Document downloaded from:

http://hdl.handle.net/10251/81131

This paper must be cited as:

Alpuente Frasnedo, M.; Ballis, D.; Frechina, F.; Sapiña-Sanchis, J. (2016). Debugging Maude programs via runtime assertion checking and trace slicing. Journal of Logical and Algebraic Methods in Programming. 85(5):707-736. doi:10.1016/j.jlamp.2016.03.001.

The final publication is available at

http://dx.doi.org/10.1016/j.jlamp.2016.03.001

Copyright Elsevier

Additional Information

This is the author’s version of a work that was accepted for publication in <Journal title>. Changes resulting from the publishing process, such as peer review, editing, corrections, structural formatting, and other quality control mechanisms may not be reflected in this document. Changes may have been made to this work since it was submitted for publication. A definitive version was subsequently published in Journal of Logical and Algebraic Methods in Programming, [VOL 85, ISSUE 5, (2016)] DOI 10.1016/j.jlamp.2016.03.001.

Debugging Maude Programs via Runtime Assertion Checking and Trace Slicing[☆]

María Alpuente[a], Demis Ballis[b], Francisco Frechina[a], Julia Sapiña[a]

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

Abstract

In this paper we propose a dynamic analysis methodology for improving the diagnosis of erroneous Maude programs. The key idea is to combine runtime checking and dynamic trace slicing for automatically catching errors at runtime while reducing the size and complexity of the erroneous traces to be analyzed (i.e., those leading to states failing to satisfy some of the assertions). First, we formalize a technique that is aimed at automatically detecting deviations of the program behavior (symptoms) with respect to two types of user-defined assertions: functional assertions and system assertions. The proposed dynamic checking is provably sound in the sense that all errors flagged are definitely violations of the specifications. Then, upon eventual assertion violations we generate accurate trace slices that help identify the cause of the error. Our methodology is based on (i) a logical notation for specifying assertions that are imposed on execution runs; (ii) a runtime checking technique that dynamically tests the assertions; and (iii) a mechanism based on (equational) least general generalization that automatically derives accurate criteria for slicing from falsified assertions. Finally, we report on an implementation of the proposed technique in the assertion-based, dynamic analyzer ABETS and show how the forward and backward tracking of asserted program properties leads to a thorough trace analysis algorithm that can be used for program diagnosis and debugging.

Keywords: Trace Slicing, Runtime Checking, Dynamic Program Slicing, Program Diagnosis and Debugging, Rewriting Logic, Maude, Equational least general generalization

1. Introduction

Program debugging is crucial to reliable software development because the size and complexity of modern software systems make it almost impossible to avoid errors in their (requirements and design) specifications. Unfortunately, debugging is generally a burdensome process

[☆]This work has been partially supported by the EU (FEDER) and the Spanish MINECO under grants TIN2015-69175-C4-1-R and TIN2013-45732-C4-1-P, and by Generalitat Valenciana ref. PROMETEOII/2015/013. F. Frechina was supported by FPU-ME grant AP2010-5681, and J. Sapiña was supported by FPI-UPV grant SP2013-0083 and mobility grant VIIT-3946.

Email addresses: `alpuente@dsic.upv.es` (María Alpuente), `demis.ballis@uniud.it` (Demis Ballis), `ffrechina@dsic.upv.es` (Francisco Frechina), `jsapina@dsic.upv.es` (Julia Sapiña)

that takes up a large portion of the software development effort, with developers painfully going through volumes of execution traces to locate the actual cause of observable misbehaviors. In order to mitigate the costs of debugging, automated tools and techniques are required to help identify the root cause of (anticipated) errors. In this paper, we propose a general approach for the debugging of programs that is based on systematically combining runtime assertion checking and automated trace (and program) simplification.

Assertion checking is one of the most useful automated techniques available for detecting program faults. In runtime assertion checking, assertions are traditionally used to express conditions that should hold at runtime. By finding inconsistencies between specified properties and the program code, dynamic assertion checking can prove that the code is incorrect. Moreover, since an assertion failure usually reports an error, the user can direct his attention to the location at which the logical inconsistency is detected and (hopefully) trace the errors back to their sources more easily. Runtime assertion checking can also be useful in finding problems in the specifications themselves, which is important for keeping the specifications accurate and up-to-date. Although not universally used, assertions seem to have widely infiltrated common programming practice, primarily for finding bugs in the later stages of development. 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 [1].

Program slicing [2, 3] is another well-established activity in software engineering with increasing recognition in error diagnosis and program comprehension since it allows one to focus on the code fragment that is relevant to the piece of information (known as slicing criterion) that we want to track from a given program point. The basic idea of program slicing is to isolate 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 one corresponds to forward slicing. Work in this area has focused on the development of progressively more effective, useful, and powerful slicing techniques, which have been transferred to many application areas including program testing, software maintenance, and software reuse.

In order to cope with very complex distributed systems, tools and methods that can improve the early specification are key to the system development effort. Maude [4] is a high-performance language and system that provides a powerful variety of correctness tools and techniques including prototyping, state space exploration, and model checking of temporal formulas. Maude programs correspond to specifications in rewriting logic (RWL) [5], which is an extension of membership equational logic [6] that, besides supporting equations and allowing the elements of a type or *sort* to be characterized by means of membership axioms, adds rewrite rules that can be non-deterministic in order to represent transitions in a concurrent system. Thanks to its reflective design and meta-level capabilities, the Maude system provides powerful and highly efficient meta-programming facilities. This has further contributed to its success, giving support to the development of sophisticated tools and techniques for the modeling and analysis of Maude specifications, such as LTLR model checking [7], abstract certification [8], Web verification [9, 10], narrowing-based code-carrying theory [11], *etc.* (for a survey of the related literature, see [12]).

The use of slicing for debugging Maude programs is discussed in [13], and relies on a rich and highly dynamic parameterized scheme for exploring rewriting logic computations defined in [14, 15] that can significantly reduce the size and complexity of the runs under examina-

tion by automatically slicing both programs and computation traces. However, Maude does not currently provide general support for asserting properties that are dynamically-checked. Hence, 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 in the context of rewriting logic. We follow the approach of modern specification and verification systems such as Spec$^\sharp$ 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 [16]). 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, there can be good reasons not to use the code itself as a contract. Assertions can be seen as a form of lightweight, possibly incomplete or weaker specification embedded in the program text that may help developers identify program properties or behaviors to be preserved when modifying code. Independent assertions can also improve the effectiveness of tests, can be used as contracts to check the conformance of an implementation to its formal specification, and are key for static verification and automated test case generation. During the design process, they can simulate a design, allowing one to explore its properties before committing to the long development process. The advantages of equipping software with assertions are extensively discussed in [17].

In our 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 derive a dynamic program slice that preserves the program behavior for the considered program inputs [2]. In the proposed approach, assertions are *external* and evaluated at runtime whenever the state associated with the assertion is reached during execution. This use of assertions involves checking *individual* (finite) program executions as well as non-deterministic execution trees (up to a finite depth), rather than proving (or disproving) the correctness of every program execution.

1.1. Our contribution and plan of the paper

The paper is organized as follows. We begin by providing a brief introduction to rewriting logic and Maude, and we present the running example that we use throughout the paper: a conditional rewrite theory that models a simple (object-oriented), distributed, on-line car-rental store (§2). Then, we proceed with the main contributions of our work, which can be summarized as follows:

1. A simple assertion language for Maude programs that allows us to express properties that are both *executable* in user-defined programs and quite versatile, by including Boolean formulas that are specialized by means of *state templates* (i.e., a sort of term patterns). 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 properties concerning the system's execution to be expressed. We give semantics to

assertions by providing a specification for what it means for an equational simplification trace (resp. a system state) to satisfy functional (resp. system) assertions. This is a purely declarative specification that says which states and traces satisfy the given assertions without saying how satisfaction checks might be performed (§3).

2. An assertion-based trace slicing technique for simplifying rewriting logic computations. This technique exploits the information that is dynamically computed when an assertion fails (called *error symptoms*) to help correlate the simple external evidence of the error with the complexity of searching possible program locations for the faults that caused the error. More precisely, by exploiting the notion of *equational least general generalization* recently investigated in [18], we soundly combine the error symptoms computed from violated assertions with the trace slicing technique of [14, 15] to derive well-suited slicing criteria that automatically bootstrap the slicing process (§4).
3. A novel procedure for the dynamic analysis of rewriting logic computations, which relies on assertion-based trace slicing and improves the diagnosis of erroneous Maude programs. The correctness of our dynamic analysis is proved by Proposition 5.7 and Theorem 5.8 (§5). A refinement of the basic technique for inferring the slicing criteria is provided in Section 5.3.
4. An implementation of the (optimized) analysis methodology in the assertion-based, dynamic trace analyzer ABETS that is available at [19] (§6) and includes an experimental analysis of the system, followed by some conclusions and directions for future work (§7).

