \hat{R} \) by the following computation

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

in which the \texttt{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 2).

**Corollary 1.** Let \( R = (\Sigma, \Delta \cup B, R) \) be a topmost modulo \( Ax \) rewrite theory, with \( Ax \in B \). Let \( A \) be an assertion set, \( \hat{R} = (\Sigma, \Delta \cup B, \hat{R}) \) be the topmost extension of \( R \), and \((\Sigma^A, \Delta^A \cup B)\) be the \( A \)-extension of \((\Sigma, \Delta \cup B)\).

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

\[
\hat{R}^A = \{ \lambda \Rightarrow \rho \mid \text{Ren}(\rho) \neq \text{fail} \}.
\]

Then \( \hat{R}' \leq^A \hat{R} \).

**Example 13.** Consider the assertion set \( A = \{ c(W1), c(W2) \mid W1 \neq W2 \} \) of Example 11 and the rewrite theory \( R \) of Example 10, together with the topmost extension \( \hat{R} \) of \( R \) given in Example 12. Then, \( \hat{R}' = (\Sigma^A, \Delta^A \cup B, \hat{R}^A) \) includes the conditional rule

\[
\text{crl [loadC-ACU']} : \{ X,c(W) \} \Rightarrow \{ X,c(W + 1) \}
\]

if \( \text{Ren}( X,c(W + 1) ) \neq \text{fail} \).

Note that the application of \texttt{[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 \( \text{Ren}( X,c(W + 1) ) \neq \text{fail} \), which is true if no violation of the assertion in \( A \) is detected (that is, the containers in \( M' \) are pairwise disjoint).

Now, as expected, the erroneous computation of Example 12, which violates \( A \), cannot be reproduced in \( \hat{R}' \).
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 $R = (\Sigma, \Delta \cup B, R)$, we are always able to partition the set of rewrite rules $R$ into two disjoint sets $R_A$ and $R_{>A}$ such that $R_A = \{ \lambda \Rightarrow \rho \; \text{if} \; C \mid \lambda, \rho \; \text{are of sort} \; \text{State}\}$ and $R_{>A} = \{ \lambda \Rightarrow \rho \; \text{if} \; C \mid \lambda, \rho \; \text{are of sort} \; \text{Config}\}$. The set $R_A$ contains the topmost rewrite rules in $R$ that globally rewrite a state at its root position, while $R_{>A}$ includes the topmost modulo $Ax$ rules that locally rewrite an inner state fragment (that is, a configuration within the state).

Now, $R$ can be turned into a topmost rewrite theory by reusing the sort $\text{State}$ included in $R$ and simply applying the program transformation of Definition 9 to $R_{>A}$, while leaving $R_A$ unchanged.

**Example 14.** Let $R = (\Sigma, \Delta \cup B, R_A \cup R_{>A})$ be a rewrite theory such that $\Sigma$ includes the operators $\{\}_: \text{Config} \rightarrow \text{State}$, $a: \rightarrow \text{Config}$, $c: \rightarrow \text{Config}$, as well as the ACU operator $\otimes: \text{Config Config} \rightarrow \text{Config}$, and

$$R_A = \{ [r_1]: \{a \otimes X \} \Rightarrow \{X\}\}$$
$$R_{>A} = \{ [r_2]: a \Rightarrow c \}$$

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

Note that the rule $r_1 \in R_A$ globally rewrites terms of the form $\{t_1 \otimes \ldots \otimes t_n\}$ of sort $\text{State}$, while $r_2 \in R_A$ performs a local state change, that is, it allows a state fragment, namely, the configuration $a$, to be rewritten into the configuration $c$.

Now, by applying the transformation of Definition 9 to $r_2$, we get the topmost rule $[r_3]: \{X \otimes a\} \Rightarrow \{X \otimes c\}$, where $X$ is a variable of sort $\text{Config}$, and the rewrite theory $(\Sigma, \Delta \cup B, \{r_1, r_3\})$ is thus a topmost rewrite theory (equivalent to $R$) 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 [13], 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\(^8\) of the form \(b : s_1 \ldots s_n \ Config \rightarrow Config, n \geq 0\), which allows a configuration of sort \(Config\) to be encapsulated in a well-delimited structure. This structure may also include additional parameters of sorts \(s_1, \ldots, s_n\) 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. \(A\) can be computed. This is essentially done by adapting the program transformation of [10], 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 [10], the formalization of the class of the Russian doll theories requires the following auxiliary definitions.

