Alpuente Frasnedo, M.; Ballis, D.; Frechina Navarro, F.; Sapiña Sanchis, J. (2015). Combining Runtime Checking and Slicing to Improve Maude Error Diagnosis. En Logic, Rewriting, and Concurrency. Essays Dedicated to José Meseguer on the Occasion of His 65th Birthday. 72-96. doi:10.1007/978-3-319-23165-5_3.

Combining Runtime Checking and Slicing to improve Maude Error Diagnosis*

M. Alpuente[1], D. Ballis[2], F. Frechina[1], and J. Sapiña[1]

[1] DSIC-ELP, Universitat Politècnica de València
Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain
{alpuente,ffrechina,jsapina}@dsic.upv.es
[2] DIMI, University of Udine,
Via delle Scienze, 206, 33100, Udine, Italy
demis.ballis@uniud.it

Abstract. This paper introduces the idea of using assertion checking for enhancing the dynamic slicing of Maude computation traces. Since trace slicing can greatly simplify the size and complexity of the analyzed traces, our methodology can be useful for improving the diagnosis of erroneous Maude programs. The proposed methodology is based on (i) a logical notation for specifying two types of user-defined assertions that are imposed on execution runs: functional assertions and system assertions; (ii) a runtime checking technique that dynamically tests the assertions and is provably safe in the sense that all errors flagged are definite violations of the specifications; and (iii) a mechanism based on equational least general generalization that automatically derives accurate criteria for slicing from falsified assertions.

1 Introduction

Back in the mid-80s, while the scientific research in Computer Science was just taking off in Spain, some magnetic manuscripts written by Joseph Goguen and José Meseguer came into our hands [17,18]. Although some predicted that no language as ambitious as that described in [17,18,25] could reach widespread or practical use, these ground-breaking documents were, for many of us, the starting point for pursuing the advancement of multi-paradigm declarative languages and their development environments. Towards this endeavor, the aim of this work is to contribute to further advancing the state-of-the-art of the leading-edge multi-paradigm language Maude.

Assertion checking is the problem of deciding whether a certain assertion holds at a given program (or execution) point. Although not universally used, assertion checking seems to have widely infiltrated common programming practice

*This work has been partially supported by the EU (FEDER) and the Spanish MINECO project ref. TIN2013-45732-C4-01 (DAMAS), and by Generalitat Valenciana ref. PROMETEOII/2015/013 (SmartLogic). F. Frechina was supported by FPU-ME grant AP2010-5681, and J. Sapiña was supported by FPI-UPV grant SP2013-0083.

as witnessed by the growth of assertion capabilities in widely used programming languages such as C♯, C++, and Java [12]. Assertions may be used statically to support program analysis and also for secondary purposes, such as documentation and to provide information to an optimizer during code generation. A brief history of the research ideas that have contributed to the assertion capabilities of modern programming languages and development tools can be found in [12]. The most obvious way to dynamically use assertions is to test them at runtime and report any detected violations. By finding inconsistencies between asserted properties and the program code, runtime assertion checking can be used to reveal program faults and to obtain information about their locations. Since an assertion failure usually reports an error, the user can direct its attention to the location at which the logical inconsistency is detected and (hopefully) trace the errors back to their sources more easily.

Program slicing [20] automatically identifies a subset of program statements that either (i) contribute to the values of a set of variables at a given point, or (ii) are influenced by the values of a given set of variables. The first approach corresponds to forms of backward slicing, whereas the second corresponds to forward slicing. Automatic slicing plays an important role in program diagnosis and understanding since it allows one to focus on code fragments that are relevant to a given slicing criterion, that is, the relevant information we want to track (backwards or forwards) from a given execution point.

Maude [13] is a high-level language and high-performance system that supports both equational and rewriting logic computations. Maude modules correspond to specifications in rewriting logic [24], which is a logic that allows the representation of many models of concurrent and distributed systems. In [6,8], a rich and highly dynamic parameterized scheme for exploring rewriting logic computations is developed that can significantly reduce the size and complexity of the runs under examination by automatically slicing both programs and computation traces [4].

The aim of this work is to provide Maude with runtime assertion-checking capabilities by first introducing a simple assertion language that suffices for the purpose of improving error diagnosis and debugging of Maude programs, while remaining tractable. We follow the approach of modern specification and verification systems such as Spec♯ or the Java Modeling Language (JML) where the specification language is typically an extension of the underlying programming language and specifications are used as *contracts* that guarantee certain properties to hold at a number of execution states, e.g., before or after a given function call [22]. We believe that this choice of a language is of practical interest because it facilitates the job of programmers. Even if Maude is a highly declarative language that supports a programming style where no conceptual difference exists between programs and high-level specifications, a separate description given by the assertions may help developers identify essential program behaviors to be preserved when modifying code.

We distinguish two groups of assertions: 1) functional assertions, for specifying properties of functions defined by an equational theory; and 2) system

assertions, which allow one to express properties concerning the system's execution. The assertions we support allow us to express properties that are quite general, including user–defined programs. However, for the functional assertions, we require the user to ensure that the execution of any property terminates for any possible initial state and that the resulting verdict is unique, in the same spirit of Maude's (canonical) equational theories. In the proposed framework, if an assertion evaluates to false at runtime, an assertion failure results, which typically causes execution to abort while delivering a huge execution trace. By automatically inferring deft slicing criteria from falsified assertions, we derive a self-initiating, enhanced dynamic slicing technique that automatically starts slicing the trace backwards at the time the assertion violation occurs, without having to manually determine the slicing criterion in advance. As a by-product of the trace slicing process, we also compute a dynamic program slice that preserves the program behavior for the considered program inputs [20].

The Maude Formal Environment (MFE) is a recent effort to integrate and interoperate most of the available Maude analysis and verification tools. These include among others an inductive theorem prover, a declarative debugger, and Maude's model checkers [23]. Maude supports strong typing and subtyping assertions via *membership axioms*, which are used to automatically 'narrow' the type T of a value into a subtype of T. Nevertheless, to the best of our knowledge, no general built-in support is provided in Maude or the MFE for the runtime checking of user-defined assertions. Related to our work, generic strategies are defined in [16,28] to guarantee that a set of invariants (that can be expressed in different logics) are satisfied at every computed state. This is achieved by avoiding the execution of actions that otherwise would conduct the system to states that do not satisfy the constraints. This is in contrast to our approach in two ways. On the one hand, our assertions are *external* and evaluated at runtime, whereas driving the system's execution in such a way that every computation state complies with the constraints makes the assertions *internal* to the programmed strategy. On the other hand, the strategy of [16,28] never results in violated assertions, which is essential for automatic trace slicing to be fired according to our approach. As another difference, we are able to check assertions that regard: 1) the normalizations carried out by using the equational part of the rewriting theory; and 2) system properties that are not necessarily global invariants but can only hold in those states that match a given state template.

Following the discussion above, this work can be seen as the first framework that exploits the synergies we can find between runtime assertion checking and automated (program and program trace) transformations for improving the diagnosis of Maude programs.

Plan of the paper The paper is organized as follows. Section 2 provides a brief introduction to rewriting logic and Maude and introduces the running example that we use throughout the paper: a conditional rewrite theory that models a simple, distributed banking system. Section 3 introduces a very simple assertion language and the notions of functional and system assertions whose violation helps signal functional and system *error symptoms*. Section 4 recalls a trace slic-

ing methodology for simplifying rewriting logic computations. Section 5 enriches the slicing methodology with runtime assertion checking in order to improve the diagnosis of erroneous Maude programs, and describes its implementation in the ABETS tool. Section 6 concludes. More details and examples, and a thorough comparison with the related literature, can be found in an extended version of this article, which is available at [7].

2 Rewriting Logic and Maude

Let us recall some important notions that are relevant to this work. We assume some basic knowledge of term rewriting [29] and rewriting logic [24] (RWL). Some familiarity with the Maude language [14,13] is also required. Throughout the paper, Maude notation will be introduced "on the fly" as required.

2.1 Preliminaries

Let Σ be a *signature* that allows operators to be specified together with their type structure by means of suitable sets of sorts and kinds. By $\tau(\Sigma)$, we specify the term algebra that includes all the ground terms built over Σ, while $\tau(\Sigma, \mathcal{V})$ is the usual nonground term algebra built over Σ and the set of variables $\mathcal{V}$. Each operator in Σ is defined along with its sort and axiom declarations that may specify algebric laws such as associativity (`assoc`), commutativity (`comm`), and identity (`id`).

A *position* w in a term t is represented by a sequence of natural numbers that addresses a subterm of t (Λ denotes the empty sequence, i.e., the root position). Given a term t, we let $\mathcal{P}os(t)$ denote the set of positions of t. By $t_{|w}$, we denote the *subterm* of t at position w, and by $t[s]_w$, we denote the result of *replacing the subterm* $t_{|w}$ by the term s in t.

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 over a set of variables $\{x_1, \ldots, x_n\}$. By $\{\}$, we denote the *identity* substitution. The application of a substitution σ to a term t, denoted $t\sigma$, is defined by induction on the structure of terms as usual [10]. Given two terms t and t', we say that t is *more general* than t' iff there exists a substitution σ such that $t\sigma = t'$. We also say that t' is an *instance* of t.

Given a syntactic expression e, by $\mathcal{V}ar(e)$, we denote the set of variables that occur in e. Given a binary relation $\rightsquigarrow$, we define the usual *transitive* (resp., *transitive and reflexive*) closure of $\rightsquigarrow$ by $\rightsquigarrow^+$ (resp., $\rightsquigarrow^*$).