Unlike most other related work on dynamic assertion checking, the ABETS analyzer targets programs that may include sorts and subsorts, rules, equations, and equational axioms (i.e., algebraic laws such as commutativity, associativity, and unity). Furthermore, an important novel feature of ABETS is that it applies to Full Maude [4], 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. Full Maude not only complements Maude, but it can also be seen as an experimentation framework that allows new language features to be developed at the maturity level required to port it to Maude. Following the discussion above, this work can be seen as the first framework that exploits the synergies between runtime assertion checking and automated (program and program trace) transformations for improving the diagnosis of any program (or program tool) that is implemented in (Full) Maude. Moreover, the underlying foundation of ABETS developed in this article can be applied with little effort to other expressive rule-based languages like CafeOBJ, OBJ, ASF+SDF, and ELAN, which support: 1) rich type structures with sorts, subsorts and overloading; and 2) equational rewriting modulo axioms such as commutativity, associativity–commutativity, and associativity–commutativity–identity. Also, it is potentially applicable to any language whose operational semantics can be specified in Maude.

This article is a revised, improved, and extended version of [20].

1.2. Related work

Tools that are useful for mechanically checking that annotated programs meet their specifications fall into two main, complementary categories: runtime assertion checking (i.e., the testing of specifications during program execution, with any violation resulting in special errors being reported) and static verification (where logical techniques are used to prove, before

runtime, that no violations of specifications will take place at runtime). It was by the mid '70s when researchers realized that monitoring assertions during program execution offers a simple and practical counterpart to formal proofs of correction. Assertion checking cannot prove that a program is correct but it does support a greater degree of automation than deductive verification, i.e., static verification of the assertions using a theorem prover, which furthermore requires the user to have broad mathematical skills and provide fairly precise and complete specifications [21]. Runtime assertion checking does not face the same difficult challenges as, say, model-checking and theorem proving and is likely closer to becoming part of mainstream software development environments. The gist of runtime verification and its convenience as a partner of model-checking, theorem proving, and program testing is discussed in, e.g., [22, 23].

Initially developed as a means of stating expected or desired program properties as a necessary step in constructing formal, deductive proofs of program correctness, the key role of assertions in software engineering applications has witnessed the growth of assertion notations, such as JML, OCL, Spec♯, and Z, and assertion capabilities in widely used programming languages such as C♯, C++, Eiffel, and Java (see [1, 24, 25] and further references therein). In general, assertions are supported in programming languages in one of two ways —either incorporating assertion constructs into the design of the language, or by using an external assertion language that is injected into the target programming language through suitable software wrappers. Assertions are embedded in the type systems of many programming languages that support strong typing via type declarations, where a type restriction on arguments can be considered a precondition. Some languages support even stronger typing and subtyping assertions (e.g., Maude's membership axioms, which are used to automatically 'narrow' the type T of a value into a subtype of T). 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. The most obvious way to dynamically use assertions is to test them at runtime and report any detected violations. Yet assertions may be applied for automated error detection during any activity in which a program is executed, including debugging, testing, and operation.

Runtime verification (RV) is a light-weight formal technique that allows checking whether a *run* of a system under scrutiny satisfies or violates a given property [23], or more precisely, a(n) (informative) *finite prefix* of a run, i.e., a finite execution trace. Common properties include state-based properties such as preconditions, normal and exceptional post-conditions, invariants, and history constraints. One prominent feature of RV is its being performed at runtime, which opens up the possibility to act whenever incorrect software behavior is detected. Its distinguishing research effort lies in synthesizing (on-line/off-line) monitors from high-level specifications, where a monitor is a device that reads a finite trace and efficiently yields a certain verdict, typically a truth value from some truth domain. On-line monitors incrementally check the current execution of the system, while off-line monitors work on a (finite set of) recorded execution(s). The problem of generating monitors can be compared to the generation of automata in model-checking, where it finds its origins.

The use of contracts or assertions to obtain more reliable programs has been proposed for many programming languages and paradigms. This is a field that has a great amount of related work; here we can only summarize a small part that is most closely related to our work. A runtime checker written in Maude for the executable modeling language ABS is described in [21]. In functional programming, a semantics for dynamically checking contracts

was first formalized in [26]. Hybrid (mixed static/dynamic) contract checking for functional languages has received increasing research attention, as recently discussed in [27]. The notion of specifications and contracts for lazy functional (logic) programs is introduced in [28], where Curry is used as a single language for efficient implementations, executable specifications (describing the intended meaning of an operation as required for program verification), and contracts (run-time checked assertions consisting of both a pre- and a post-condition given as Boolean functions that can be weaker than a precise specification). In [28], post-conditions can be derived from existing program specifications in order to (hopefully) detect incorrect implementations. In contrast to our work, dynamic assertion checking is achieved by integrating the contract into the implementation, that is, all existing pre- and post-conditions are translated into correlated function conditions. Also different from our work, any result that a function produces must satisfy the function's (Boolean) post-condition, while we are able to discriminate among cases by specifying different state templates I and conditions φ_{in} in functional assertions $I\{\varphi_{in}\} \Rightarrow O\{\varphi_{out}\}$.

The Maude Formal Environment (MFE) is a recent effort to integrate and interoperate most of the available Maude analysis and verification tools [12]. It includes several program analyzers and theorem provers, which are all accessible in [29]. Other available tools in [29] are not yet integrated, such as the declarative debugger of Maude [30] and Maude's model checkers [4, 7]. The declarative debugger is based on Shapiro's algorithmic debugging technique [31] and supports the debugging of wrong results (erroneous reductions, sort inferences, and rewrites) and incomplete results (not completely reduced normal forms, greater than expected least sorts, and incomplete sets of reachable terms) in object-oriented, parameterized modules written in (Full) Maude. The declarative debugging process starts from a computation that is considered incorrect by the user (unexpected outcome) and tries to locate a program fragment that is responsible for that error symptom. The tool builds a debugging tree that represents the anomalous computation and guides the user while he/she explores the tree to find the bug. Moreover, the debugger offers the user several options to prune and traverse the tree. During the process, the system asks questions to an external oracle (typically the user) until a so-called buggy node is found (i.e., a node that contains an erroneous result but whose children have all correct results). Since a buggy node produces an erroneous output from correct inputs, it corresponds to an erroneous fragment of code that is pointed out as an error. Typical questions to the user have the form "Is it correct that term t rewrites (or fully reduces) to t'?" When the debugging tree is large, a main drawback is the frequency, size, and complexity of the questions to the oracle; hence, the tool allows some statements (and even whole modules) to simply be trusted in order to alleviate the process. We believe the slicing capabilities described in this article could be of great help to further shorten the declarative debugging process, avoiding unnecessary questions to the user while allowing the user to identify the very buggy components within relatively large nodes.

To the best of our knowledge, no general built-in support is provided in the MFE for runtime assertion checking or related disciplines such as *contract enforcement* in order to monitor *contract fulfillment* or enforce some penalty when a contract violation is observed. Related to our work, a generic strategy is defined in [32] to guarantee in Maude 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 [32] never results in violated assertions, which is essential in our approach for automatic trace slicing to be fired. As another difference, we are able to check assertions that regard the normalizations carried out by using the equational part of the rewriting theory. In [33], a dynamic validator of OCL constraints (class invariants and method pre/post-conditions) is proposed that evaluates Maude prototypes of UML models where both, UML models and OCL expressions, are represented as Maude specifications. OCL is specially tailored to specify constraints or queries over UML model objects; that is, the constraints are used to give an exact description of the information contained in the UML models and the queries are used to analyze these models and to validate them. Although OCL is not specific to any programming language, it explicitly targets UML in the same way that JML is tied to Java. In contrast, our notation is independent from the target programming language or modeling language so that, by keeping our syntax close to Maude, assertions can be easily specified by any Maude developer who wants to analyze a Maude representation of any programs or models of interest.

Finally, a parametric trace slicing and monitoring methodology is formalized in [34]. This technique allows parametric execution traces (i.e., traces which contain events with parameter bindings) to be sliced and subsequently checked online with respect to parametric properties. It is worth noting that both the notion of execution trace of [34] and the accompanying slicing algorithm differ from ours. In our framework, an execution trace is a Maude computation consisting of a rich combination of rule, equation, membership and axiom applications that is sliced by tracking backwards the relevant symbols of an automatically synthesized slicing criterion, whereas [34] defines traces as sequences of parametric events that are distributed into the corresponding trace slices by analyzing their associated parameter values.

2. Rewriting Logic and Maude

Let us recall some important notions that are relevant to this work (for deeper details, we refer to [4]).