**Definition 10.** Let \(\Sigma\) be a signature, whose set of sorts is \(S\). We say that \(\Sigma\) 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 \(\Sigma\) whose arity includes a sort \(s\) such that \(\text{top}(s) = Config\) are:
  
  \[
  \cdot \otimes \cdot : FlatConfig \ FlatConfig \rightarrow FlatConfig \\
  \cdot \otimes \cdot : Config \ Config \rightarrow Config \\
  b : \# \rightarrow \ Config \rightarrow Config
  \]

  where \(\otimes\) obeys the algebraic axioms \(ACU\). Furthermore, for each operator \(f : \# \rightarrow Config \in \Sigma\), \(f\) is either \(\otimes\) or \(b\).

**Definition 11.** Given a Russian doll signature \(\Sigma\), a term \(t \in \tau(\Sigma, V)\) is of bounded nesting if, for all \(x \in \text{Var}(t)\), \(x\) 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:

\[
\text{depth}(t) = \begin{cases} 
0 & t \notin \tau(\Sigma, V)_{Config} \text{ or } t \in \tau(\Sigma, V)_{FlatConfig} \\
\max(\text{depth}(t_1), \text{depth}(t_2)) & t = t_1 \otimes t_2 \\
\text{depth}(t) + 1 & t = b(\# p, t_1)
\end{cases}
\]

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

---

\(^8\) To keep the exposition simple, here we consider a single boundary operator \(b\). However, as described in [13], multiple boundary operators of the form \(b_j : \#_{s_j} Config \rightarrow Config, j = 1, \ldots, m\) could be specified, each of which has distinct argument sorts \(s_j\).
Definition 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. $\Sigma$ is a Russian doll signature;
2. for each equation $(\lambda = \rho \text{ if } C) \in \Delta$ and substitution $\sigma$, $\sigma(\lambda)$ is of bounded nesting iff $\sigma(\rho)$ is of bounded nesting, and if $\sigma(\lambda)$ and $\sigma(\rho)$ are of bounded nesting, then $\text{depth}(\sigma(\lambda)) = \text{depth}(\sigma(\rho))$;
3. for each rewrite rule $(\lambda \Rightarrow \rho \text{ if } C) \in R$, $\lambda$ and $\rho$ are of sort $\text{Config}$, and for each substitution $\sigma$ such that $\sigma(\lambda)$ and $\sigma(\rho)$ are of bounded nesting, $\text{depth}(\sigma(\lambda)) \geq \text{depth}(\sigma(\rho))$.

Roughly speaking, Condition 2 in Definition 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, $t_1 \rightarrow_R t_2$ implies that $t_2$ is of bounded nesting and $\text{depth}(t_1) \geq \text{depth}(t_2)$.

Example 15. The following fragment of Maude code extends the Cargo data structure of Example 1 that models flat lists of containers $c(W_1), c(W_2), \ldots, c(W_n)$ by considering multisets of elements that can be either simple containers $c(W)$ 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 $\text{FlatCargo}$ and $\text{Cargo}$ as the sorts $\text{FlatConfig}$ and $\text{Config}$ of Definition 12, and the operators

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

as the operators $\otimes : \text{FlatConfig} \text{ FlatConfig} \rightarrow \text{FlatConfig}$, $\otimes : \text{Config} \text{ Config} \rightarrow \text{Config}$, and $b : \bar{s} \text{ Config} \rightarrow \text{Config}$, where the sort list $\bar{s}$ is empty. The new Cargo data structures 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 specifies 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 [10] that turns a Russian doll rewrite theory \( R \) into a topmost rewrite theory \( R_n \), which is able to deal with configurations whose rule structure has a fixed nesting depth.

**Definition 13.** Let \( R = (\Sigma, \Delta \cup B, R) \) be a Russian doll rewrite theory, and \( n \) be a natural number. We define the topmost \( n \)-extension of \( R \) as the topmost rewrite theory \( R_n = (\Sigma_n, \Delta \cup B, R_n) \) where

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

\[
\{ C_0 \otimes b(\mathcal{X}_1, C_1 \otimes b(\mathcal{X}_2, \ldots b(\mathcal{X}_k, C_k \otimes \lambda) \ldots)) \} \Rightarrow \{ C_0 \otimes b(\mathcal{X}_1, C_1 \otimes b(\mathcal{X}_2, \ldots b(\mathcal{X}_k, C_k \otimes \rho) \ldots)) \}
\]

where \( C_0, C_1, \ldots, C_k \) are variables of sort Config, and \( \mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_k \) are sequences of variables of the sorts required by the boundary operator \( b \).