2.2 Rewrite Theories and Maude Modules

The static state structure as well as the dynamic behavior of a concurrent system can be formalized as a RWL specification that encodes a *conditional rewrite theory*. More specifically, a *conditional rewrite theory* (or simply *rewrite theory*) is a triple $\mathcal{R} = (\Sigma, E, R)$, where:

(i) (Σ, E) is a membership equational theory that allows us to define the system data types via equations, as well as algebraic and membership axioms. Σ is a signature that specifies the operators of $\mathcal{R}$, while $E = \Delta \cup B$ is the disjoint union of the set Δ, which contains conditional equations and conditional membership axioms, and the set B, which contains algebraic axioms associated with binary operators in Σ. The general Maude syntax of conditional equations and membership axioms is the following:

$$\texttt{ceq [}l\texttt{] : }\lambda\texttt{ = }\rho\texttt{ if }C\texttt{ .} \qquad \texttt{cmb [}l\texttt{] : }\lambda\texttt{ : }s\texttt{ if }C\texttt{ .}$$

where l is a label (i.e., a name that identifies the equation), $\lambda, \rho \in \tau(\Sigma, \mathcal{V})$, s is a sort and C is an *equational* condition, that is, a (possibly empty) conjunction of equations $\mathtt{t} = \mathtt{t}'$, matching equations $\mathtt{p} := \mathtt{t}$, and memberships $\mathtt{t} : \mathtt{s}'$ that is built using the binary conjunction connective `/\`, which is assumed to be associative. When C is empty, the syntax for equations and memberships is simplified as follows:

$$\texttt{eq [}l\texttt{] : }\lambda\texttt{ = }\rho\texttt{ .} \qquad \texttt{mb [}l\texttt{] : }\lambda\texttt{ : }s\texttt{ .}$$

A membership equational theory (Σ, E) is encoded in Maude through a *functional module* that is syntactically delimited by keywords `fmod` and `endfm`. Functional modules provide executable models for the specified equational theories.

Example 1. The following Maude functional module[1] encodes an equational theory that defines the functional part of a simple, distributed banking system.

```
fmod BANK-EQ is inc BANK-INT+ID . pr SET{Id} .
  sorts Account PremiumAccount Status Msg State .
  subsort PremiumAccount < Account .
  subsorts  Account Msg < State .

  var ID : Id .           op <_|_|_> : Id Int Status -> Account [ctor] .
  var BAL : Int .         op active : -> Status [ctor] .
  var STS : Status .      op blocked : -> Status [ctor] .

  op Alice : -> Id [ctor] .          op Bob : -> Id [ctor] .
  op Charlie : -> Id [ctor] .        op Daisy : -> Id [ctor] .

  cmb < ID | BAL | STS > : PremiumAccount if ID in PreferredClients .

  op PreferredClients : -> Set{Id} .
  eq PreferredClients = Bob, Charlie .

  op updateStatus : Account -> Account .
  ceq updateStatus(< ID | BAL | active >) = < ID | BAL | blocked >
      if BAL < 0 .
  eq updateStatus(< ID | BAL | STS >) = < ID | BAL | STS > [owise] .
endfm
```

[1] `BANK-EQ` includes the functional module `BANK-INT+ID`, which (i) imports `INT` for integer manipulation and (ii) declares the sort `Id` that is used to parameterize `SET{X :: TRIV}`.

A bank account is represented as a term of the form `< ID | BAL | STS >` where `ID` is the owner of the account, `BAL` is the account balance, and `STS` is the account status, which can be `blocked` or `active`. The defined conditional membership axiom states that an account is a `PremiumAccount` if its owner is included in the `PreferredClients` set. Finally, the `updateStatus` operation updates the status account to `blocked` when the account balance is negative.

(ii) R is a set of conditional labeled rules whose Maude syntax is the following:

$$\texttt{crl [}l\texttt{] : } \lambda \texttt{ => } \rho \texttt{ if } C \texttt{ .}$$

where l is a label, $\lambda, \rho \in \tau(\Sigma, \mathcal{V})$, and C is a *rule* condition, i.e., an equational condition that may also contain rewrite expressions of the form $\mathtt{t} = \mathtt{t}'$. When a rule has no condition, we simply write `rl [`l`] : `λ` => `ρ` .`

A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is specified in Maude by means of a *system module*, which is introduced by the syntax `mod...endm`. A system module may include both a functional representation of the equational theory (Σ, E) and the specification of the rewrite rules in R.

Example 2. The following Maude rewrite theory models the distributed behavior of the banking system of Example 1.

```
mod BANK is inc BANK-EQ .
  vars ID ID1 ID2 : Id .
  vars BAL BAL1 BAL2 M : Int .

  op empty-state : -> State [ctor] .
  op _;_ : State State -> State [ctor assoc comm id: empty-state] .
  ops credit debit : Id Int -> Msg [ctor] .
  op  transfer : Id Id Int -> Msg  [ctor] .

  rl [credit] : credit(ID,M) ; < ID | BAL | active > =>
                updateStatus(< ID | BAL + M | active >) .
  rl [debit] : debit(ID,M) ; < ID | BAL | active > =>
                updateStatus(< ID | BAL - M | active >) .
  rl [transfer] : transfer(ID1,ID2,M) ;
                  < ID1 | BAL1 | active > ; < ID2 | BAL2 | active >
                => updateStatus(< ID1 | BAL1 - M | active >) ;
                   updateStatus(< ID2 | BAL2 + M | active >) .
endm
```

Each state of the system is modeled as a multiset (i.e., an associative and commutative list) of elements of the form $\mathtt{e}_1; \mathtt{e}_2; \ldots ...; \mathtt{e}_n$. Each element $\mathtt{e}_i$ is either (i) a bank account; or (ii) a message modeling a debit, credit, or transfer operation. These account operations are implemented via three rewrite rules: namely, `debit`, `credit`, and `transfer` rules.

2.3 Rewriting and Generalization modulo Equational Theories

Let us consider a conditional rewrite theory (Σ, E, R), with $E = \Delta \cup B$, where Δ is a set of conditional equations and membership axioms, and B is a set of

equational axioms associated with some binary operators in Σ. The conditional rewriting modulo E relation (in symbols, $\rightarrow_{R/E}$) can be defined by lifting the usual conditional rewrite relation on terms [19] to the E-congruence classes $[t]_E$ on the term algebra $\tau(\Sigma, \mathcal{V})$ that are induced by $=_E$ [11]. In other words, $[t]_E$ is the class of all terms that are equal to t *modulo* E. Unfortunately, $\rightarrow_{R/E}$ is, in general, undecidable since a rewrite step $t \rightarrow_{R/E} t'$ involves searching through the possibly infinite equivalence classes $[t]_E$ and $[t']_E$.

The Maude interpreter implements conditional rewriting modulo E by means of two much simpler relations, namely $\rightarrow_{\Delta,B}$ and $\rightarrow_{R,B}$. These allow rules, equations and memberships to be intermixed in the rewriting process by simply using an algorithm of matching modulo B. We define $\rightarrow_{R\cup\Delta,B}$ as $\rightarrow_{R,B} \cup \rightarrow_{\Delta,B}$. Roughly speaking, the relation $\rightarrow_{\Delta,B}$ uses the equations of Δ (oriented from left to right) as simplification rules. Thus, by repeatedly applying the equations as simplification rules from a given term t, we eventually reach a term $t\downarrow_{\Delta,B}$ to which no further equations can be applied. The term $t\downarrow_{\Delta,B}$ is called a *canonical (or normal)* form of t w.r.t. Δ modulo B. An *equational simplification* of a term t in Δ modulo B is a rewrite sequence of the form $t \rightarrow^*_{\Delta,B} t\downarrow_{\Delta,B}$. Informally, the relation $\rightarrow_{R,B}$ implements rewriting with the rules of R, which might be non-terminating and non-confluent, whereas Δ is required to be terminating and Church-Rosser modulo B in order to guarantee the existence and unicity (modulo B) of a canonical form w.r.t. Δ for any term [14]. Terms are rewritten into canonical forms according to their sort structure, which is induced by the signature Σ and the membership axioms specified in Δ. In particular, through membership axioms of the form `cmb [l] : λ : s if C`, we can assert that any term B-matching λ has a specific sort `s` whenever a condition `C` holds. Equational simplification of terms is naturally lifted to substitutions as follows: given $\sigma = \{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$, we define the *normalized* substitution $\sigma\downarrow_{\Delta,B} = \{x_i/(t_i\downarrow_{\Delta,B})\}_{i=1}^n$.

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

Roughly speaking, a conditional rewrite step on the term t applies a rewrite rule/equation to t by replacing a *red*ucible (sub-)*ex*pression of t (namely $t_{|w}$), called the *redex*, by its contracted version $\rho\sigma$, called the *contractum*, whenever the condition `C` is fulfilled. Note that the evaluation of a condition `C` is typically a recursive process since it may involve further (conditional) rewrites in order to normalize `C` to *true*. Specifically, an equation `e` *evaluates to true* w.r.t. σ if $\mathtt{e}\sigma\downarrow_{\Delta,B} =_B$ *true*; a matching equation `p := t` *evaluates to true* w.r.t. σ if $\mathtt{p}\sigma =_B \mathtt{t}\sigma\downarrow_{\Delta,B}$; a rewrite expression `t ⇒ p` *evaluates to true* w.r.t. σ if there

exists a rewrite sequence $\mathtt{t}\sigma \rightarrow^*_{R\cup\Delta,B} \mathtt{u}$, such that $\mathtt{u} =_B \mathtt{p}\sigma$[2]; and, finally, a membership $\mathtt{t} : \mathtt{s}$ *evaluates to true* w.r.t. σ if $\mathtt{t}\sigma$ has sort $\mathtt{s}$.

Under appropriate conditions on the rewrite theory, a rewrite step $s \rightarrow_{R/E} t$ modulo E on a term s can be implemented without loss of completeness by applying a rewrite strategy that first simplifies the term s into its canonical form $s\downarrow_{\Delta,B}$, and then applies a rule $r \in R$ to $s\downarrow_{\Delta,B}$ [15].