We assume some basic knowledge of term rewriting [35] and rewriting logic [5], which is a logical framework that is particularly suitable for dealing with highly non-deterministic concurrent systems and computations. Some familiarity with the Maude language [36] is also required. Maude is a rewriting logic 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 rules. Equations are used to express deterministic computations that lead to a unique final result, while rules naturally express concurrent, non-deterministic, and possibly non-terminating computations. 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. The kinds allow equivalent sorts[1] to be grouped together and, intuitively, can be considered as an *error supersort.* Therefore, terms (built over Σ) that have a kind but not a sort are understood to be undefined or error terms. By $\mathcal{T}(\Sigma)$, we specify the term algebra that includes all the ground terms built over Σ, while $\mathcal{T}(\Sigma, \mathcal{V})$ is the usual non-ground term algebra built over Σ and the set of variables $\mathcal{V}$. Each operator in Σ is defined along with its sort and axiom declaration using the Maude syntax:

$$\texttt{op}\ \mathit{opname} : s_0 \ldots s_n \to s\ [\mathit{axiom\ declaration}]\ .$$

where s_i, $i = 0, \ldots, n$, and s are sorts, and *axiom declaration* is a (possibly empty) list of equational attributes (e.g., `assoc` for associativity, `comm` for commutativity, `id` for identity) that denote the algebraic laws that the operator *opname* must obey. By default, declared operators adopt the prefix notation; however, the user can also specify mixfix operators, which is done by using underscores as place fillers for the input arguments. So, for instance, the declaration

```
op _+_ : Int Int -> Int [assoc comm id: 0] .
```

defines `+` as a binary mixfix operator that takes two integer numbers and returns an integer number. The operator `+` is also declared associative and commutative and its identity is the constant `0`.

A *position* w in a term t is represented by a sequence of natural numbers that addresses a subterm of t (Λ denotes the empty sequence, i.e., the root position). Given a term t, we let $\mathcal{P}os(t)$ denote the set of positions of t. By $\mathcal{VP}\mathrm{os(t)}$, we denote the set of variable positions of a term t. By notation $w_1.w_2$, we denote the concatenation of positions (sequences) w_1 and w_2. 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 $\mathcal{T}(\Sigma, \mathcal{V})$, which is equal to the identity except for a finite 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 [37]:

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

Given two terms 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 term t, by $\mathcal{V}ar(t)$, we denote the set of variables that occur in t. Given a binary relation $\leadsto$, we define the usual *transitive* (resp., *transitive and reflexive*) closure of $\leadsto$ by $\leadsto^+$ (resp., $\leadsto^*$).

[1]Two sorts are in the same equivalence class if and only if they belong to the same connected component. Sorts are user-defined and explicitly declared in Σ, while kinds are implicitly associated with equivalence classes of sorts.

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 equational 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 equational 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 or membership axiom), $\lambda, \rho \in \mathcal{T}(\Sigma, \mathcal{V})$, 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 as stated in the standard prelude of Maude. 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 the keywords `fmod` and `endfm`. Functional modules provide executable models for the specified equational theories.

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

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.

Intuitively, (Σ, E) allows system states to be formalized as terms, while rules in R specify general patterns that are used to model state transitions and allow the dynamics of the system to be specified. More specifically, the system evolves by applying the rules of the rewrite theory R to the system states by means of *rewriting modulo* E. An in-depth explanation of the operational semantics underlying RWL and Maude is summarized in Section 2.3.

Concurrent object-oriented systems can be defined in Full Maude by means of *object-oriented modules*, which are delimited by the keywords `omod` and `endom`. Object-oriented modules implicitly define sorts (i) `Oid` (for object identifier); (ii) `Cid` (for class identifier); (iii) `Object`; and (iv) `Msg` for messages (declared by using the keyword `msg`), which are used to model object message-passing.

Objects are represented as terms of the following form:

< O : C | $a_1 : v_1$, ... , $a_n : v_n$ >

where O is a term of sort Oid, C is a term of sort Cid, and $a_1 : v_1, ..., a_n : v_n$ is a list of attributes of sort Attribute that consist of an identifier a_i followed by its respective value v_i. Concurrent states of object-oriented systems are represented as a multiset (i.e., an associative and commutative list) of objects and messages.

Classes are defined by using the keyword class, followed by the name of the class, a bar, and a list of attribute declarations separated by commas:

class C | a_1 : $\langle Sort_1 \rangle$, ... , a_n : $\langle Sort_n \rangle$.

Class names are considered to be a particular case of sort names. Therefore, class inheritance is directly supported by Maude's order-sorted type structure. A subclass declaration is an expression of the form subclass C < C' where C and C' are the names of the classes. Multiple inheritance is also supported, allowing a class C to be defined as a subclass of several classes.

As for the object-oriented rules, they are similar to standard Maude rules, except that it is possible not to mention irrelevant object attributes (that is, attributes that play no role in the rule transition). Moreover, attributes on the left-hand side of the rule that are not mentioned on the corresponding right-hand side are assumed to be unchanged.

Example 2.1

The following Full Maude object system models the logic of a (faulty) distributed, object-oriented, online car-rental store, which is inspired by a specification in [36].

```
(omod RENT-A-CAR-ONLINE-STORE is
 pr CONVERSION .
 pr QID .

 subsort Qid < Oid .

 class Register | date : Nat , rentals : Nat .
 class Customer | credit : Int , suspended : Bool .
 class Car | available : Bool , rate : Nat .
 class Rental | deposit : Nat , dueDate : Nat , pickUpDate : Nat , customer : Oid , car : Oid .

 class PreferredCustomer .
 subclass PreferredCustomer < Customer .

 class EconomyCar .
 class MidSizeCar .
 class FullSizeCar .
 subclasses EconomyCar MidSizeCar FullSizeCar < Car .

 vars U C R RG : Oid .
 vars CREDIT AMNT : Int .
 vars TODAY PDATE DDATE RATE DPST RNTLS : Nat .

 rl [new-day] : < RG : Register | date : TODAY > => < RG : Register | date : TODAY + 1 > .
```

```
crl [3-day-rental] :
--- Faulty rule: customer's credit is not checked before renting.
  < U : Customer | credit : CREDIT , suspended : false >
  < C : Car | available : true , rate : RATE >
  < RG : Register | date : TODAY , rentals : RNTLS >
  =>
  < U : Customer | credit : CREDIT - AMNT >
  < C : Car | available : false >
  < RG : Register | rentals : RNTLS + 1 >
  < qid("R" + string(RNTLS,10)) : Rental | pickUpDate : TODAY , dueDate : TODAY + 3 , car : C ,
  deposit : AMNT , customer : U , rate : RATE >
  if AMNT := 3 * RATE .

crl [on-date-return] :
  < U : Customer | credit : CREDIT >
  < C : Car | rate : RATE >
  < R : Rental | customer : U , car : C , pickUpDate : PDATE , dueDate : DDATE , deposit : DPST >
  < RG : Register | date : TODAY >
  =>
  < U : Customer | credit : (CREDIT + DPST) - AMNT >
  < C : Car | available : true >
  < RG : Register | >
  if (TODAY <= DDATE) /\ AMNT := RATE * (TODAY - PDATE) .

crl [late-return] :
  < U : Customer | credit : CREDIT >
  < C : Car | rate : RATE >
  < R : Rental | customer : U , car : C , pickUpDate : PDATE , dueDate : DDATE , deposit : DPST >
  < RG : Register | date : TODAY >
  =>
  updateSuspension(< U : Customer | credit : (CREDIT - AMNT) + DPST >)
  < C : Car | available : true >
  < RG : Register | >
  if DDATE < TODAY /\ AMNT := RATE * (DDATE - PDATE) +
      (120 * RATE * (TODAY - DDATE)) quo 100 .

op updateSuspension : Object -> Object .
  ceq [suspend] :
  --- Faulty equation: preferred customers in debt can be suspended, which is not intended.
    updateSuspension(< U : Customer | credit : CREDIT , suspended : false >) =
    < U : Customer | credit : CREDIT , suspended : true >
    if (CREDIT < 0) .

   eq [maintainSuspension] :
       updateSuspension(< U : Customer | suspended : B:Bool >) =
       < U : Customer | suspended : B:Bool > [owise] .
endom)
```

Here, each state of the system is modeled as a multiset of objects $\mathtt{e_1}\ \mathtt{e_2}\ \ldots\ \mathtt{e_n}$, where each $\mathtt{e_i}$ is one of the following: (i) a customer (who is registered at the store with a certain `credit`); (ii) a (rented or available) car; (iii) a renting contract; or (iv) the register, that models time elapsing and records the number of active car rentals. We consider two kinds of customers: standard customers and preferred customers (who are allowed to rent even if they run out of credit). Basic operations of the store (i.e., rental and return of cars) are

implemented via three rewrite rules: `3-day-rental`, `on-date-return`, and `late-return`. The `3-day-rental` rule enables car rental only if the chosen car is `available` and, at the time when the contract is signed, the customer makes a `deposit` (that is subtracted from his credit) aimed to cover the estimated charge depending on the daily rental `rate` of the car. Note that the `3-day-rental` rule is flawed because it does not check if the current credit of the customer is sufficient to cover the requested deposit, which could lead to erroneous system behaviors. When a rented car is returned before the due date, the `on-date-return` rule is applied. In this case, the customer is reimbursed for the payment of the initial deposit and is only charged for the number of days he used the car. If the car is returned past the due date, the `late-return` rule is instead applied and the customer is charged an additional sanction that amounts to 20% of the established fee (for each day past the due date). Non-preferred customers are suspended when they reach a negative `credit`, i.e., if they have a debt. Defaulter (non-preferred) customer suspension is modeled by the equations `suspend` and `maintainSuspension`, which applies when the defaulter customer repeats infringement while already suspended. Note `late-return` rightly admits negative credit and deals with the issue by triggering the function `updateStatus` that suspends the debtor customers who are non-preferred. However, the equations for modeling suspension are erroneous because they cause preferred customers to be suspended as well, which is not what is intended.