**Example 16.** Consider the Russian doll rewrite theory encoded by the Maude fragment of Example 15. Then, its topmost 1-extension is a topmost rewrite theory that contains the following rewrite rules:

\[
\begin{align*}
\text{rl \ [insert-0]} & : \{ C_0, X_1, [ X_2 ] \} \Rightarrow \{ C_0, [ X_1, X_2 ] \}. \\
\text{rl \ [insert-1]} & : \{ C_0, [ C_1, X_1, [ X_2 ] ] \} \Rightarrow \{ C_0, [ C_1, X_1, X_2 ] \}. \\
\text{rl \ [load-simple-0]} & : \{ C_0, c(w) \} \Rightarrow \{ C_0, c(w + 1) \}. \\
\text{rl \ [load-simple-1]} & : \{ C_0, [ C_1, c(w) ] \} \Rightarrow \{ C_0, [ C_1, c(w + 1) ] \}. 
\end{align*}
\]

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 \( X_1 \) located at depth 0 and 1, respectively, to a deeper compound container.

Equivalence between a Russian doll rewrite theory and its \( n \)-extension \( R_n \) 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 \( R \), then \( t_i \) can be rewritten in \( t_f \) in \( R \) if and only if \( \{t_i\} \) can be rewritten in \( \{t_f\} \) in \( R_n \). More formally,

**Proposition 4 ([10]).** Let \( R \) 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 \( R_n \) be the topmost \( n \)-extension of \( R \). Then, \( t_i \rightarrow^*_R t_f \) iff \( \{t_i\} \rightarrow^*_{R_n} \{t_f\} \).
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 2) w.r.t. $\mathcal{A}$. Indeed, the following corollary holds.

**Corollary 2.** 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 $(\Sigma_n^A, \Delta^A \cup B)$ be the $A$-extension of $(\Sigma_n, \Delta \cup B)$.

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

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

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

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

Now, by Definition 13 the topmost 1-extension $\mathcal{R}_1$ of $\mathcal{R}$ contains the following two rules that mimick $r$ within $\mathcal{R}_1$:

$$[r_1]: \{C_0 \otimes a\} \Rightarrow \{C_0 \otimes c\}$$

$$[r_2]: \{C_0 \otimes [C_1 \otimes a]\} \Rightarrow \{C_0 \otimes [C_1 \otimes c]\}$$

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:

$$[r_1]: \{C_0 \otimes a\} \Rightarrow \{C_0 \otimes c\}$$

$$[r_2]: \{C_0 \otimes [C_1 \otimes a]\} \Rightarrow \{C_0 \otimes [C_1 \otimes c]\}$$

$$[r_3]: \{a\} \Rightarrow \{c\}$$

$$[r_4]: \{|a|\} \Rightarrow \{|c|\}$$

$$[r_5]: \{C_0 \otimes |a|\} \Rightarrow \{C_0 \otimes |c|\}$$

$$[r_6]: \{|C_1 \otimes a|\} \Rightarrow \{|C_1 \otimes c|\}$$

Therefore, we need 6 rules in $\mathcal{R}_1$ to mimick the behavior of $r$ 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 ATAME (Assertion-based Theory Amendment in Maude). The tool has been implemented in Maude itself by using Maude’s meta-level capabilities. ATAME 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 ATAME 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 ATAME. Then, we summarize the experimental results that we obtain on a set of representative benchmarks.

7.1 ATAME to the Rescue: a Typical Repair Session

Let us show how ATAME works in practice by showing a repair session for our container terminal specification of Example 1.

Maude programs can be either uploaded in ATAME 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 1 is provided with the tool for demonstration purposes.

Hence, to start the repair session, we can just select the CONTAINER-TERMINAL module under examination from the preloaded programs (see Figure 1) and proceed through the next steps.

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 2 illustrates the input phase of the system assertions of Example 3 in ATAME. Finally, by pressing 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 3 shows (a fragment of) the correction generated for the CONTAINER-TERMINAL module (i.e., the CONTAINER-TERMINAL-FIX module).