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

$$s_0 \rightarrow^*_{\Delta,B} s_0\downarrow_{\Delta,B} \rightarrow_{R,B} s_1 \rightarrow^*_{\Delta,B} s_1\downarrow_{\Delta,B}\rightarrow_{R,B} \cdots$$

that interleaves $\rightarrow_{\Delta,B}$ rewrite steps and $\rightarrow_{R,B}$ rewrite steps following the strategy mentioned above. After each conditional rewriting step using $\rightarrow_{R,B}$, in general, the resulting term s_i, $i = 1, \ldots, n$, is not in canonical normal form. Therefore, it is normalized before the subsequent rewrite step with $\rightarrow_{R,B}$ 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. By ε, we denote the *empty* computation.

We define a *Maude step* from a given term s as any of the sequences $s \rightarrow^*_{\Delta,B} s\downarrow_{\Delta,B} \rightarrow_{R,B} t \rightarrow^*_{\Delta,B} t\downarrow_{\Delta,B}$ that head the non-deterministic Maude computations for s. Note that, for a canonical form s, a Maude step for s boils down to $s \rightarrow_{R,B} t \rightarrow^*_{\Delta,B} t\downarrow_{\Delta,B}$. We define $m\mathcal{S}(s)$ as the set of all the non-deterministic Maude steps from s.

A *generalization* of a pair of terms t_1, t_2 is a triple (g, θ_1, θ_2) such that $g\theta_1 = t_1$ and $g\theta_2 = t_2$. The triple (g, ϕ_1, ϕ_2) is the *least general generalization (lgg)* of the pair of terms t_1, t_2, written $lgg(t_1, t_2)$, if (1) (g, ϕ_1, ϕ_2) is a generalization of t_1, t_2 and (2) for every other generalization (g', ψ_1, ψ_2) of t_1, t_2, g' is more general than g. The *lgg* of a pair of terms is unique up to variable renaming [21].

In [9], the notion of least general generalization is extended to work modulo (order-sorted) equational theories, where function symbols can obey any combination of associativity, commutativity, and identity axioms (including the empty set of such axioms). Unlike the untyped case, for a pair of terms t_1, t_2 there is generally no single lgg, due to order-sortedness or to the equational axioms. Instead, there is a finite, minimal, and complete set of lggs (denoted by $lgg_E(t_1, t_2)$) so that any other equational generalizer has at least one of them as an instance. Given any element (g, ϕ_1, ϕ_2) of the set $lgg_E(t_1, t_2)$, we define the function π from $\mathcal{P}os(t)$ to $\mathcal{P}os(t_1)$ that provides an injective correspondence between (the position of) any variable in g and (the position of) the corresponding term in t_1; we need this because computing modulo algebraic axioms may cause the term structure of g to be different from both, t_1 and t_2. For instance, consider an associative and commutative symbol $\mathtt{f}$ and the terms $t_1 = \mathtt{f}(\mathtt{b}, \mathtt{c}, \mathtt{a})$ and $t_2 = \mathtt{f}(\mathtt{d}, \mathtt{a}, \mathtt{b})$. Then, a possible lgg modulo the associativity and commutativity

[2] Technically, to properly evaluate a rewrite expression $\mathtt{t} \Rightarrow \mathtt{p}$ or a matching condition $\mathtt{p} := \mathtt{t}$, the term $\mathtt{p}$ is required to be a Δ-pattern modulo B (i.e., a term $\mathtt{p}$ such that, for every substitution σ, if $x\sigma$ is a canonical form w.r.t. Δ modulo B for every $x \in Dom(\sigma)$, then $\mathtt{p}\sigma$ is also a canonical form w.r.t. Δ modulo B).

of f is $(\mathtt{f}(\mathtt{a},\mathtt{b},\mathtt{X}),\{\mathtt{X}/\mathtt{c}\},\{\mathtt{X}/\mathtt{d}\}) \in lgg_E(t_1,t_2)$, where X is a variable. Note that both t_1 and t_2 are syntactically different from $\mathtt{f}(\mathtt{a},\mathtt{b},\mathtt{X})$, and the value $\pi(3)=2$ indicates the subterm c of t_1 that is responsible for the mismatch with t_2. By $\widehat{lgg}_E(t_1,t_2)$ we denote the pair (G,π) where $G=(g,\phi_1,\phi_2)$ is arbitrarily chosen among those lggs in the set $lgg_E(t_1,t_2)$ that have fewer variables, and π is the corresponding position mapping from positions of g's variables to the relative subterms of t_1.

One of the main motivations of our work is to help automate as much as possible the validation and debugging of programs with respect to properties that are outside of Maude's typing system. Some of the properties we consider can arguably be expressed by means of sorts and memberships in Maude. Nevertheless, in the following section we deal with properties that these facilities cannot handle.

3 The Assertion Language

Assertions are linguistic constructions that formally express properties of a software system. Throughout this section, we consider a software system that is specified by a rewrite theory $\mathcal{R}=(\Sigma,\Delta\cup B,R)$. Without loss of generality, we assume that Σ includes at least the sort State. Terms of sort State are called *system states* (or simply *states*). State transitions are obtained by nondeterministically applying the rewrite rules in R to canonical forms of system states. A state s is simplified into its canonical form $s\downarrow_{\Delta,B}$ by using equations and algebraic/membership axioms in $\Delta\cup B$.

In our specification language, assertions are formulas built on user-defined functions. The meaning of such functions is specified by a user-defined program. Our framework supports two kinds of assertions: *functional* assertions and *system* assertions. Functional assertions allow properties to be logically defined on the equational component of the rewrite theory $\mathcal{R}$ while system assertions specify formal constraints on the possibly nondeterministic rule component of $\mathcal{R}$. The benefit of using a logic framework is that the definition and checking of all asserted properties can be performed in a uniform and familiar setting.

3.1 The Assertion Logic

The core of our assertion language is based on (order-sorted) predicate logic, where first order formulas are built over the signature Σ of the rewrite theory $\mathcal{R}$ enriched with a set of user-defined boolean function symbols (predicates). The truth values are given by the formulas true and false. 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.

Logic formulas can be defined in Maude by means of the predefined functional module BOOL [14], which specifies the built-in sort Bool, the truth values, the logic operators, and the built-in operators for membership predicates _:: S for each sort S, and term equality _==_ and inequality _=/=_.

```
mod BANK-PRED is inc BANK .
  var ACC : Account .            op isPremium : Account -> Bool .
  var ID : Id .                  op getBalance : Account -> Int .
  var BAL : Int .                op getId : Account -> Id .
  var STS : Status .             op getStatus : Account -> Status .

  ceq isPremium(ACC) = true if ACC : PremiumAccount .
  eq isPremium(ACC) = false [owise] .

  eq getBalance(< ID | BAL | STS >) = BAL .
  eq getId(< ID | BAL | STS >) = ID .
  eq getStatus(< ID | BAL | STS >) = STS .
endm
```

Figure 1. System properties specified by the `BANK-PRED` module.

The built-in Boolean functions `_==_` and `_=/=_` have a straightforward operational meaning: given an expression `u == v`, then both `u` and `v` are simplified by the equations in the module (which are assumed to be Church-Rosser and terminating) to their canonical forms (perhaps modulo some axioms such as associativity) and these canonical forms are compared for equality. If they are equal, the value of `u == v` is `true`; if they are different, it is `false`. The predicate `u =/= v` is just the negation of `u == v`. In the module `BOOL`, valid formulas are reduced to the constant `true`, invalid formulas are reduced to the constant `false`, and all the others are reduced to a canonical form (modulo associativity and commutativity) consisting of an *exclusive or* of conjunctions.

Predicates that are not specified in `BOOL` are module-dependent and can be equationally defined as total Boolean functions over the domain formalized by $\mathcal{R}$. Therefore, we can define basic properties on a given rewrite theory $\mathcal{R}$ by means of a system module `PRED`($\mathcal{R}$) that

- imports the (Maude encoding of) the rewrite theory $\mathcal{R}$; and
- specifies predicates via user-defined operators that are associated with terminating and Church-Rosser equational definitions of some total Boolean function.

In this scenario, a well-formed formula is any term of sort `Bool` built using the operators and variables declared in the system module `PRED`($\mathcal{R}$).

We say that a formula φ *holds* in $\mathcal{R}$, iff φ can be reduced to `true` in `PRED`($\mathcal{R}$) (in symbols, $\mathcal{R} \models \varphi$).

Example 3. Consider the `BANK` system module of Example 2 and the new predicates given in the `BANK-PRED` module of Figure 1. Then, within `BANK-PRED` we can specify the formula

```
not(isPremium(ACC:Account)) implies getBalance(ACC:Account) > 0
```

which is true for every nonpremium bank account `ACC` with a positive balance.

3.2 System and Functional Assertions

System assertions formalize properties over (portions of) system states. Formally, a *system assertion* (also called *constrained term* in [26]) is an expression of the

form $S\{\varphi\}$ where S is a (possibly non ground) term in $\tau(\Sigma, \mathcal{V})$ of sort `State`, and φ is a well-formed formula such that $\mathcal{V}ar(\varphi) \subseteq \mathcal{V}ar(S)$.

System assertions are checked against states of the system specified by $\mathcal{R}$. Roughly speaking, a system assertion $S\{\varphi\}$ allows us to validate all system states s that match (modulo the equational theory E) the state "template" S w.r.t. the formula φ. More formally, we define the satisfaction of a system assertion in a system state as follows.