Despite offering a more convenient syntax and conceptual advantages, object-oriented modules are just syntactic sugar and they are internally transformed into system modules for execution purposes. However, a side effect of this translation is that positions in object terms can be easily misinterpreted since each attribute `attrName` of sort `AttrSort` is automatically translated into a term of sort `Attribute` by dynamically introducing *ad-hoc* unary operators `` attrName`:_ :  AttrSort -> Attribute `` (one per attribute), as illustrated in the following example.

Example 2.2

Consider the object-oriented, *Full Maude* specification of Example 2.1 together with the following object:

```
< 'A1 : EconomyCar | available : true , rate : 30 >
```

that represents a car with identifier `'A1`, class `EconomyCar`, and the attributes `available` and `rate` (with values `true` and `30`, respectively).

Since object-oriented modules are automatically translated into system modules, the following source-level declarations of unary operators `` 'available`: `` and `` 'rate`: `` are dynamically created by Full Maude:

```
op 'available`:_ : 'Bool -> 'Attribute [ gather('&) ] .
op 'rate`:_ : 'Nat -> 'Attribute [ gather('&) ] .
```

A graphical, meta-level[2] representation of the transformed term is depicted in Figure 1.

[2]By using the meta-level notation, we can easily and unequivocally identify the arity of each operator, with the underscores indicating the exact place of its arguments in the mixfix notation.

Note that the value `true` of the attribute `available` is addressed by position 3.1.1, which is not obvious by only looking to the source-level representation of the original term.

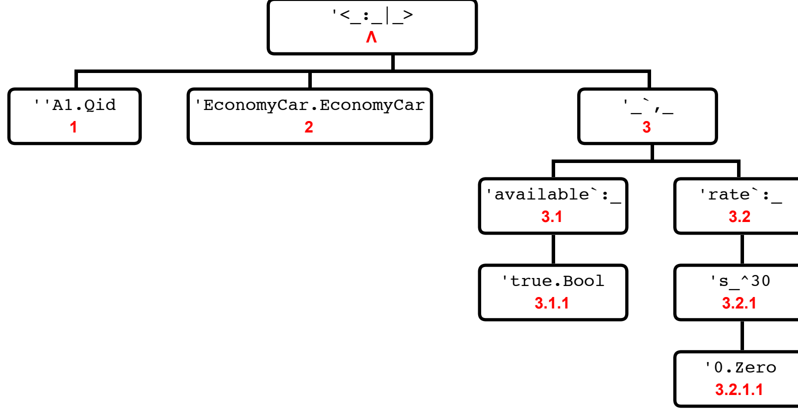

Figure 1: Positions of object `< 'A1 : EconomyCar | available : true , rate : 30 >`.

Applying our techniques to Full Maude is thus quite simple, but implementing them in ABETS in a way that is easy to use in practice was much trickier than we expected, as we briefly mention in Section 6, where we describe the graphical analyzer ABETS, which totally relieves users from the burden of textuallly dealing with positions.

A thorough description of Full Maude object-based programming can be found in [4].

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 [38] to the E-congruence classes $[t]_E$ on the term algebra $\mathcal{T}(\Sigma, \mathcal{V})$ that are induced by $=_E$ [39]. 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 $[t']_E$.

For a conditional rewrite theory to be executable, its equations Δ should be Church-Rosser and terminating modulo the given axioms B, and their rules R should be (ground) coherent with Δ modulo B. This allows the Maude interpreter to implement conditional rewriting $\rightarrow_{R/E}$ with R modulo E by means of two much simpler relations, namely $\rightarrow_{\Delta,B}$ and $\rightarrow_{R,B}$, that allow rules, equations and memberships to be intermixed in the rewriting process by simply using an algorithm of matching modulo B. The relation $\rightarrow_{R\cup\Delta,B}$ is defined 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 with respect to 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 Church-Rosser and terminating modulo B in order to guarantee the existence and unicity (modulo B) of a canonical form with respect to Δ for any term [36]. 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`] : $\lambda = \rho$ `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 $\mathtt{C}\sigma$ *evaluates to true*. 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 $\mathtt{C}\sigma$ 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 $t = t'$ *evaluates to true* if $\mathtt{t}\downarrow_{\Delta,B} =_B \mathtt{t}'\downarrow_{\Delta,B}$; a matching equation $\mathtt{p} := \mathtt{t}$ *evaluates to true* if $\mathtt{p} =_B \mathtt{t}\downarrow_{\Delta,B}$; a rewrite expression $\mathtt{t} \Rightarrow \mathtt{p}$ *evaluates to true* if there exists a rewrite sequence $\mathtt{t} \rightarrow^{*}_{R\cup\Delta,B} \mathtt{u}$ such that $\mathtt{u} =_B \mathtt{p}$ [3]; and, finally, a membership `t : s` *evaluates to true* if `t` has sort `s`.

Under appropriate coherence conditions [40] 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 involves the repeated application of the two following basic steps [40]:

1. **Equational simplification of s in Δ modulo** B, that is, reduce s using $\rightarrow_{\Delta,B}$ until the canonical form with respect to Δ 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 $t' \in [t]_E$.

An *execution trace* (or *computation*) $\mathcal{C}$ 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,

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

$i = 1, \ldots, n$, 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. Therefore, any computation can be interpreted as a sequence of juxtaposed $\rightarrow_{R,B}$ and $\rightarrow^*_{\Delta,B}$ transitions, with an additional equational simplification $\rightarrow^*_{\Delta,B}$ (if needed) at the beginning of the computation as depicted below.

$$\overbrace{s_0 \rightarrow^*_{\Delta,B} s_0{\downarrow}_{\Delta,B} \rightarrow_{R,B} s_1 \rightarrow^*_{\Delta,B} s_1{\downarrow}_{\Delta,B}}\; \underbrace{\rightarrow_{R,B} s_2 \rightarrow^*_{\Delta,B} s_2{\downarrow}_{\Delta,B} \cdots}$$

By coercion, any term in canonical form that cannot be further rewritten via $\rightarrow_{R,B}$ is also considered to be a 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 possible *Maude steps* stemming from s in R. Finally, by $length(\mathcal{C})$ we define the number of Maude steps that are contained in the computation $\mathcal{C}$.

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. For the free theory, the lgg of a pair of terms is unique up to variable renaming [41].

In [42, 43, 18], 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 generalization has at least one of them as an instance. Given any element g of the set $lgg_E(t_1, t_2)$, we define the function π from $\mathcal{VP}os(g)$ 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 equational 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 `f` and the terms $t_1 = \mathtt{f(b,c,a)}$ and $t_2 = \mathtt{f(d,a,b)}$. Then, a possible lgg modulo the associativity and commutativity of `f` is $(\mathtt{f(a,b,X)}, \{\mathtt{X/c}\}, \{\mathtt{X/d}\}) \in lgg_E(t_1, t_2)$, where `X` is a variable. Note that both t_1 and t_2 are syntactically different from $\mathtt{f(a,b,X)}$, 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 (i.e., Maude's typing and subtyping assertions given by the membership axioms). 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. Assertions act as an oracle, giving a pass/fail indication to program runs. 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*). A state s is simplified into its canonical form $s \downarrow_{\Delta,B}$ by using equations and equational/membership axioms in $\Delta \cup B$.

In our specification language, assertions are not mere Boolean expressions but truly *executable* formulas that are built on user-defined functions and specialized by means of state patterns. 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 non-deterministic rule component of $\mathcal{R}$. The benefit of the logic framework being integrated into our Maude specification and analysis environment 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 (membership) 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` [36], 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 `_=/=_`.