As an additional feature, ATAME provides the interconnection with the ANIMA Maude stepper [14], which integrates program animation capabilities [15] into ATAME. Specifically, by means of the Animate button, the user is allowed to interactively inspect a stepwisely generated fragment of the computation tree of the corrected program for a given initial state.

---

9 The computation tree of a Maude program M for a given initial state s subsumes all the concurrent, nondeterministic, computations of M stemming from s.
Figure 1: CONTAINER-TERMINAL module loaded in ÁTAME.

Figure 4 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

$$CG_1, c(W), CG_2 | isFull(c(W)) \implies isFull(CG_1)$$

and thus it cannot appear in any computation of CONTAINER-TERMINAL-FIX.
Add the extra predicates used in your assertions:

```
mod CONTAINER-TERMINAL-PRED is
  pr CONTAINER-TERMINAL .
  sort Assertion .
  op _ |_ : Universal Bool -> Assertion [ ctor prec 125 gather (e e) poly (1) ] .
endm
```

Based on your program and predicates, specify your assertions (one per line):

```
< c(W:Int) | W:Int >= 0 and W:Int <= 5
< MAXW:Int , MAXS:Int | CG:Cargo > | weight(CG:Cargo) <= MAXW:Int and size(CG:Cargo) <= MAXS:Int
CG1:Cargo,c(W:Int),CG2:Cargo | isFull(c(W:Int)) implies isFull(CG1:Cargo)
```

You may extend this module with additional declarations in order to fully support the reduction of your boolean assertion formulas.

---

**Figure 2:** Input of the system assertions of Example 3 for the CONTAINER-TERMINAL 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 ma-
nipulation in a container terminal of the running Example 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.

### Table 1: Experimental results of the correction technique.

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

All of the experiments were conducted on an Intel Xeon E5-1660 3.3GHz CPU with 64GB RAM. Table 1 summarizes our results. The #Ops column (resp. the #Eqs column) records the number of operator declarations (resp. the number of equations) in both the original program $R$ and the statically repaired program $R\textsubscript{fix}$. 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 $R$ with and without assertion checking (columns $T_R$ and $T_R\textsubscript{Chk}$, respectively) and in the repaired program $R\textsubscript{fix}$ (column $T_R\textsubscript{fix}$). 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 $(1 - T_{R\textsubscript{fix}} / T_{R\textsubscript{Chk}})$ that we achieve w.r.t. the highly optimized execution with runtime assertion checking of [4]. Finally, we record the overheads of the execution times for the original program $R$ with respect to: 1) the monitored execution times for $R$, i.e., the ratio $(T_{R\textsubscript{Chk}} - T_R)/T_R$.
(in column $O_{Chk}$, taken from \cite{4}); and 2) the execution times of the repaired program, i.e., the ratio $(T_{R_{fix}} - T_R)/T_R$ (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 $O_{fix}$).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$O_{fix}$</th>
<th>$O_{Chk}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blocks World</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Container</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Crossing River</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dekker</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maude-NPA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Philosophers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Semaphore Problem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stock Ex.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Webmail app</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wolfram’s Rule 30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: 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. In exchange for that, the correction transformation has a positive impact on the execution times w.r.t. the monitored execution times. As expected, the corrected program $R_{fix}$ is typically slower than the original program $R$; nevertheless, it exhibits a better performance than $R$ when run with the assertion-checking enabled. Indeed, in all cases $T_R \leq T_{R_{fix}} \leq T_{Chk}^R$. Moreover, the average value of the speedups in column $S$ is 0.53, which means that running the corrected program $R_{fix}$ is 53% faster than the monitored execution of $R$.

The overheads $O_{fix}$ and $O_{Chk}$ are graphically compared in Figure 5. 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
Static Correction of Maude Programs with Assertions

8 Related Work and Conclusion

While the research on automated diagnosis of Maude programs has recently made substantial progress (see [3]), effective fault localization and debugging still pose some important challenges. Assertion checking in Maude is a useful contribution of [3], but checking assertions at runtime increases the overall execution time. Automated correction can help in this regard by providing an additional automation layer that includes the generation of verified program corrections. Apart from the handling of concurrency, we give semantic guarantees on the corrections: they do not remove good runs and they reduce the number of failing assertions to zero. Since all assertions are always satisfied after the 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 and improves both the safety and efficiency of the software.