Definition 1 (system assertion satisfaction). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory. Let $S\{\varphi\}$ be a system assertion for $\mathcal{R}$ and s be a state in $\mathcal{R}$. Then, $S\{\varphi\}$ is* satisfied *in s (in symbols, $s \models S\{\varphi\}$) iff for each $w \in \mathcal{P}os(s)$, for each substitution σ if $s_{|w} =_E S\sigma$ then $\varphi\sigma$ holds in $\mathcal{R}$.*

Note that, if there is no subterm $s_{|w}$ of s that matches S (modulo E), we trivially have $s \models S\{\varphi\}$. This implies that $S\{\varphi\}$ is *not* satisfied in s (in symbols, $s \not\models S\{\varphi\}$) only in the case when there exist w and σ such that $s_{|w} =_E S\sigma$, and the formula $\varphi\sigma$ does not hold in $\mathcal{R}$. We call w a *system error symptom*. Roughly speaking, a system error symptom is the position of a subterm of the state s that is responsible for the violation of the considered assertion in s.

Definition 2 (system error symptoms). *The set of all system error symptoms for a state s and a system assertion $S\{\varphi\}$ is defined as follows:*

$$SysErr(s, S\{\varphi\}) = \{w \mid \exists\sigma.\ s_{|w} =_E S\sigma, w \in \mathcal{P}os(s),\ \text{and } \varphi\sigma \not\models \mathcal{R}\}.$$

Observe that $SysErr(s, S\{\varphi\}) = \emptyset$, whenever $s \models S\{\varphi\}$.

Example 4. Consider the extended rewrite theory of Example 3 together with the system assertion

$$\Theta = \texttt{< C:Id | B:Int | S:Status > \{ not(isPremium(< C:Id | B:Int | S:Status >)) implies B:Int > 0 \}}$$

Then, Θ is satisfied in the state `< Alice | 50 | active > ; < Bob | 40 | active > ; debit(Alice,60),` but it is not satisfied in $s_{err} =$ `< Alice | -10 | blocked > ; < Bob | 40 | active >`, since Alice's non-premium account has a negative balance. The detected error symptom for the considered state and assertion is the position `1` that refers to the subterm `< Alice | -10 | blocked >` of s_{err}.

The second type of assertions that we consider are *functional assertions*. Functional assertions allow one to specify the general pattern O of the canonical form for any input term t that matches a given template I, while allowing pre- and post-conditions $\varphi_{in}, \varphi_{out}$ over the equational simplification to also be declared. Their general form is $I\ \{\varphi_{in}\} \rightarrow O\ \{\varphi_{out}\}$ where $I, O \in \tau(\Sigma, \mathcal{V})$, $\varphi_{in}, \varphi_{out}$ well-formed formulas, $\mathcal{V}ar(\varphi_{in}) \subseteq \mathcal{V}ar(I)$ and $\mathcal{V}ar(\varphi_{out}) \subseteq \mathcal{V}ar(I) \cup \mathcal{V}ar(O)$. Intuitively, functional assertions allow us to specify the I/O behaviour of the equational simplification of a term t by providing

Input: an input template I that t can match and a pre-condition φ_{in} that t can meet;

Output: an output template O that the canonical form of t has to match and a post-condition φ_{out} that the computed canonical form of t has to meet (whenever the input term t matching I meets φ_{in}).

Note that, while system assertions $S\{\varphi\}$ resemble Matching Logic (ML) formulas $\pi \wedge \phi$ (called ML *patterns*), where π is a configuration term and ϕ is a first order logic formula, functional assertions $I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$ remind Reachability Logic (RL) formulas $\varphi \Rightarrow \varphi'$, where φ, φ' are ML patterns (for a survey on ML/RL, see [27]). Different from our functional assertions, which predicate on equational simplification sequences, RL formulas are evaluated on system computations. Namely, the semantics of a RL formula $\varphi \Rightarrow \varphi'$ is that any state satisfying φ transits (in zero or more steps) into a state satisfying φ', while ML formulas are used to express (and reason about) static state properties, similarly to our system assertions.

The notion of satisfaction for a functional assertion is given w.r.t. the equational simplification $\mu = t \rightarrow^*_{\Delta,B} t\downarrow_{\Delta,B}$ of term t into its canonical form $t\downarrow_{\Delta,B}$.

Definition 3 (functional assertion satisfaction). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, with $E = \Delta \cup B$. Let $I\{\varphi_{in}\} \rightarrow O\{\varphi_{out}\}$ be a functional assertion for $\mathcal{R}$, and μ be the equational simplification of the term t in $\tau(\Sigma, \mathcal{V})$ into its canonical form $t\downarrow_{\Delta,B}$ w.r.t. Δ modulo B. Then, $I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$ is* satisfied *in μ (in symbols, $\mu \models I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$) iff for each substitution σ_{in} s.t. $t =_B I\sigma_{in}$, if $\varphi_{in}\sigma_{in}$ holds in $\mathcal{R}$, then there exists σ_{out} such that $t\downarrow_{\Delta,B} =_B O(\sigma_{in}\downarrow_{\Delta,B})\sigma_{out}$ and $\varphi_{out}(\sigma_{in}\downarrow_{\Delta,B})\sigma_{out}$ holds in $\mathcal{R}$.*

Note that $I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$ is (trivially) satisfied in μ when either t does not match $I\sigma_{in}$ (modulo B) or $\varphi_{in}\sigma_{in}$ does not hold in $\mathcal{R}$. Intuitively, a functional error occurs in an equational simplification μ where the computed canonical form fails to match the structure or meet the properties of the output template O. In other words, $\Phi = I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$ is *not* satisfied in μ only in the case when there exists an *input* substitution σ_{in} s.t.

- $t =_B I\sigma_{in}$ and $\varphi_{in}\sigma_{in}$ holds in $\mathcal{R}$;
- $t\downarrow_{\Delta,B} \neq_B O(\sigma_{in}\downarrow_{\Delta,B})\sigma_{out}$ or $\varphi_{out}(\sigma_{in}\downarrow_{\Delta,B})\sigma_{out}$ does not hold in $\mathcal{R}$, for any substitution σ_{out}.

Definition 4 (functional error symptoms). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, with $E = \Delta \cup B$. Let $\Phi = I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$ be a functional assertion for $\mathcal{R}$. Let $\mu = t \rightarrow^*_{\Delta,B} t\downarrow_{\Delta,B}$ be an equational simplification such that $\mu \not\models \Phi$ with input substitution σ_{in}. Then, a* functional error symptom *for μ w.r.t. Φ is any position in $\mathcal{P}os(t\downarrow_{\Delta,B})$ that belongs to the following set:*

$$\begin{aligned} \mathit{FunErr}(\mu, \Phi) = \{&\pi(w) \in \mathcal{P}os(t\downarrow_{\Delta,B}) \mid ((g, \sigma_1, \sigma_2), \pi) \\ &= \widehat{lgg}_{\Delta\cup B}(t\downarrow_{\Delta,B}, O(\sigma_{in}\downarrow_{\Delta,B})) \text{ and } g_{|w} \in \mathcal{V}ar(g), w \in \mathcal{P}os(g)\} \end{aligned}$$

Roughly speaking, $\mathit{FunErr}(\mu, \Phi)$ is computed by "comparing" the canonical form $t\downarrow_{\Delta,B}$ with the instance $O(\sigma_{in}\downarrow_{\Delta,B})$ of the output template O by the (normalized) substitution $\sigma_{in}\downarrow_{\Delta,B}$ using a least general generalization algorithm modulo equational theories. More specifically, an arbitrarily-selected least general generalization (g, σ_1, σ_2) (modulo $\Delta \cup B$) between $t\downarrow_{\Delta,B}$ and $O(\sigma_{in}\downarrow_{\Delta,B})$ is chosen via $\widehat{lgg}_{\Delta\cup B}$, and potentially erroneous subterms of $t\downarrow_{\Delta,B}$ are detected by selecting every position $\pi(w) \in \mathcal{P}os(t\downarrow_{\Delta,B})$ in correspondence with a position $w \in \mathcal{P}os(g)$. The intuition behind this method is that variables in g reflect possible discrepancies between the canonical form and the instantiated output template, and, thus, subterms $(t\downarrow_{\Delta,B})_{|\pi(w)}$ represent, to some extent, a possible anomalous subterm of $t\downarrow_{\Delta,B}$.

It is worth noting that the use of $\widehat{lgg}_{\Delta\cup B}$ is generally preferable to the adoption of a pure syntactic *lgg* algorithm since it minimizes the number of variables in g (and, hence, the points of discrepancy between $t\downarrow_{\Delta,B}$ and $O(\sigma_{in}\downarrow_{\Delta,B})$, which facilitates isolating erroneous information. Let us see an example.

Example 5. Let us consider the equational simplification $\mathtt{f}(0,0) \rightarrow^{+}_{\Delta,B} \mathtt{c}(1,3)$ w.r.t. an equational theory $(\Sigma, \Delta \cup B)$ in which the operator `c` is declared commutative. Let $\Phi = \mathtt{f}(\mathtt{X},\mathtt{Y})\ \{\mathtt{true}\} \rightarrow \mathtt{c}(\mathtt{Z},1)\ \{\mathtt{even}(\mathtt{Z})\}$ be a functional assertion, where predicate `even`(`Z`) checks whether `Z` is an even number.

Then, $(\mathtt{f}(0,0), \mathtt{c}(1,3)) \not\models \Phi$ (with input substitution $\sigma_{in} = \{\mathtt{X}/0, \mathtt{Y}/0\}$), since variable `Z` in the output template $\mathtt{c}(\mathtt{Z},1)$ is bound to `3` and `even`(`3`) is false. Then, $\widehat{lgg}_{\Delta\cup B}(\mathtt{c}(1,3), \mathtt{c}(\mathtt{Z},1))$ returns a pair $((g, \sigma_1, \sigma_2), \pi)$ such that g contains the minimum number of variables. For instance, $\widehat{lgg}_{\Delta\cup B}(\mathtt{c}(1,3), \mathtt{c}(\mathtt{Z},1)) = ((\mathtt{c}(\mathtt{Z},1), \{\mathtt{Z}/3\}, \{\}), \{1 \mapsto 2\})$ and $\mathit{FunErr}(\mu, \Phi) = \{2\}$, which precisely detects that the term $\mathtt{c}(1,3)_{|2} = 3$ is what causes the violation of Φ.