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 (modulo the equational axioms) 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 axioms) consisting of an *exclusive or* of conjunctions. By default, the `BOOL` module is implicitly imported as a submodule of any other user-defined module.

Predicates that are not specified in `BOOL` are module-dependent and can be equationally defined as total Boolean functions over the system entities (e.g., states, function calls) formalized within $\mathcal{R}$. In the same spirit of Maude's equational theories, where a single result is expected to be delivered for each input term, we require the user to ensure that the evaluation (i.e., the equational simplification) of any property terminates for any possible initial state and that the resulting verdict is unique.

In the proposed framework, basic properties on a given rewrite theory $\mathcal{R}$ are defined by means of a system module `PRED`($\mathcal{R}$) that

```
mod RENT-A-CAR-PRED is
  inc RENT-A-CAR-ONLINE-STORE .
  op isPreferredCustomer : Cid -> Bool .
    eq isPreferredCustomer(PreferredCustomer) = true .
    eq isPreferredCustomer(U:Cid) = false [owise] .
  op isFullSize : Object -> Bool .
    eq isFullSize(< C:Oid : FullSizeCar | available : B:Bool ,
                    rate : RATE:Nat >) = true .
    eq isFullSize(< C:Oid : Car | available : B:Bool ,
                    rate : RATE:Nat >) = false [owise] .
endm
```

Figure 2: System properties specified by the `RENT-A-CAR-PRED` module.

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

Note that, the system module `PRED`($\mathcal{R}$) must fulfill the same properties of $\mathcal{R}$, that is, its rewrite rules must be coherent with respect to its equations (modulo the equational axioms), and its embedded, extended equational theory, which includes the equational definition of P, must be terminating and Church-Rosser (modulo the equational axioms). 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 3.1

Consider the `RENT-A-CAR-ONLINE-STORE` object module of Example 2.1 and the new predicate `isFullSize` given in the `RENT-A-CAR-PRED` module of Figure 2. Then, we can specify the formula

```
isFullSize(< O:Oid :  C:Cid | available :  true , rate :  RATE:Nat >)
         implies RATE:Nat >= 70
```

which is true for every `FullSizeCar` object with an `available` attribute set to true and a `rate` attribute greater than or equal to 70.

3.2. System and Functional Assertions

System assertions formalize invariant properties over (portions of) system states. We define a system assertion as a *constrained term* $S\{\varphi\}$ [44], where φ is an order-sorted, quantifier-free Boolean formula that is specialized to a (possibly non–ground) term S in $\mathcal{T}(\Sigma, \mathcal{V})$ of sort `State`, with $\mathcal{V}ar(\varphi) \subseteq \mathcal{V}ar(S)$, which we call *state template*.

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 with respect to the formula φ. More formally, we define the satisfaction of a system assertion in a system state as follows.

Definition 3.2 (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 $\mathcal{T}(\Sigma, \mathcal{V})$. 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 3.3 (system error symptoms) *The set of all system error symptoms for a state s and a system assertion $S\{\varphi\}$ is defined as follows:*

$$\xi_{sys}(s, S\{\varphi\}) = \{w \mid \exists\sigma.\ s|_w =_E S\sigma, w \in \mathcal{P}os(s),\ \textit{and}\ \mathcal{R} \not\models \varphi\sigma\}.$$

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

Example 3.4

Consider the extended rewrite theory of Example 3.1 together with the system assertion

$\Theta =$
```
< O:Oid : C:Cid | credit : B:Int , suspended : S:Bool >
{ not(isPreferredCustomer(C:Cid)) implies B:Int >= 0 }
```

Then, Θ is satisfied in the state

```
< 'A5 : FullSizeCar | available : true , rate : 70 >
< 'C1 : Customer | credit : 50 , suspended : false >
< 'RG : Register | date : 0 , rentals : 0 >
```

but it is not satisfied in

$s_{err} =$
```
< 'A5 : FullSizeCar | available : false , rate : 70 >
< 'C1 : Customer | credit : - 160 , suspended : false >
< 'R0 : Rental | car : 'A5 , customer : 'C1 , deposit : 210 ,
dueDate : 3 , pickUpDate : 0 , rate : 70 >
< 'RG : Register | date : 0 , rentals : 1 >
```

since non-preferred customer `'C1` has a negative credit. The computed error symptom is the position 2 that refers to the subterm

```
< 'C1 : Customer | credit : - 160 , suspended : false >
```

of the anomalous state s_{err}.

The second type of assertions that we consider are *functional assertions*. Roughly speaking, functional assertions are implicative formulas between two constrained terms $I\ \{\varphi_{in}\}$ and $O\ \{\varphi_{out}\}$ that specify the general pattern O of the canonical form for any input term t that matches the given template I, while allowing pre- and post-conditions $\varphi_{in}, \varphi_{out}$ over the equational simplification to also be declared. Formally, their general form is $I\ \{\varphi_{in}\} \to O\ \{\varphi_{out}\}$ where $I, O \in \mathcal{T}(\Sigma, \mathcal{V})$, $\varphi_{in}, \varphi_{out}$ are 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 two ingredients:

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 $I\ \{\varphi_{in}\} \to O\ \{\varphi_{out}\}$ remind Reachability Logic (RL) formulas $\varphi \Rightarrow \varphi'$, where φ, φ' are ML patterns (for a survey on ML/RL, see [45]). In contrast to our functional assertions, which predicate on equational simplifications, RL formulas are evaluated on system computations: 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. Nevertheless, we would like to recall that our assertions are quantifier-free and can be efficiently evaluated by relying on Maude standard infrastructure (such as `metaReduce`, `metaMatch`, and `metaNormalize` commands).

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

Definition 3.5 (functional assertion satisfaction) *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, with $E = \Delta \cup B$. Let $I\ \{\varphi_{in}\} \to O\ \{\varphi_{out}\}$ be a functional assertion for $\mathcal{R}$, and μ be the equational simplification of the term t in $\mathcal{T}(\Sigma, \mathcal{V})$ into its canonical form $t\downarrow_{\Delta,B}$ with respect to Δ modulo B. Then, $I\{\varphi_{in}\} \to O\{\varphi_{out}\}$ is* satisfied *in μ (in symbols, $\mu \models I\{\varphi_{in}\} \to O\{\varphi_{out}\}$) iff for each substitution σ_{in} such that $t =_B I\sigma_{in}$, 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 $\mathcal{R}$.*

The satisfaction of functional assertions could be equivalently defined on the call term t (rather than on its equational simplification $\mu : t \to^*_{\Delta,B} t\downarrow_{\Delta,B}$) since the normal form $t\downarrow_{\Delta,B}$ is uniquely defined in a canonical equational theory. Nonetheless, we prefer to define the satisfaction with respect to μ since we believe that this notion is much closer to the intuitive meaning of functional assertions (whose satisfiability depends on both the input term t and the reduced term $t\downarrow_{\Delta,B}$ of μ). Therefore, using μ greatly simplifies our description. Given the set P of new user-defined predicates and their equational definition Q, we could split the set of all functions defined in the extended equational theory $E \cup Q$ into two disjoint sets, $U \oplus T$, where U are the untrusted functions of E (those to be debugged) and T is the extension of P with the set of all trusted functions defined in E. Now, requiring that T includes all

functions allowed in admissible functional assertions plus the functions they depend on (which can be easily approximated by analyzing the graph of functional dependencies of the extended theory), we do not even need the canonicity of the whole equational theory $E \cup Q$; we only need the canonicity of the sub-theory that defines the trusted set T. We do not formalize this generalization in order to keep our description simple.

Note that $I\ \{\varphi_{in}\} \rightarrow O\ \{\varphi_{out}\}$ is (trivially) satisfied in μ when either t does not match I (modulo B), or $t =_B I\sigma_{in}$ and $\varphi_{in}\sigma_{in}$ 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} (i.e., a substitution that matches t within the input template I modulo B; in symbols, $t =_B I\sigma_{in}$) such that

- $\varphi_{in}\sigma_{in}$ 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 3.6 (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 with respect to Φ is any position in $\mathcal{P}os(t\downarrow_{\Delta,B})$ that belongs to the following set:*

$$\xi_{fun}(\mu,\Phi) = \begin{cases} \{\pi(w) \mid ((g,\sigma_1,\sigma_2),\pi) = \widehat{lgg}_B(t\downarrow_{\Delta,B}, O(\sigma_{in}\downarrow_{\Delta,B})) \text{ and } w \in \mathcal{VP}os(g)\} & (1)\\ \qquad\qquad \text{if } \nexists\sigma_{out} \text{ s.t. } t\downarrow_{\Delta,B} =_B O(\sigma_{in}\downarrow_{\Delta,B})\sigma_{out} & \\ \{\Lambda\} \qquad \text{if } \forall\sigma_{out} \text{ s.t. } t\downarrow_{\Delta,B} =_B O(\sigma_{in}\downarrow_{\Delta,B})\sigma_{out}, \mathcal{R} \not\models \varphi(\sigma_{in}\downarrow_{\Delta,B})\sigma_{out} & (2) \end{cases}$$

Roughly speaking, $\xi_{fun}(\mu,\Phi)$ is computed by distinguishing two cases.