Automated program correction and related problems are not new, with proposed techniques ranging from semantic analysis to stochastic search [16]. 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. [17] use behavioral specifications to fix buggy Java programs, and they use the SAT-based Alloy tool-set to prune any non-determinism that could be introduced by the repair actions. Autofix [18] 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 [16]. 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 [16,19,20] and references therein.

More closely related to our work is the concept of automated program repair of [21], 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
Beyond the technical differences with our work, a more basic difference is in the intention. While [21] aims to automatically produce a collection of fixes for faulty programs that are not necessarily applied in an automatic way, 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.

In [22] 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 finding fixes automatically for programs with incomplete specifications given by system assertions while keeping the transformed program as close as possible to the original one.

**Acknowledgements**

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

**References**


---

10 In [21], repairs cannot be applied automatically since, in general, there is more than one repair possible for a given bug or warning.
A Proofs of Technical Results

Proof (of Proposition 4). Let \( E = (\Sigma, \Delta \cup B) \) be an equational theory. Let \( a \in A \) be a system assertion of the form \( \Pi \mid \varphi \) and \( st \in \tau(\Sigma, V) \) be a system state such that \( (st)↓_{\Delta,B} \not\nmid \Pi \mid \varphi \). Hence, there exist a position \( w \in \mathcal{P}os(st) \) and a substitution \( \sigma \) such that \( ((st)↓_{\Delta,B})|_w =_B \Pi \sigma \) and \( \varphi \sigma \rightarrow_{\Delta,B}^* \text{false} \). By Definition 4 there exists the assertion checking equation

\[
e_{\Pi|\varphi} = (\Pi' = \text{fail} | \not\text{not}(\text{Ren}^{-1}(\varphi))) \in \Delta^A
\]

Now, it is immediate to see that

\[
\text{(Ren}(st)\text{)}|_w e_{\Pi|\varphi,\sigma',w} \rightarrow_{\Delta,A\cup B}^* \text{fail}
\]

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

\begin{itemize}
  \item \( \text{Ren}(st)|_w =_{B} (\text{Ren}(st)|_w)↓_{\Delta,B} =_{B} \Pi' \sigma' \), since \( st|_w =_{B} (st|_w)↓_{\Delta,B} =_{B} \Pi \sigma \), and \( \text{Ren}(st)|_w \) and \( \Pi' \) are respectively the corresponding renamed versions of the canonical form \( (st|_w)↓_{\Delta,B} \) of \( st|_w \) and \( \Pi \) in \( (\Sigma^A, \Delta^A \cup B) \);
  \item \( \not\text{not}(\text{Ren}^{-1}(\varphi))\sigma' = \not\text{not}(\text{Ren}^{-1}(\varphi\sigma')) = \not\text{not}(\varphi\sigma) \rightarrow_{\Delta,A\cup B}^* \text{true} \) because \( \varphi\sigma \rightarrow_{\Delta,B}^* \text{false} \) and \( \sigma' \) only introduces in \( \varphi \) renamed bindings of \( \sigma \) (in fact, \( \sigma' = \{x/t' \mid x/t \in \sigma\} \)).
\end{itemize}

Now, let us prove that if \( (st)↓_{\Delta,B} \not\nmid \Pi|\varphi \), then \( \text{Ren}(st) \rightarrow_{\Delta,A\cup B}^* \text{fail} \). The proof is done by induction on the position \( w \in \mathcal{P}os(\text{Ren}(st)) \) used in step 5 above, i.e.,

\[
\text{Ren}(st)|_w e_{\Pi|\varphi,\sigma',w} \rightarrow_{\Delta,A\cup B}^* \text{fail}.
\]

\( (w = 1) \) Immediate, since \( \text{Ren}(st) = \text{Ren}(st)|_A \) and \( \text{Ren}(st)|_A e_{\Pi|\varphi,\sigma',w} \rightarrow_{\Delta,A\cup B}^* \text{fail} \) by Equation 5.