By contrast, the computation of a purely syntactic least general generalization would have delivered the more general result $(\mathtt{c}(\mathtt{W},\mathtt{Z}), \{\mathtt{W}/1, \mathtt{Z}/3\}, \{\})$ and the larger functional error symptom set $\{1,2\}$ (which represents the positions of both arguments of the canonical form $\mathtt{c}(1,3)$), thereby hindering the isolation of the erroneous subterm of $\mathtt{c}(1,3)$.

Example 6. Consider again the extended rewrite theory of Example 3. Then, the functional assertion

$$\Phi = \mathtt{updateStatus(ACC:Account)\{isPremium(ACC:Account)\}} \rightarrow \mathtt{ACC:Account\{true\}}$$

states that premium account statuses (as well as other information in the account) remain unchanged after `updateStatus` is invoked. Thus, Φ is not satisfied in the following equational simplification

$$\mathtt{updateStatus(<\ Bob\ |\ 95\text{-}100\ |\ active\ >)} \rightarrow^{+}_{\Delta,B} \mathtt{<\ Bob\ |\ -5\ |\ blocked\ >}$$

with input substitution $\sigma_{in} = \{\mathtt{ACC/<\ Bob\ |\ 95\text{-}100\ |\ active\ >}\}$ and $\sigma_{in}\downarrow_{\Delta,B} = \{\mathtt{ACC/<\ Bob\ |\ -5\ |\ active\ >}\}$. Hence, there is a single (syntatic) least general generalizer

$$\widehat{lgg}_{\Delta\cup B}(\mathtt{<\ Bob\ |\ -5\ |\ blocked\ >}, \mathtt{ACC}(\sigma_{in}\downarrow_{\Delta,B})) =$$

$$= ((\texttt{< Bob | -5 | X:Status >},\{\texttt{X/blocked}\},\{\texttt{X/active}\}),\{3 \mapsto 3\})$$

where $\mathit{FunErr}(\mu, \Phi) = \{3\}$ is the functional error symptom set that pinpoints the anomalous status on Bob's premium account `< Bob | -5 | blocked >`.

Finally, an *assertional specification* $\mathcal{A}$ for a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is a set of functional and system assertions for $\mathcal{R}$. By $\mathcal{F}(\mathcal{A})$, we denote the set of functional assertions in $\mathcal{A}$, while $\mathcal{S}(\mathcal{A})$ denotes the set of system assertions in $\mathcal{A}$. By $s \models \mathcal{S}(\mathcal{A})$ (resp. $\mu \models \mathcal{F}(\mathcal{A})$), we denote that s satisfies all assertions in $\mathcal{S}(\mathcal{A})$ (resp. μ satisfies all assertions in $\mathcal{F}(\mathcal{A})$).

In the following section, we outline our previous work on trace slicing for RWL theories.

4 Enhancing Trace Slicing

Trace slicing [1,2,3,4,5] is a transformation technique for RWL theories that can drastically reduce the size and complexity of entangled, textually-large execution traces by focusing on selected computation aspects. This is done by uncovering data dependences among related parts of the trace w.r.t. a user-defined slicing criterion (i.e., a set of symbols that the user wants to observe). This technique aims to improve the analysis, comprehension, and debugging of sophisticated rewrite theories by helping the user inspect involved traces in an easier way. By step-wisely reducing the amount of information in the simplified trace, it is easier for the user to locate program faults because pointless information or unwanted rewrite steps have been automatically removed. Roughly speaking, in our slices, the irrelevant subterms of a term are omitted, leaving "holes" that are denoted by special variable symbols $\bullet$.

A term *slice* of the term s is a term $s^\bullet$ that hides part of the information in s; that is, the irrelevant data in s that we are not interested in are simply replaced by (fresh) $\bullet$-variables of appropriate sort, denoted by $\bullet_i$, with $i = 0, 1, 2, \ldots$.

The next auxiliary definition formalizes the function $\mathit{Tslice}(t, P)$, which allows a term slice of t to be constructed w.r.t. a set of positions P of t. The function *Tslice* relies on the function $\mathit{fresh}^\bullet$ whose invocation returns a (fresh) variable $\bullet_i$ of appropriate sort that is distinct from any previously generated variable $\bullet_j$.

Definition 5 (Term Slice). *Let $t \in \tau(\Sigma, \mathcal{V})$ be a term and let P be a set of positions s.t. $P \subseteq \mathcal{P}os(t)$. Then, the term slice $\mathit{Tslice}(t, P)$ of t w.r.t. P is computed as follows.*

$$\mathit{Tslice}(t, P) = \mathit{recslice}(t, P, \Lambda), \text{ where}$$

$$\mathit{recslice}(t, P, p) = \begin{cases} f(\mathit{recslice}(t_1, P, p.1), \ldots, \mathit{recslice}(t_n, P, p.n)) & \\ \qquad \text{if } t = f(t_1, \ldots, t_n), n \geq 0, \text{ and } p \in \bar{P} \\ t \qquad \text{if } t \in \mathcal{V} \text{ and } p \in \bar{P} \\ \mathit{fresh}^\bullet \qquad \text{otherwise} \end{cases}$$

and $\bar{P} = \{u \mid u \leq p \wedge p \in P\}$ is the prefix closure *of P.*

Roughly speaking, the function $Tslice(t, P)$ yields a term slice of t w.r.t. a set of positions P that includes all symbols of t that occur within the paths from the root of t to any position in P, while the remaining information of t is abstracted by means of $\bullet$-variables.

Example 7. Consider the specification of Example 1 and the state $t =$ `< Alice | 50 | active > ; < Bob | 100 | active > ; debit(Alice,20)`. Consider the set $P = \{1.2, 3.1, 3.2\}$ of positions in t. Then,

$$Tslice(t, P) = \texttt{<}\ \bullet_1\ \texttt{|}\ 50\ \texttt{|}\ \bullet_2\ \texttt{>}\ \texttt{;}\ \bullet_3\ \texttt{;}\ \texttt{debit(Alice,20)}.$$

Trace slicing can be carried out forward or backward. While the forward trace slicing results in a form of impact analysis that identifies the scope and potential consequences of changing the program input, backward trace slicing allows provenance analysis to be performed; i.e., it shows how (parts of) a program output depend(s) on (parts of) its input and helps estimate which input data need to be modified to accomplish a change in the outcome. While dependency provenance provides information about the origins of (or influences upon) a given result, the notion of descendants is the key for impact evaluation. In the sequel, we focus on backward trace slicing.

Let us illustrate by means of an example how it can help the user *think backwards* (i.e., to deduce the conditions under which a program produces some observed data).

Example 8. Consider the `BANK` system module of Example 2 and the computation trace $\mathcal{C}_{bank}$ in program `BANK` that starts in the initial state

```
< Alice | 50 | active > ; < Bob | 20 | active > ; < Charlie | 20 |
active > ; < Daisy | 20 | active > ; debit(Alice,80) ; credit(Alice,20)
```

and ends in the final state

```
< Alice | - 10 | blocked > ; < Bob | 20 | active > ; < Charlie | 20 |
active > ; < Daisy | 20 | active >
```

Let us assume we manually define as the slicing criterion the negative balance `-10` for client `Alice`, which is a possible malfunction of the `BANK` specification, since regular account balances must be non-negative numbers according to the semantics intended by the programmer. Therefore, we execute trace slicing on the trace $\mathcal{C}_{bank}$ w.r.t. the slicing criterion $\texttt{<}\ \bullet_1\ \texttt{|}\ \texttt{-10}\ \texttt{|}\ \bullet_2\ \texttt{>}\ \texttt{;}\ \bullet_3\ \texttt{;}\ \bullet_4\ \texttt{;}\ \bullet_5$ that observes the negative balance of Alice's account in order to determine the cause of such a disfunction. The output trace slice delivered by the trace slicing technique is as follows

$$\begin{array}{l}
\texttt{<}\ \bullet_1\ \texttt{|}\ 50\ \texttt{|}\ \bullet_2\ \texttt{>}\ \texttt{;}\ \bullet_3\ \texttt{;}\ \bullet_4\ \texttt{;}\ \bullet_5\ \texttt{;}\ \texttt{debit(}\bullet_1\texttt{,80)}\ \texttt{;}\ \texttt{credit(}\bullet_1\texttt{,20)} \overset{\texttt{credit}}{\bullet\!\!\rightarrow} \\
\texttt{<}\ \bullet_1\ \texttt{|}\ 70\ \texttt{|}\ \bullet_2\ \texttt{>}\ \texttt{;}\ \bullet_3\ \texttt{;}\ \bullet_4\ \texttt{;}\ \bullet_5\ \texttt{;}\ \texttt{debit(}\bullet_1\texttt{,80)} \overset{\texttt{debit}}{\bullet\!\!\rightarrow} \\
\texttt{<}\ \bullet_1\ \texttt{|}\ \texttt{-10}\ \texttt{|}\ \bullet_2\ \texttt{>}\ \texttt{;}\ \bullet_3\ \texttt{;}\ \bullet_4\ \texttt{;}\ \bullet_5
\end{array}$$

which greatly simplifies the trace $\mathcal{C}_{bank}$ by only showing the origins of the observed negative balance while excluding all the bank accounts that are not related to `Alice`.