Case (1) If no matching substitution σ_{out} exists that allows the canonical form $t\downarrow_{\Delta,B}$ to be matched within the instance $O(\sigma_{in}\downarrow_{\Delta,B})$ of the output template O by the (normalized) substitution $\sigma_{in}\downarrow_{\Delta,B}$, we "compare" $t\downarrow_{\Delta,B}$ with $O(\sigma_{in}\downarrow_{\Delta,B})$ by 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 $\widehat{lgg}_B$, and 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 $w \in \mathcal{VP}os(g)$. The intuition behind this method is that variables in g reflect the discrepancies between the computed canonical form and the instantiated output template, and therefore subterms $(t\downarrow_{\Delta,B})|_{\pi(w)}$ represent anomalies in $t\downarrow_{\Delta,B}$.

Case (2) If for every matcher (modulo B) σ_{out} of the computed canonical form $t\downarrow_{\Delta,B}$ in $O(\sigma_{in}\downarrow_{\Delta,B})$, the (instantiated) formula $\varphi(\sigma_{in}\downarrow_{\Delta,B})\sigma_{out}$ does not hold in $\mathcal{R}$, then $t\downarrow_{\Delta,B}$ does not meet the property φ and its root position (which identifies the whole erroneous term) is signalled as a functional error symptom. Note that, in this case, the detection of the error source could be only roughly approximated, since the whole computed canonical

form is considered faulty, even though only some parts could be responsible for the error. A more practical, and refined symptom detection algorithm is provided in Section 5.3 that allows error sources to be located more precisely.

Example 3.7 ____

Consider again the extended rewrite theory of Example 3.1. Then, the functional assertion

```
Φ =  updateSuspension(< U:Oid : PreferredCustomer | credit : B:Int ,
     suspended : false >) { B:Int < 0 }
  →  < U:Oid : PreferredCustomer | credit : B:Int ,
     suspended : false > { true }
```

states that, for preferred customers, the `suspended` flag (and other customer attributes) remain unchanged after `updateSuspension` is invoked. Roughly speaking, preferred customers are never suspended, even if they were slow payers. Thus, Φ is not satisfied in the following equational simplification

```
          updateSuspension(< 'C1 : PreferredCustomer | credit : - 25 ,
          suspended : false >)
suspend
  --->
          < 'C1 : PreferredCustomer | credit : - 25 , suspended : true >
```

with input substitution $\sigma_{in} = \{$ `U/'C1, B/- 25`$\}$. The violation of Φ corresponds to case (1) of Definition 3.6 since the computed canonical form for the `updateSuspension` function call does not match the instantiation of the output template Φ with $\sigma_{in}\downarrow_{\Delta,B}$ (which is equal to σ_{in} in this case).

Hence, we compute the only (actually syntatical) least general generalization

```
lgg^_B(< 'C1 : PreferredCustomer | credit : - 25, suspended : true >,
   < U:Oid : PreferredCustomer | credit : B:Int, suspended : false >(σ_in ↓Δ,B))
   = ((< 'C1 : PreferredCustomer | credit : - 25, suspended : X:Bool >,
      {X/true},{X/false}),{3.2.1 ↦ 3.2.1})
```

where $\xi_{fun}(\mu, \Phi) = \{3.2.1\}$ is the set of functional error symptoms that pinpoint the anomalous `suspended` flag value in `'C1`'s data structure, that is,

```
< 'C1 : PreferredCustomer | credit : - 25 , suspended : true >|3.2.1 = true.
```

Now, consider this slight mutation of the assertion Φ

```
Φ' =  updateSuspension(< U:Oid : PreferredCustomer | credit : B:Int ,
      suspended : S:Bool >) { B:Int < 0 }
   →  < U:Oid : PreferredCustomer | credit : B:Int ,
      suspended : S':Bool > { S:Bool == S':Bool }
```

whose post-condition explicitly states that `updateSuspension` calls cannot change the value of the `suspended` flag. Also Φ' is not satisfied in the equational simplification above, but,

in this case, the reason of the violation stands in the refutation of the (instantiated) post-condition, which corresponds to case (2) of Definition 3.6. Therefore, our methodology delivers $\xi_{fun}(\mu, \Phi') = \{\Lambda\}$, thereby providing a less precise error detection analysis that marks the whole computed canonical form

```
< 'C1 : PreferredCustomer | credit : - 25 , suspended : true >
```

as incorrect.

Remarkably, in Section 5.3 we introduce an optimization technique that allows the same, more refined error symptom set $\{\texttt{3.2.1}\}$ to be also computed for Case 2 with the modified functional assertion Φ'.

It is worth noting that the use of $\widehat{lgg}_B$ 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 3.8

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

Then, $(\texttt{f(0,0)}, \texttt{c(1,3)}) \not\models \Phi$ (with input substitution $\sigma_{in} = \{\texttt{X/0}, \texttt{Y/0}\}$), since variable `Z` in the output template `c(Z,1)` is bound to `3` and `even(3)` is false. Then, $\widehat{lgg}_B(\texttt{c(1,3)}, \texttt{c(Z,1)})$ returns a pair $((g, \sigma_1, \sigma_2), \pi)$ such that g contains the minimum number of variables. For instance, $\widehat{lgg}_B(\texttt{c(1,3)}, \texttt{c(Z,1)}) = ((\texttt{c(Z,1)}, \{\texttt{Z/3}\}, \{\}), \{1 \mapsto 2\})$ and $\xi_{fun}(\mu, \Phi) = \{\texttt{2}\}$, which precisely detects that the term $\texttt{c(1,3)}|_2 = \texttt{3}$ 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 $(\texttt{c(Z,W)}, \{\texttt{Z/1}, \texttt{W/3}\}, \{\texttt{W/1}\})$ and the larger functional error symptom set $\{\texttt{1}, \texttt{2}\}$ (which represents the positions of both arguments of the canonical form `c(1,3)`), thereby hindering the isolation of the erroneous subterm of `c(1,3)`.

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 [13, 46, 47, 48] 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 with respect to 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 $Tslice(t, P)$, which allows a term slice of t to be constructed with respect to a set of positions P of t. The function *Tslice* relies on the function $fresh^\bullet$ whose invocation returns a (fresh) variable $\bullet_i$ of appropriate sort that is distinct from any previously generated variable $\bullet_j$.

Definition 4.1 (Term Slice) *Let $t \in \mathcal{T}(\Sigma, \mathcal{V})$ be a term and let P be a set of positions such that $P \subseteq \mathcal{P}os(t)$. Then, the term slice $Tslice(t, P)$ of t with respect to P is computed as follows.*

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

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

and $\bar{P} = \{u \mid u \leq p \wedge p \in P\}$ is the prefix closure *of P. Note that the inductive case ($n = 0$) includes the case when f is a 0-ary function symbol; hence, $f(recslice(\emptyset, P, p)) = f$.*

Roughly speaking, the function $Tslice(t, P)$ yields a term slice of t with respect to a set of positions P that includes all (and only the) symbols of t occurring within the access paths from the root of t to each position in P, while the remaining information of t is abstracted by means of $\bullet$-variables.

Example 4.2

Consider the specification of Example 2.1 and the state

```
< 'A1 : EconomyCar | available : true , rate : 20 > < 'RG : Register | rentals : 0 ,
date : 0 >
```

Consider the set $P = \{1.1,\ 1.2,\ 1.3.1,\ 1.3.2\}$ of positions in t. Then,

$Tslice(t, P) =$ `< 'A1 : EconomyCar | available :` $\bullet_1$ `, rate :` $\bullet_2$ `>` $\bullet_3$

Trace slicing can be carried out forwards or backwards. 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.

Throughout this paper, we assume the existence of a $backwardSlicing(s_0 \to^*_{\Delta\cup B} s_n, s^\bullet_n)$ function as defined in [13] that yields the backward trace slice $s^\bullet_0 \bullet\!\!\to^* s^\bullet_n$ of the computation trace $s_0 \to^*_{\Delta\cup B} s_n$ with respect to a term slice $s^\bullet_n$ of s_n, which is called the *slicing criterion*. 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 (which may cause the components of the terms to be implicitly reordered in the original trace). Also, the dynamic dependencies exposed by backward trace slicing are exploited in [13] 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.