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

Furthermore, by Definition 4 there exists the equation

\[
f'(x_1, \ldots, x_{i-1}, \text{fail}, x_{i+1}, \ldots, x_n) = \text{fail} \in \Delta^A
\]

Therefore, we can finally build the following rewrite sequence

\[
\text{Ren}(st) = f'(t'_1, \ldots, t'_{i-1}, t'_i, t'_{i+1}, \ldots, t'_n) \rightarrow_{\Delta,A\cup B}^* f'(t'_1, \ldots, t'_{i-1}, \text{fail}, t'_{i+1}, \ldots, t'_n) \rightarrow_{\Delta,A\cup B}^* \text{fail}.
\]
Proof (of Proposition 2). Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a topmost rewrite theory, $\mathcal{A}$ be an assertion set, and $\mathcal{R}' = (\Sigma^A, \Delta^A \cup B, R^A)$ be the $\mathcal{A}$-extension of $\mathcal{R}$. By Definition 2 in order to show that $\mathcal{R}' \leq A \mathcal{R}$ (i.e., $\mathcal{R}'$ is a correction of $\mathcal{R}$ w.r.t. $\mathcal{A}$), we have to prove that

1. 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. $s_i \models \mathcal{A}$, $i = 0, \ldots, n$
2. 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 $C^A = (s_0 \rightarrow_{\mathcal{R}'} \ldots \rightarrow_{\mathcal{R}'} s_n)$ in $\mathcal{R}' = (\Sigma^A, \Delta^A \cup B, R^A)$ such that $s_0 \models \mathcal{A}$. We proceed by induction on the length $n$ of the computation $C^A$.

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

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

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

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

$$C^A_{n-1} = (s_0 \rightarrow_{\mathcal{R}} \ldots \rightarrow_{\mathcal{R}} s_{n-1}),$$

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 $s_{n-1} \rightarrow_{\mathcal{R}'} s_n$ as follows

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

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

Therefore,

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

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

$$s_{n-1} \xrightarrow{r^A, \sigma, w} R^A, B \rho \sigma$$

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

$$\text{Ren}(\rho \sigma) \not\rightarrow_{\Delta^A, B}^* \text{fail}$$

which implies that $(\rho \sigma) \downarrow_{\Delta, B} \models \mathcal{A}$ by Proposition 1.
[Claim ii] We consider an arbitrary computation \( C = (s_0 \rightarrow_R \ldots \rightarrow_R s_n) \) s.t. \( s_i \models A_i, \ i = 0, \ldots, n \). Similarly to the proof of Claim i, we just proceed by induction on the length \( n \) of the computation \( C \) to show that there exists \( s_0 \rightarrow_{R'} \ldots \rightarrow_{R'} s_n \). Again, the base case is trivial, while the inductive case is similar to the inductive case of [Claim i] by making use of Proposition 1.

**Proof (of Proposition 3).** The case when \( Ax=ACU \) has been stated in Lemma 5.3 of [10]. Here, we prove the case when \( Ax=AC \), which involves two extension rewrite rules, namely, \( \{X \otimes \lambda \} \Rightarrow \{X \otimes \rho \} \) if \( C \) and \( \{\lambda \} \Rightarrow \{\rho \} \) if \( C \). The remaining cases are straightforward adaptations of the following proof scheme.

We have to prove the following two implications for the case when \( Ax=AC \):

1. \((\rightarrow)\) for any term \( t_i \) and \( t_f \) of sort \( \text{Config} \), \( t_i \rightarrow_R^* t_f \) implies \( \{t_i\} \rightarrow_R^* \{t_f\} \);
2. \((\leftarrow)\) for any term \( t_i \) and \( t_f \) of sort \( \text{Config} \), \( \{t_i\} \rightarrow_R^* \{t_f\} \) implies \( t_i \rightarrow_R^* t_f \).

\((\rightarrow)\) Assume that \( t_i \rightarrow_R^* t_f \), where \( t_i \) and \( t_f \) are arbitrary terms of sort \( \text{Config} \).

Then, \( t_i \rightarrow_R^* t_f \) has the form

\[
t_i = t_0 \rightarrow_R \ldots \rightarrow_R t_{n-1} \rightarrow_R t_n = t_f, \text{ for some natural number } n \geq 0.
\]

We proceed by induction on the length \( n \) of the rewriting sequence \( t_i \rightarrow_R^* t_f \).