Throughout this paper, we assume the existence of a *backwardSlicing*($s_0 \rightarrow^*_{\Delta \cup B} s_n, s_n^\bullet$) function as defined in [8] that yields the backward trace slice $s_0^\bullet \bullet\!\!\rightarrow^* s_n^\bullet$ of the computation trace $s_0 \rightarrow^*_{\Delta \cup B} s_n$ w.r.t. a term slice $s_n^\bullet$ of s_n. This function relies on an instrumentation technique for Maude steps that allows the relevant information of the step, such as the selected redex and the contractum produced by the step, to be traced explicitly despite the fact that terms are rewritten modulo a set B of equational axioms that may cause their components to be implicitly reordered in the original trace. Also, the dynamic dependencies exposed by backward trace slicing are exploited in [8] to provide a (preliminary) program slicing capability that can identify those parts of a Maude theory that can (potentially) affect the values computed at some point of interest.

The main idea of this work is to enhance backward trace slicing by using runtime assertion checking to automatically identify the relevant symbols to be traced back from the erroneous states of the trace, that is, those states where an assertion is falsified. In conventional program development environments, when a given assertion check fails, the programmer must thoughtfully identify which program statements impacted on the value(s) causing the assertion failure. An additional advantage of blending trace slicing and runtime checking together is that the runtime checking not only helps automate the trace slicing, but trace slicing also helps answer the question that immediately arises when an assertion is violated. This question is "What caused it?". By using our enhanced, backward trace slicing methodology, error diagnosis is greatly simplified because accurate criteria for slicing are automatically inferred from the computed error symptoms that immediately bootstrap the slicing process so that much of the irrelevant data that does not influence the falsified assertions is automatically cut off.

5 Integrating Assertion-Checking and Trace Slicing

Dynamic assertion-checking and trace slicing can be smoothly combined together to facilitate the debugging of ill-defined rewrite theories. In this section, we formulate an assertion-checking methodology to verify whether a given computation trace $\mathcal{C}$ meets the requirements formalized by an assertional specification $\mathcal{A}$. In the case when a functional or system assertion $A \in \mathcal{A}$ fails to be satisfied over $\mathcal{C}$, a fragment $\mathcal{P}$ of $\mathcal{C}$ (that exhibits the anomalous behaviour w.r.t. A) is returned together with the corresponding set of system/functional error symptoms. Then, we show how backward trace slicing can take advantage of the computed error symptoms to produce small, easy-to-inspect computation slices of all those fragments that have been proven to be erroneous by the assertion-checking methodology.

5.1 Dynamic Assertion-Checking

We first extend the notion of satisfaction of the functional assertions to state equational simplifications (i.e., equational simplifications that reduce a state into its canonical form), where the state may contain an arbitrary number of function calls that might eventually be simplified. For this purpose, we introduce the

following auxiliary definitions. Given $\mathcal{R} = (\Sigma, E, R)$, with $E = \Delta \cup B$, the term t is an equational redex in $\mathcal{R}$ if there is $(\lambda = \rho \texttt{ if } C) \in \Delta$ and substitution σ such that $t =_B \lambda\sigma$. Given $\mathcal{R}$ and a system state s in $\mathcal{R}$, $Top(s)$ is the set of minimal positions $w \in \mathcal{P}os(s)$ such that $s_{|w}$ is an equational redex in $\mathcal{R}$. Formally,

$$Top(s) = \{w \in \mathcal{P}os(s) \mid s_{|w} \text{ is an equational redex and } \nexists w' \leq w \text{ s.t. } s_{|w'} \text{ is an equational redex}\}.$$

Roughly speaking, $Top(s)$ selects all the positions in $\mathcal{P}os(s)$ that identify those outermost subterms of s to be equationally simplified into their canonical form in order to compute $s{\downarrow_{\Delta,B}}$. In other words, given the equational simplification of the state s, $\mathcal{S} : s \rightarrow^+_{\Delta,B} s{\downarrow_{\Delta,B}}$, each subterm $s_{|w}$, with $w \in Top(s)$, is reduced to $(s{\downarrow_{\Delta,B}})_{|w}$ in $\mathcal{S}$. This allows functional assertions to be effectively checked over each equational simplification $s|_w \rightarrow^+_{\Delta,B} (s{\downarrow_{\Delta,B}})_{|w}$ such that $w \in Top(s)$.

Definition 6 (extended functional assertion satisfaction). *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, with $E = \Delta \cup B$, and let s be a system state in $\mathcal{R}$ such that $Top(s) \neq \{\Lambda\}$. Let $s \rightarrow^+_{\Delta,B} s{\downarrow_{\Delta,B}}$ be an equational simplification for the state s in $\mathcal{R}$. Let $\mathcal{A}$ be an assertional specification for $\mathcal{R}$. We say that $\mathcal{F}(\mathcal{A})$ is* satisfied *in $s \rightarrow^+_{\Delta,B} s{\downarrow_{\Delta,B}}$ (in symbols, $s \rightarrow^+_{\Delta,B} s{\downarrow_{(\Delta,B)}} \models \mathcal{F}(\mathcal{A})$), iff for each $w \in Top(s)$, $s|_w \rightarrow^+_{\Delta,B} (s{\downarrow_{\Delta,B}})_{|w} \models \mathcal{F}(\mathcal{A})$.*

System and functional error symptoms (whose definitions have been given in Section 3 for a single system/functional assertion) can be naturally extended to assertional specifications in the following way.

Definition 7 (state error symptoms). *Let $\mathcal{R} = (\Sigma, E, R)$, with $E = \Delta \cup B$, be a rewrite theory. Let $\mathcal{A}$ be an assertional specification for $\mathcal{R}$. Let s be a state of $\mathcal{R}$. Then,*

$$SysErr(s, \mathcal{A}) = \bigcup_{\Theta \in \mathcal{S}(\mathcal{A})} SysErr(s, \Theta)$$

$$FunErr(s \rightarrow^+_{\Delta,B} s{\downarrow_{\Delta,B}}, \mathcal{A}) = \bigcup_{\Phi \in \mathcal{F}(\mathcal{A}), w \in Top(s)} \{(s_{|w} \rightarrow^+_{\Delta,B} (s{\downarrow_{\Delta,B}})_{|w}, FunErr(s_{|w} \rightarrow (s{\downarrow_{\Delta,B}})_{|w}, \Phi))\}$$

The notion of satisfaction for an assertional specification in a given computation is then formalized as follows.

Definition 8 (satisfaction of an assertional specification). *Let $\mathcal{R} = (\Sigma, E, R)$, with $E = \Delta \cup B$, be a rewrite theory and $\mathcal{C}$ be a computation in $\mathcal{R}$. Let $\mathcal{A}$ be an assertional specification for $\mathcal{R}$. Then the specification $\mathcal{A}$ is* satisfied *in $\mathcal{C}$ iff*

- *for each state s in $\mathcal{C}$ that is a canonical form w.r.t. Δ modulo B, $s \models \mathcal{S}(\mathcal{A})$;*
- *for each state s in $\mathcal{C}$ that is not a canonical form w.r.t. Δ modulo B, $s \rightarrow^+_{\Delta,B} s{\downarrow_{(\Delta,B)}} \models \mathcal{F}(\mathcal{A})$.*

To check an assertional specification $\mathcal{A}$ in a given computation $\mathcal{C}$, we can simply traverse $\mathcal{C}$ and progressively evaluate system assertions over states and

functional assertions over state equational simplifications, respectively. Definition 9 formalizes this methodology into the function $check(\mathcal{C}, \mathcal{A})$ that takes as input a computation $\mathcal{C}$ and an assertional specification $\mathcal{A}$ and delivers a triple $(\mathcal{P}, Err, flag)$ where $\mathcal{P}$ is a prefix of $\mathcal{C}$, Err is a set of functional or system error symptoms w.r.t. $\mathcal{A}$, and $flag \in \{none, sys, fun\}$.

Roughly speaking, function $check(\mathcal{C}, \mathcal{A})$ returns $(\mathcal{P}, Err, flag)$ as soon as it encounters either a state or a state equational simplification in which $\mathcal{A}$ is not satisfied: $\mathcal{P}$ represents a prefix of $\mathcal{C}$ that reaches a state in which a system/functional assertion is violated, Err specifies the associated error symptom set, and $flag$ declares the nature of the computed symptoms (fun stands for functional error symptoms, sys for system error symptoms, and the keyword $none$ indicates that no symptom has been identified).

Definition 9 (assertion checking). *Let $\mathcal{R} = (\Sigma, E, R)$, with $E = \Delta \cup B$, be a rewrite theory and $\mathcal{C}$ be a computation in $\mathcal{R}$. Let $\mathcal{A}$ be an assertional specification for $\mathcal{R}$.*

$$check(\mathcal{C}, \mathcal{A}) = \begin{cases} (\varepsilon, \emptyset, none) & \text{if } \mathcal{C} = \varepsilon \\ (\mu \rightarrow^*_{R,B} \mathcal{C}'', Err, flag) & \text{if } \mathcal{C} = \mu \rightarrow^*_{R,B} \mathcal{C}' \text{ and } \mu \models \mathcal{F}(\mathcal{A}) \\ & \text{and } (\mathcal{C}'', Err, flag) = check(\mathcal{C}', \mathcal{A}) \\ (\mu, FunErr(\mu, \mathcal{F}(\mathcal{A})), fun) & \text{if } \mathcal{C} = \mu \rightarrow^*_{R,B} \mathcal{C}' \text{ and } \mu \not\models \mathcal{F}(\mathcal{A}) \\ (s \rightarrow_{R,B} \mathcal{C}'', Err, flag) & \text{if } \mathcal{C} = s \rightarrow^*_{R,B} \mathcal{C}',\ s = s\downarrow_{(\Delta,B)} \\ & \text{and } s \models \mathcal{S}(\mathcal{A}) \\ & \text{and } (\mathcal{C}'', Err, flag) = check(\mathcal{C}', \mathcal{A}) \\ (s, SysErr(s, \mathcal{S}(\mathcal{A})), sys) & \text{if } \mathcal{C} = s \rightarrow^*_{R,B} \mathcal{C}',\ s = s\downarrow_{(\Delta,B)} \\ & \text{and } s \not\models \mathcal{S}(\mathcal{A}) \end{cases}$$

where $\mu = s \rightarrow^+_{\Delta,B} s\downarrow_{\Delta,B}$ is a non-empty equational simplification for s.