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 4.3

Consider the `RENT-A-CAR-ONLINE-STORE` object module of Example 2.1 and the computation trace $\mathcal{C}_{rent} = s_0 \stackrel{\texttt{3-day-rental}}{\longrightarrow} s_1 \stackrel{\texttt{3-day-rental}}{\longrightarrow} s_2$ that starts in the initial state

```
s0 = < 'A1 : EconomyCar | available : true , rate : 30 >
     < 'A5 : FullSizeCar | available : true , rate : 70 >
     < 'C1 : Customer | credit : 50 , suspended : false >
     < 'C2 : PreferredCustomer | credit : 100 , suspended : false >
     < 'RG : Register | date : 0 , rentals : 0 >
```

and ends in the state

```
s2 = < 'A1 : EconomyCar | available : false , rate : 30 >
     < 'A5 : FullSizeCar | available : false , rate : 70 >
     < 'C1 : Customer | credit : - 160 , suspended : false >
     < 'C2 : PreferredCustomer | credit : 10 , suspended : false >
     < 'R0 : Rental | car : 'A1 , customer : 'C2 , deposit : 90 ,
       dueDate : 3 , pickUpDate : 0 , rate : 30 >
     < 'R1 : Rental | car : 'A5 , customer : 'C1 , deposit : 210 ,
       dueDate : 3 , pickUpDate : 0 , rate : 70 >
     < 'RG : Register | date : 0 , rentals : 2 >
```

Roughly speaking, $\mathcal{C}_{rent}$ models the following actions[4] : (i) customer `'C2` subscribes a 3-day rental contract (rule `3-day-rental`) to rent an economy car whose rate is 30 and his/her credit

[4]For the sake of clarity, we have intentionally omitted, from $\mathcal{C}_{rent}$, all of the built-in equational simplifications that are needed to simplify arithmetic expressions.

is reduced by 90, (ii) customer `'C1` subscribes a 3-day rental contract (rule `3-day-rental`) to rent a full size car whose rate is 70 and his/her credit is reduced by 210.

Let us assume we manually define as the slicing criterion the negative credit `- 160` for customer `'C1`, which indicates a possible malfunction of the `RENT-A-CAR-ONLINE-STORE` specification since the regular client credit must be non-negative according to the semantics intended by the programmer. Therefore, we execute trace slicing on the trace $\mathcal{C}_{rent}$ with respect to the slicing criterion $\bullet_1$ $\bullet_2$ `<` $\bullet_3$ `:` $\bullet_4$ `| credit : - 160 ,` $\bullet_5$ `>` $\bullet_6$ $\bullet_7$ $\bullet_8$ $\bullet_9$ that allows for the observation of `'C1`'s negative credit.

By applying the backward trace slicing technique to $\mathcal{C}_{rent}$ with respect to $s_3^\bullet$, we get the output trace slice $\mathcal{C}_{rent}^\bullet$:

$$
\begin{array}{ll}
 & \bullet_{13}\ \texttt{<}\ \bullet_{10}\ \texttt{:}\ \bullet_{11}\ \texttt{| available : true , rate : 70 >} \\
 & \texttt{<}\ \bullet_3\ \texttt{:}\ \bullet_4\ \texttt{| credit : 50 , suspended : false >}\ \bullet_{14}\ \bullet_{15} \\
\stackrel{\texttt{3-day-rental}}{\bullet\!\!\to} & \\
 & \bullet_1\ \texttt{<}\ \bullet_{10}\ \texttt{:}\ \bullet_{11}\ \texttt{| available : true , rate : 70 >} \\
 & \texttt{<}\ \bullet_3\ \texttt{:}\ \bullet_4\ \texttt{| credit : 50 , suspended : false >}\ \bullet_6\ \bullet_7\ \bullet_{12} \\
\stackrel{\texttt{3-day-rental}}{\bullet\!\!\to} & \\
 & \bullet_1\ \bullet_2\ \texttt{<}\ \bullet_3\ \texttt{:}\ \bullet_4\ \texttt{| credit : - 160 ,}\ \bullet_5\ \texttt{>}\ \bullet_6\ \bullet_7\ \bullet_8\ \bullet_9
\end{array}
$$

which greatly simplifies the trace $\mathcal{C}_{rent}$ by showing the origins of the observed negative credit while excluding all the objects and attributes that are not related to `'C1`'s credit.

Indeed, by observing the first sliced state in $\mathcal{C}_{rent}^\bullet$, we can easily verify that the conditions for the rental are met by customer `'C1` and car `'A5`. In particular, `'A5` is available and (non-preferred) customer `'C1` is not suspended. However, the car should not be rented because the credit `50` does not cover the charge `210` (`70` for each day), which causes the negative credit `-160` of customer `'C1`.

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 initiate 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 of $\mathcal{C}$ (that exhibits the anomalous behaviour with respect to 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 $\mathcal{T}(\Sigma, \mathcal{V})$, $Top(s)$ is the set of minimal positions $w \in \mathcal{P}os(s)$ such that $s|_w$ is an equational redex in $\mathcal{R}$.

Formally,

$$Top(s) = \{w \in \mathcal{P}os(s) \mid s|_w \text{ is an equational redex and } \nexists w' \leq w \text{ such that } s|_{w'} \text{ is an equational redex}\}.$$

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 $(s|_w{\downarrow_{\Delta,B}})$ in $\mathcal{S}$. This allows functional assertions to be effectively checked over each equational simplification $s|_w \rightarrow^+_{\Delta,B} (s|_w{\downarrow_{\Delta,B}})$ such that $w \in Top(s)$.

Definition 5.1 (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 $\mathcal{T}(\Sigma, \mathcal{V})$ such that $Top(s) \neq \{\Lambda\}$. Let $s \rightarrow^+_{\Delta,B} s{\downarrow_{\Delta,B}}$ be an equational simplification for the state s in $\mathcal{T}(\Sigma, \mathcal{V})$. 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 5.2 (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 in $\mathcal{T}(\Sigma, \mathcal{V})$. Then,*

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

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

Example 5.3

Consider the rewrite theory $\mathcal{R}$ of Example 3.1 together with the assertional specification $\mathcal{A}$ composed of the system assertion

$\Theta =$
```
< O:Oid : C:Cid | credit : B:Int , suspended : S:Bool >
{ not(isPreferredCustomer(C:Cid)) implies B:Int >= 0 }
```

and the functional assertion

$\Phi =$
```
   updateSuspension(< U:Oid : PreferredCustomer | credit : B:Int ,
   suspended : false >) { B:Int < 0 }
→  < U:Oid : PreferredCustomer | credit : B:Int ,
   suspended : false > { true }
```

Let $\mu_{\texttt{rent}}$ be the state equational simplification issuing from

$s =$
```
< 'A1 : EconomyCar | available : true , rate : 30 >
< 'C1 : Customer | credit : - 160 , suspended : false >
updateSuspension(< 'C2 : PreferredCustomer | credit : - 120 ,
suspended : false >)
< 'RG : Register | date : 0 , rentals : 1 >
```

that ends into the canonical form

$s \downarrow_{\Delta,B} =$
```
< 'A1 : EconomyCar | available : true , rate : 30 >
< 'C1 : Customer | credit : - 160 , suspended : false >
< 'C2 : PreferredCustomer | credit : - 120 ,
suspended : true >
< 'RG : Register | date : 0 , rentals : 1 >
```

Note that $\mu_{\texttt{rent}}$ includes the following equational simplification

$\mu_{\texttt{'C2}} =$
```
updateSuspension(< 'C2 : PreferredCustomer | credit : - 120 ,
suspended : false >) →+_{Δ,B}
< 'C2 : PreferredCustomer | credit : - 120 , suspended : true >
```

for the outermost equational redex of s that is rooted at position $\texttt{3} \in \mathit{Top}(\texttt{s})$.

Then,

$$\xi_{sys}(s, \mathcal{A}) = \{2\}$$

which signals the system error symptom associated with the negative credit of non-preferred customer `'C1`.

Moreover,

$$\xi_{fun}(\mu_{\texttt{rent}}, \mathcal{A}) = \{(\mu_{\texttt{'C2}}, \{3.2.1\})\}$$

since Φ is not satisfied in $\mu_{\texttt{rent}}$ (and hence in $\mu_{\texttt{'C2}}$). The computed functional error symptom allows us to isolate the anomalous `suspended` flag value in `'C2`'s data structure, that is,

```
< 'C2 : PreferredCustomer | credit : - 120 , suspended : true >|_{3.2.1} = true.
```

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

Definition 5.4 (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 $\mathcal{C}$ (in symbols $\mathcal{C} \models \mathcal{A}$) iff*

- *for each state s in $\mathcal{C}$ that is a canonical form with respect to Δ modulo B, $s \models \mathcal{S}(\mathcal{A})$;*
- *for each state s in $\mathcal{C}$ that is not a canonical form with respect to 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 5.5 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 with respect to $\mathcal{A}$, 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 5.5 (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 $\mathcal{R}$.*

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

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

Example 5.6

Consider the rewrite theory $\mathcal{R}$ of Example 3.1 together with the assertional specification $\mathcal{A}$ and the state equational simplification $\mu_{rent} = s \rightarrow^{+}_{\Delta,B} s\downarrow_{\Delta,B}$ of Example 5.3. Recall that μ_{rent} erroneously suspends the preferred customer `'C2` through the equational simplification $\mu_{'\texttt{C2}}$ included in μ_{rent}. This error is pinpointed by the refutation of the functional assertion $\Phi \in \mathcal{A}$.