- **n = 0.** Immediate, since there are no rewrite steps.
- **n > 0.** By induction hypothesis, we have

\[
t_i = t_0 \rightarrow_R^* t_{n-1} \text{ implies } \{t_i\} = \{t_0\} \rightarrow_R^* \{t_{n-1}\}. \quad (6)
\]

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

\[
\{t_{n-1}\} \rightarrow_R \{t_n\}, \quad (7)
\]

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

\[
t_{n-1} \rightarrow_{R,B} \hat{t}_{n-1} \rightarrow_{\Delta,B}^* t_{n-1} \downarrow_{\Delta,B} = t_n
\]

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

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

\[
\{t_{n-1}\} \equiv_B \{\lambda \sigma\} \rightarrow_{R,B}^* \{\rho \sigma\} = B \{t_{n-1}\} \rightarrow_{\Delta,B}^* \{t_{n-1}\} \downarrow_{\Delta,B} = \{t_n\}
\]

with \( \hat{r} = (\lambda \Rightarrow \rho \text{ if } C) \in \hat{R} \). Hence, \( \{t_{n-1}\} \rightarrow_R \{t_n\} \) when \( w = \Lambda \).
\( (w \neq A) \) Since \( \mathcal{R} \) is topmost modulo \( \mathcal{A} \) and \( w \neq A \), there exist \( s_i \in \tau(\Sigma, \mathcal{V})_{\text{Config}}, i = 1, \ldots, k \), with \( k > 1 \) such that

\[
\tau_{n-1} = s_1 \otimes \ldots \otimes s_k
\]

and \( \tau_{n-1} \in \tau(\Sigma, \mathcal{V})_{\text{Config}} \). Now, since \( \tau_{n-1} \xrightarrow{r,\sigma,w}_{R,B} \tau_{n-1} \xrightarrow{\lambda} \mathcal{A} \mathcal{B} \tau_n \), with \( \mathcal{A} = \mathcal{A} \mathcal{C} \) and \( r = (\lambda \Rightarrow \rho) \) if \( C \),

\[
\tau_{n-1} = s_1 \otimes \ldots \otimes s_k \xrightarrow{r,\sigma,w}_{R,B} \tau_{n-1} \xrightarrow{s_{\pi(1)} \otimes \ldots \otimes s_{\pi(m)} \otimes \lambda} \mathcal{A} \mathcal{B} \tau_n
\]

where \( 1 \leq m \leq k-1 \), and \( \pi: \{1, \ldots, m\} \rightarrow \{1, \ldots, k\} \) 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

\[
\{\tau_{n-1}\} = \{s_1 \otimes \ldots \otimes s_k\} \xrightarrow{r,\sigma,w}_{R,B} \{s_{\pi(1)} \otimes \ldots \otimes s_{\pi(m)} \otimes \lambda\} \xrightarrow{s_{\Delta}} \mathcal{A} \mathcal{B} \{\tau_n\}
\]

with \( \hat{\sigma} = \sigma \cup \{X/s_{\pi(1)} \otimes \ldots \otimes s_{\pi(m)}\} \), and \( \hat{r} = (\{X \otimes \lambda\} \Rightarrow \{X \otimes \rho\}) \) if \( C \in \hat{R} \). This proves that \( \{\tau_{n-1}\} \xrightarrow{\hat{r}} \{\tau_n\} \) also in the case when \( w \neq A \).

Finally, by using the induction hypothesis \( \downarrow \) and the rewrite step \( \uparrow \) we easily derive the implication \((\rightarrow)\).

\((\leftarrow)\) Assume that \( \{t_i\} \xrightarrow{s_{\Delta}} \{t_f\} \), where \( t_i \) and \( t_f \) are arbitrary terms of sort \( \text{Config} \). Then, \( \{t_i\} \xrightarrow{s_{\Delta}} \hat{R} \{t_f\} \) is of the form

\[
\{t_i\} = \{t_0\} \xrightarrow{s_{\Delta}} \hat{R} \{t_{n-1}\} \xrightarrow{s_{\Delta}} \hat{R} \{t_n\} = \{t_f\}
\]

for some natural number \( n \geq 0 \). We proceed by induction on the length \( n \) of the computation \( \{t_i\} \xrightarrow{s_{\Delta}} \hat{R} \{t_f\} \).

\( n = 0 \). Immediate, since there are no rewrite steps.

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

\[
\{t_i\} = \{t_0\} \xrightarrow{s_{\Delta}} \hat{R} \{t_{n-1}\} \text{ implies } t_i = t_0 \xrightarrow{s_{\Delta}} \hat{R} t_{n-1}. \tag{8}
\]

Therefore, it suffices to show that \( t_{n-1} \xrightarrow{s_{\Delta}} \hat{R} t_n \) and combine this result with the induction hypothesis to finally prove Case \((\leftarrow)\).