The runtime checking methodology formalized in Definition 9 can be interpreted either as an asynchronous (and trace-storing) technique or as a synchronous one (by considering that the input trace $\mathcal{C}$ is lazily generated as successive Maude steps are incrementally consumed by the calculus). In the following section, we formalize a truly synchronous methodology where traces, or rather whole search trees, can be stepwisely examined in a forward direction, reporting a violation at the exact step where it occurs.

5.2 Runtime Assertion-Based Backward Trace Slicing

Given a conditional rewrite theory $\mathcal{R} = (\Sigma, E, R)$, with $E = \Delta \cup B$, the transition space of all computations in $\mathcal{R}$ from the initial state s_0 can be represented as a *computation tree*,[3] $\mathcal{T}_{\mathcal{R}}(s_0)$. RWL computation trees are typically large and complex objects that represent the highly-concurrent, nondeterministic nature of rewrite theories.

[3] In order to facilitate trace inspection, computations are visualized as trees, although they are internally represented by means of more efficient graph-like data structures that allow common subexpressions to be shared.

function $analyze(s_0, (\Sigma, \Delta \cup B, R), \mathcal{A}, depth)$
1. $\mathcal{T} = s_0$
2. $d = 0$
3. **while** $(d \leq depth)$ **do**
4. $F = frontier(\mathcal{T})$
5. **for each** $s \in F$
6. **for each** $\mathcal{M} \in m\mathcal{S}(s)$
7. $(\mathcal{P}, Err, flag) = check(\mathcal{M}, \mathcal{A})$
8. **case** *flag* **of**
9. *none* **:**
10. $\mathcal{T} = expand(\mathcal{T}, s, \mathcal{M})$
11. *sys* **:**
12 $w = selectSysSymptom(Err)$
13 $l^\bullet = TSlice(last(\mathcal{P}), \mathcal{P}os_w(last(\mathcal{P}))$
14. **return** $backwardSlice(s_0 \rightarrow^*_{R \cup \Delta, B} \mathcal{P}, l^\bullet)$
15. *fun***:**
16. $(t \rightarrow^+_{\Delta,B} t{\downarrow_{\Delta,B}}, L) = \{selectFunSymptom(Err)\})$
17. $(t{\downarrow_{\Delta,B}})^\bullet = TSlice(t{\downarrow_{\Delta,B}}, \bigcup_{w \in L} \mathcal{P}os_w(t{\downarrow_{\Delta,B}}))$
18. **return** $backwardSlice(t \rightarrow^+_{\Delta,B} t{\downarrow_{\Delta,B}}, (t{\downarrow_{\Delta,B}})^\bullet)$
19. **end case**
20. **end for**
21. **end for**
22. $d = d + 1$
23. **end while**
24. **return** $\mathcal{T}$

end

Figure 2. The *analyze* function.

Our methodology checks rewrite theories w.r.t. an assertional specification $\mathcal{A}$ at runtime by incrementally generating and checking the computation tree $\mathcal{T}_\mathcal{R}(s_0)$ until a fixed depth. In fact, the complete generation of $\mathcal{T}_\mathcal{R}(s_0)$ is generally not feasible since some of its branches may be infinite as they encode nonterminating computations. The general analysis algorithm, which is specified by the routine $analyze(s_0, \mathcal{R}, \mathcal{A}, depth)$, is given in Figure 2. We use the following auxiliary notation: given a position w of a term t, $\mathcal{P}os_w(t) = \{w.w' \mid w.w' \in \mathcal{P}os(t)\}$. The computation tree is constructed breadth-first, starting from a tree $\mathcal{T}$ that consists of a single root node s_0. At each expansion stage, the leaf nodes of the current $\mathcal{T}$ are computed by the function $frontier(\mathcal{T})$. Expansion of an arbitrary node s is done by deploying all the possible Maude computation steps stemming from s that are given by $m\mathcal{S}(s)$. Whenever a Maude step $\mathcal{M}$ is produced, it is also checked w.r.t. the specification $\mathcal{A}$ by calling $check(\mathcal{M}, \mathcal{A})$ that computes the triple $(\mathcal{P}, Err, flag)$. According to the computed *flag* value, the algorithm distinguishes the following cases:

flag = *none***.** No error symptoms have been computed; hence, $\mathcal{A}$ is satisfied in the Maude step $\mathcal{M}$, and $\mathcal{M}$ can safely expand the node s by replacing s with the path represented by $\mathcal{M}$ (via the invocation of $expand(\mathcal{T}, s, \mathcal{M})$), thereby augmenting $\mathcal{T}$.

flag = *sys*. In this case, *check* returns a set of system error symptoms *Err* together with a computation $\mathcal{P}$ (which is a prefix of the Maude step $\mathcal{M}$) that violates a system assertion of $\mathcal{A}$. The computation $s_0 \rightarrow^*_{R \cup \Delta, B} \mathcal{P}$ is then generated and backward sliced w.r.t. a term slice $l^\bullet$ of the last state of $\mathcal{P}$. This term slice conveys all the relevant information that we automatically retrieve by using Definition 5 from the (system) error symptom w selected by the function *selectSysSymptom*(*Err*), while all other symbols in l are considered meaningless and simply pruned away. This way, the algorithm delivers a trace slice $s_0^\bullet \bullet\!\!\rightarrow \mathcal{P}^\bullet$ that removes from the computation all of the information that does not affect the production of the chosen error symptom.

flag = *fun*. Some functional assertions have been violated by the considered Maude step $\mathcal{M}$. Hence, the algorithm selects a functional error symptom $(t \rightarrow^+_{\Delta,B} t{\downarrow_{\Delta,B}}, L)$ and returns the backward trace slicing of $t \rightarrow^+_{\Delta,B} t{\downarrow_{\Delta,B}}$ w.r.t. a term slice of $t{\downarrow_{\Delta,B}}$ that includes all the subterms of $t{\downarrow_{\Delta,B}}$ that are rooted at positions in L. As explained in Section 3.2, these subterms indicate possible causes of the assertion violation.

It is worth noting that, in our framework, we do not attach any specific semantics to *selectSysSymptom* and *selectFunSymptom* functions since many selection strategies can be specified with different degrees of automation and associated tradeoffs. For instance, we can simply obtain a fully automatic selection strategy by selecting the first symptom in *Err*. On the other hand, a purely interactive strategy can be implemented by asking the user to choose a symptom at runtime.

Finally, if the *analyze* function terminates without detecting any assertion violation, then a (verified) tree $\mathcal{T}$ is delivered that encodes the first *depth* levels of the computation tree $\mathcal{T}_{\mathcal{R}}(s_0)$; otherwise, the trace slice of the first computation that is found to violate an assertion is delivered. When multiple assertions are violated, *analyze* can be invoked iteratively.

5.3 The ABETS system

The assertion-based slicing methodology of Section 5.2 has been fully implemented in a prototype tool we call ABETS (*Assertion-BasEd Trace Slicer*), which is publicly available at `http://safe-tools.dsic.upv.es/abets` together with some documentation and examples. The implementation comprises: (i) a front-end consisting of a RESTful Web service written in Java, with an intuitive user interface based on AJAX technology written in HTML5, Canvas, and Javascript; and (ii) a back-end that implements the proposed trace analysis methodology in Maude. The implementation of the backend consists of about 350 Maude function definitions (approximately 2700 lines of source code) that partially reuse the slicing and exploration machinery developed in previous work [4,5,8], extending it with the constraint-checking capabilities described in this paper.

To perform dynamic analysis with ABETS, the user must provide (i) the Maude program to analyze together with an initial state, and (ii) the list of assertions to be checked together with the module that defines the extra predicates

that are used in the assertions. In order to non-deterministically search for assertion violations, the tree expansion is carried out up to a given depth bound that is measured in Maude steps. Whenever an assertion fails to be satisfied in the computation tree, the user is given an automatically generated counterexample trace slice that he/she can fully inspect, query, and slice further.

In ABETS, the trace slices can be easily navigated and all of the relevant information of the rewrite steps involved (e.g., equation/rule applications, matching substitutions, redex/contractum positions) is recorded and made available to the user. Furthermore, by disregarding rules and equations that are not used in the computed trace slice, ABETS can also generate a dynamic program slice where only potentially faulty fragments of the code are kept. Our preliminary experience has shown that the synergistic capabilities of ABETS can provide a very powerful Swiss knife in error diagnosis and debugging. The system allows assertion checking to be disabled when the functions/states they refer to are no longer under consideration so that no overhead is incurred after program analysis.

For demonstration purposes, let us analyze the `BANK` system of Example 2 together with an assertional specification $\mathcal{A}_{\texttt{BANK}}$ that includes the system assertion Θ of Example 4 and the functional assertion Φ of Example 6. Let us consider the expansion of all Maude steps that originate from the initial state

```
s0 = < Alice | 20 | active > ; < Bob | 50 | active > ; < Charlie | 10 |
     active > ; debit(Alice,30) ; transfer(Bob,Charlie,60)
```

This is achieved in ABETS by calling $\mathit{analyze}(\mathcal{R}_{\texttt{BANK}}, \mathcal{A}_{\texttt{BANK}}, 1)$, where $\mathcal{R}_{\texttt{BANK}}$ is the rewrite theory specified by the `BANK` system module.