By hypothesis, \( \{t_{n-1}\} \xrightarrow{s_{\Delta}} \hat{R} \{t_n\} \), which can be expanded into the following rewrite sequence

\[
\{t_{n-1}\} \xrightarrow{r,\sigma,w}_{R,B} \{\tilde{t}_{n-1}\} \xrightarrow{s_{\Delta}} \mathcal{A} \mathcal{B} \{t_n\} \tag{9}
\]

where \( \tilde{r} \in \hat{R} \), and \( \{t_{n-1}\}, \{\tilde{t}_{n-1}\}, \{t_n\} \in \tau(\hat{\Sigma}, \hat{\mathcal{V}})_{\text{State}} \). Observe that the first rewrite step of the rewrite sequence \( \{9\} \) must occur at position \( \Lambda \), since the rewrite theory \( \hat{R} \) is topmost.
Here, we distinguish two cases according to the form of the rewrite rule \( \hat{r} \in \hat{R} \) applied in \( \{ t_{n-1} \} \xrightarrow{\hat{r}, \hat{A}} R, B \{ \hat{t}_{n-1} \} \).

By Definition 9, \( \hat{r} \) is either \( \{ \lambda \} \Rightarrow \{ \rho \} \) if \( C \) or \( \{ X \otimes \lambda \} \Rightarrow \{ X \otimes \rho \} \) if \( C \), as \( Ax = AC \) and \( \{ t_{n-1} \}, \{ \hat{t}_{n-1} \} \in \tau(\Sigma, \mathcal{V})_{\text{State}} \).

**Case** \( (\hat{r} = (\{ \lambda \} \Rightarrow \{ \rho \} \text{ if } C)) \). In this case, \( \{ t_{n-1} \} \xrightarrow{\hat{r}, \hat{A}} R, B \{ \hat{t}_{n-1} \} \) assumes the following form:

\[
\{ t_{n-1} \} = AC \{ \lambda \sigma \} \xrightarrow{\hat{r}, \hat{A}} R, B \{ \rho \sigma \} = AC \{ \hat{t}_{n-1} \}.
\]

Now, by Definition 9, \( \lambda \sigma \) and \( \rho \sigma \) are terms of sort \( \text{Config} \); thus, we can also apply \( r = (\lambda \Rightarrow \rho \text{ if } C) \in R \) to \( \lambda \sigma \), thereby obtaining the following computation

\[
t_{n-1} = AC \lambda \sigma \xrightarrow{r, \sigma} R, B \rho \sigma \xrightarrow{*_{\Delta, B}} (\rho \sigma \downarrow_{\Delta, B}) = t_n
\]

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

**Case** \( (\hat{r} = (\{ X \otimes \lambda \} \Rightarrow \{ X \otimes \rho \} \text{ if } C)) \). In this case, \( \{ t_{n-1} \} \xrightarrow{\hat{r}, \hat{A}} R, B \{ \hat{t}_{n-1} \} \) must have the following form:

\[
\{ t_{n-1} \} = AC \{ c \otimes \lambda \hat{\sigma} \} \xrightarrow{\hat{r}, \hat{A}} R, B \{ c \otimes \rho \hat{\sigma} \} = AC \{ \hat{t}_{n-1} \}
\]

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

Now, by Definition 9, variable \( X \) does not occur in either \( \lambda \) or \( \rho \); this implies that \( \lambda \hat{\sigma} = \lambda \sigma \) and \( \rho \hat{\sigma} = \rho \sigma \). Therefore, we can construct the following computation:

\[
t_{n-1} = AC c \otimes \lambda \hat{\sigma} = c \otimes \lambda \sigma \xrightarrow{r, \sigma} R, B c \otimes \rho \sigma = c \otimes \rho \hat{\sigma} = AC \{ \hat{t}_{n-1} \} \rightarrow \Delta, B \ t_n
\]

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

**Proof (of Corollary 7).** Immediate by applying Proposition 2 to the topmost rewrite theory \( \mathcal{R} = (\Sigma, \Delta, \mathcal{A}, \mathcal{H}, R) \).

**Proof (of Corollary 8).** The rewrite theory \( \mathcal{R}' \) is topmost, so the result is immediate by Proposition 2.