The assertion checking algorithm immediately discovers that Θ is not satisfied in the following Maude step $\mathcal{M}$

```
s0 --debit--> updateStatus(< Alice | 20 - 30 | active >) ; < Bob | 50 | active
              > ; < Charlie | 10 | active > ; transfer(Bob,Charlie,60)
   -->+ < Alice | -10 | blocked > ; < Bob | 50 | active > ; < Charlie |
              10 | active > ; transfer(Bob,Charlie,60)
```

since Alice's nonpremium account has a negative balance. Here, the transition $\longrightarrow^+$ represents the equational state simplification that follows the rewrite step from `s0` by using the `debit` rule.

Then, a system error symptom is automatically computed by the tool, which unambiguously signals the anomalous subterm `< Alice | -10 | blocked >` of the last state of $\mathcal{M}$, and produces the associated term slice

$$l^{\bullet} = \texttt{< Alice | -10 | blocked > ; } \bullet_1 \texttt{ ; } \bullet_2 \texttt{ ; } \bullet_3$$

Finally, the algorithm automatically generates the backward trace slice of $\mathcal{M}$ w.r.t. $l^{\bullet}$, that is,

```
< Alice | 20 | active > ; •1 ; •2 ; debit(Alice,30) ; •3 •-->+
              < Alice | -10 | blocked > ; •1 ; •2 ; •3
```

Trace information ×

State	Label	Trace	Trace Slice
1	'Start	< Alice | 20 | active > ; < Bob | 50 | active > ; < Charlie | 10 | active > ; debit(Alice,30) ; transfer(Bob,Charlie,60)	•
2	toBnf	debit(Alice,30) ; < Alice | 20 | active > ; < Bob | 50 | active > ; < Charlie | 10 | active > ; transfer(Bob,Charlie,60)	•
3	fromBnf	< Bob | 50 | active > ; < Charlie | 10 | active > ; transfer(Bob, Charlie,60) ; debit(Alice,30) ; < Alice | 20 | active >	•
4	debit	< Bob | 50 | active > ; < Charlie | 10 | active > ; transfer(Bob, Charlie,60) ; updateAccountStatus(< Alice | 20 - 30 | active >)	• ; • ; • ; **updateAccountStatus(< • | 50 - 60 | active >)**
5	builtIn	< Bob | 50 | active > ; < Charlie | 10 | active > ; transfer(Bob, Charlie,60) ; updateAccountStatus(< Alice | - 10 | active >)	• ; • ; • ; **updateAccountStatus(< • | - 10 | active >)**
6	Label-EQ43	< Bob | 50 | active > ; < Charlie | 10 | active > ; transfer(Bob, Charlie,60) ; < Alice | - 10 | blocked >	• ; • ; • ; **<** • **|** • **| blocked >**
Total size:		828	105
Reduction Rate: 87%			

Figure 3. Trace Slice after refuting the functional assertion Φ of Example 6.

which suggests an erroneous implementation of the `debit` rule. Indeed, `debit` always authorizes withdrawals from a nonpremium account even when the intended payout exceeds the account balance, which is in contrast with the statement asserted by Θ.

Similarly, by re-executing the analysis algorithm on a mutation of the original `BANK` module that fixes the buggy `debit` rule, we can also discover a violation of the functional assertion Φ that detects an anomalous behaviour of function `updateStatus`: in fact, `updateStatus` blocks *every* bank account with a negative balance, while premium accounts should always be kept active. The delivered trace slice is shown in Figure 3 toghether with the achieved reduction (87%). Finally, by running the *program slice* option of ABETS, all program statements related to the updateStatus function are automatically identified and isolated in a program slice, since they can (potentially) cause the erroneous program behavior.

6 Conclusions and Further Work

We have formalized a framework that integrates dynamic slicing and runtime assertion checking to help diagnose programming errors in rewriting logic theories. A key feature of our approach is that the assertions (or more precisely, their runtime checks) are used to automatically synthesize advisable slicing criteria from inferred error symptoms. Our methodology smoothly blends in with the general framework for the analysis and exploration of rewriting logic computations that we developed in previous research [8].

The techniques we have developed are adequately fast and usable when applied to programs of several hundred lines, yet there are certainly several ways that our prototype implementation can be improved. For instance, one issue of interest would be to properly extend the current linear representation of Maude steps in ABETS, which intentionally obviates recording the traces for the recur-

sive evaluation of conditions for the sake of efficiency. Other planned improvements are to add more flexibility to the selection and processing of violated assertions and to refine the slicing criterion C that we infer from the falsified functional assertion $I\ \{\varphi_{in}\} \rightarrow O\ \{\varphi_{out}\}$, by further generalizing C using φ_{out} to reduce the number of variables of interest. Finally, we are also working on extending the system to deal with (object-oriented) Full Maude specifications.

References

1. M. Alpuente, D. Ballis, J. Espert, and D. Romero. Backward Trace Slicing for Rewriting Logic Theories. In *Proceedings of the 23rd International Conference on Automated Deduction (CADE 2011)*, volume 6803 of *Lecture Notes in Computer Science*, pages 34–48. Springer-Verlag, 2011.
2. M. Alpuente, D. Ballis, F. Frechina, and D. Romero. Backward Trace Slicing for Conditional Rewrite Theories. In *Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2012)*, volume 7180 of *Lecture Notes in Computer Science*, pages 62–76. Springer-Verlag, 2012.
3. M. Alpuente, D. Ballis, F. Frechina, and D. Romero. Julienne: A Trace Slicer for Conditional Rewrite Theories. In *Proceedings of the 18th International Symposium on Formal Methods (FM 2012)*, volume 7436 of *Lecture Notes in Computer Science*, pages 28–32. Springer-Verlag, 2012.
4. M. Alpuente, D. Ballis, F. Frechina, and D. Romero. Using Conditional Trace Slicing for improving Maude Programs. *Science of Computer Programming*, 80, Part B:385 – 415, 2014.
5. M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Slicing-Based Trace Analysis of Rewriting Logic Specifications with *i*Julienne. In *Proceedings of the 22nd European Symposium on Programming (ESOP 2013)*, volume 7792 of *Lecture Notes in Computer Science*, pages 121–124. Springer-Verlag, 2013.
6. M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Inspecting Rewriting Logic Computations (in a Parametric and Stepwise Way). In *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi (SAS 2014)*, volume 8373 of *Lecture Notes in Computer Science*, pages 229–255. Springer-Verlag, 2014.
7. M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Debugging Maude Programs via Runtime Assertion Checking and Trace Slicing. Technical report, Department of Computer Systems and Computation, Universitat Politècnica de València, 2015. Available at: `http://safe-tools.dsic.upv.es/abets/abets-tr.pdf`.
8. M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Exploring Conditional Rewriting Logic Computations. *Journal of Symbolic Computation*, 69:3–39, 2015.
9. M. Alpuente, S. Escobar, J. Espert, and J. Meseguer. A Modular Order-Sorted Equational Generalization Algorithm. *Information and Computation*, 235:98–136, 2014.
10. F. Baader and W. Snyder. Unification Theory. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 447–533. Elsevier Science, 2001.
11. R. Bruni and J. Meseguer. Semantic Foundations for Generalized Rewrite Theories. *Theoretical Computer Science*, 360(1–3):386–414, 2006.
12. L.A. Clarke and D.S. Rosenblum. A Historical Perspective on Runtime Assertion Checking in Software Development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, 2006.

13. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*. Springer-Verlag, 2007.
14. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude Manual (Version 2.6). Technical report, SRI International Computer Science Laboratory, 2011. Available at: `http://maude.cs.uiuc.edu/maude2-manual/`.
15. F. Durán and J. Meseguer. A Maude Coherence Checker Tool for Conditional Order-Sorted Rewrite Theories. In *Proceedings of the 8th International Workshop on Rewriting Logic and Its Applications (WRLA 2010)*, volume 6381 of *Lecture Notes in Computer Science*, pages 86–103. Springer-Verlag, 2010.
16. F. Durán, M. Roldán, and A. Vallecillo. Invariant-driven Strategies for Maude. *Electronic Notes in Theoretical Computer Science*, 124(2):17–28, 2005.
17. J. A. Goguen and J. Meseguer. Equality, Types, Modules, and (why not?) Generics for Logic Programming. *The Journal of Logic Programming*, 1(2):179–210, 1984.
18. J. A. Goguen and J. Meseguer. Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Oriented Programming*, pages 417–478. The MIT Press, 1987.
19. J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.
20. B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163, 1988.
21. J. L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, California, 1988.
22. G. T. Leavens and Y. Cheon. Design by Contract with JML, 2005. Available at: `http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf`.
23. N. Martí-Oliet, M. Palomino, and A. Verdejo. Rewriting Logic Bibliography by Topic: 1990–2011. *The Journal of Logic and Algebraic Programming*, 81(7–8):782–815, 2012.
24. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
25. J. Meseguer. Multiparadigm Logic Programming. In *Proceedings of the 3rd International Conference on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 158–200. Springer-Verlag, 1992.
26. C. Rocha, J. Meseguer, and C. Muñoz. Rewriting Modulo SMT and Open System Analysis. In *Proceedings of the 10th International Workshop on Rewriting Logic and its Applications (WRLA 2014)*, volume 8663 of *Lecture Notes in Computer Science*, pages 247–262. Springer-Verlag, 2014.
27. G. Roşu. From Rewriting Logic, to Programming Language Semantics, to Program Verification. In *Logic, Rewriting, and Concurrency - Festschrift Symposium in Honor of José Meseguer*, Lecture Notes in Computer Science. Springer-Verlag, 2015. This volume.
28. M. Roldán, F. Durán, and A. Vallecillo. Invariant-driven Specifications in Maude. *Science of Computer Programming*, 74(10):812–835, 2009.
29. TeReSe. *Term Rewriting Systems*. Cambridge University Press, 2003.