Now, by Definition 5.5,

$$check(\mu_{rent}, \mathcal{A}) = (\mu_{rent},\ \{(\mu_{'\mathtt{C2}}, \{3.2.1\})\}, fun),$$

since $\mu_{rent} \not\models \mathcal{F}(\mathcal{A})$ where $\mathcal{F}(\mathcal{A}) = \{\Phi\}$, and $\xi_{fun}(\mu_{\mathtt{rent}}, \mathcal{F}(\mathcal{A})) = \{(\mu_{'\mathtt{C2}}, \{3.2.1\})\}$.

The following proposition states that function *check* can be effectively used to dynamically check assertional specifications in given computations.

Proposition 5.7 *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and $\mathcal{C}$ be a computation in $\mathcal{R}$. Let $\mathcal{A}$ be an assertional specification for $\mathcal{R}$. Then, $\mathcal{C} \models \mathcal{A}$ iff $check(\mathcal{C}, \mathcal{A}) = (\mathcal{C}, \emptyset, none)$.*

Proof. Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and $\mathcal{C}$ be a computation in $\mathcal{R}$. Let $\mathcal{A}$ be an assertional specification for $\mathcal{R}$.

($\Longrightarrow$) We assume that $\mathcal{A}$ is satisfied in $\mathcal{C}$, that is, $\mathcal{C} \models \mathcal{A}$. Hence, by Definition 5.4, for each state s in $\mathcal{C}$ that is a canonical form with respect to Δ modulo B, $s \models \mathcal{S}(\mathcal{A})$; and for each state s in $\mathcal{C}$ that is not a canonical form with respect to Δ modulo B, $s \to^{+}_{\Delta,B} s{\downarrow_{(\Delta,B)}} \models \mathcal{F}(\mathcal{A})$. We now proceed by induction on the length of the computation $\mathcal{C}$.

Base case: $\mathcal{C} = s_0 \downarrow_{\Delta,B}$**.** In this case $\mathcal{C}$ consists of the single initial state $s_0 \downarrow_{\Delta,B}$, which is a canonical form with respect to Δ modulo B. Since $\mathcal{A}$ is satisfied in $\mathcal{C}$, we have $s_0 \downarrow_{\Delta,B} \models \mathcal{S}(\mathcal{A})$. Hence, by Definition 5.5, we trivially have

$$check(\mathcal{C}, \mathcal{A}) = check(s_0 \downarrow_{\Delta,B}, \mathcal{A}) = (s_0 \downarrow_{\Delta,B}, \emptyset, none) = (\mathcal{C}, \emptyset, none).$$

Inductive case 1: $\mathcal{C} = (s_0 \to^{+}_{\Delta,B} s_0 \downarrow_{\Delta,B} \to^{*}_{R,B} \mathcal{C}')$**.** In this case, $\mathcal{C}$ is a non-empty computation that initially simplifies the non-normalized input term s_0 by means of the equational state simplification $\mu = (s_0 \to^{+}_{\Delta,B} s_0 \downarrow_{\Delta,B})$. By inductive hypothesis, we have that

$$check(s_0 \downarrow_{\Delta,B} \to^{*}_{R,B} \mathcal{C}', \mathcal{A}) = (\varepsilon, \emptyset, none).$$

Furthermore, $\mu \models \mathcal{F}(\mathcal{A})$ since $\mathcal{A}$ is satisfied in $\mathcal{C}$ (and hence in μ). Therefore, $check(\mathcal{C}, \mathcal{A}) = check(\mu \to^{*}_{R,B} \mathcal{C}', \mathcal{A}) = (\mu \to^{*}_{R,B} \mathcal{C}', \emptyset, none) = (\mathcal{C}, \emptyset, none)$.

Inductive case 2: $\mathcal{C} = s \to_{R,B} \mathcal{C}'$**.** Since the first rewrite step in $\mathcal{C}$ is a rule application, this implies that s is already in canonical form, that is, $s = s \downarrow_{\Delta,B}$. Since $\mathcal{A}$ is satisfied in $\mathcal{C}$ we have that $s \models \mathcal{S}(\mathcal{A})$, which implies

$$s \downarrow_{\Delta,B} \models \mathcal{S}(\mathcal{A}). \tag{1}$$

Also, by inductive hypothesis, it holds that

$$check(\mathcal{C}', \mathcal{A}) = (\varepsilon, \emptyset, none). \tag{2}$$

By combining Claims 1 and 2, we get

$$check(\mathcal{C}, \mathcal{A}) = check(s \to_{R,B} \mathcal{C}', \mathcal{A}) = (s \to_{R,B} \mathcal{C}', \emptyset, none) = (\mathcal{C}, \emptyset, none).$$

($\Longleftarrow$) By contradiction, we assume that $check(\mathcal{C}, \mathcal{A}) = (\mathcal{C}, \emptyset, none)$ and $\mathcal{C} \not\models \mathcal{A}$. Since $\mathcal{A}$ is not satisfied in $\mathcal{C}$, there exists either a state s in canonical form such that $s \not\models \mathcal{S}(\mathcal{A})$ or an equational state simplification μ such that $\mu \not\models \mathcal{F}(\mathcal{A})$. Thus, by Definition 5.5, the function call $check(\mathcal{C}, \mathcal{A})$ delivers a triple $(\mathcal{C}'', Err, flag)$ with $Err \neq \emptyset$. This leads to a contradiction since we have assumed $check(\mathcal{C}, \mathcal{A}) = (\mathcal{C}, \emptyset, none)$.

■

The runtime checking methodology formalized in Definition 5.5 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*,[5] $T_{\mathcal{R}}(s_0)$. RWL computation trees are typically large and complex objects that represent the highly-concurrent, non-deterministic nature of rewrite theories.

Our methodology checks rewrite theories with respect to an assertional specification $\mathcal{A}$ at runtime by incrementally generating and checking the computation tree $T_{\mathcal{R}}(s_0)$ until a fixed depth. In fact, the complete generation of $T_{\mathcal{R}}(s_0)$ is generally not feasible since some of its branches may be infinite as they encode non-terminating computations. The general analysis algorithm, which is specified by the routine $analyze(s_0, \mathcal{R}, \mathcal{A}, depth)$, is given in Figure 3. 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 T that consists of a single root node s_0. At each expansion stage, the leaf nodes of the current T are computed by the function $frontier(T)$. 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 with respect to 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 $expand(T, s, \mathcal{M})$), thereby augmenting 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 with respect to a term slice $l^\bullet$ of the last state of $\mathcal{P}$. This term slice conveys all the relevant information

[5] 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, (Σ, Δ ∪ B, R), A, depth)
1.  T = s_0
2.  d = 0
3.  while (d ≤ depth) do
4.    F = frontier(T)
5.    for each s ∈ F
6.      for each M ∈ mS(s)
7.        (P, Err, flag) = check(M, A)
8.        case flag of
9.          none :
10.           T = expand(T, s, M)
11.         sys :
12            w = selectSysSymptom(Err)
13            l• = TSlice(last(P), Pos_w(last(P)))
14.           return backwardSlice(s_0 →*_{R∪Δ,B} P, l•)
15.         fun:
16.           (t →+_{Δ,B} t↓_{Δ,B}, L) = selectFunSymptom(Err)
17.           (t↓_{Δ,B})• = TSlice(t↓_{Δ,B}, ⋃_{w∈L} Pos_w(t↓_{Δ,B}))
18.           return backwardSlice(t →+_{Δ,B} t↓_{Δ,B}, (t↓_{Δ,B})•)
19.       end case
20.     end for
21.   end for
22.   d = d + 1
23. end while
24. return T
end
```

Figure 3: The *analyze* function.

that we automatically retrieve by using Definition 4.1 from the (system) error symptom w selected by the function $\mathit{selectSysSymptom}(\mathit{Err})$, while all other symbols in l are considered meaningless and simply pruned away. This way, the algorithm delivers a trace slice $s_0^\bullet \bullet\!\!\to^* \mathcal{P}^\bullet$ that removes from the computation all the information that does not affect the production of the chosen error symptom.

$\mathit{flag} = \mathit{fun}$**.** Some functional assertions have been violated by the considered Maude step $\mathcal{M}$. Hence, the algorithm selects a functional error symptom $(t \to^+_{\Delta,B} t{\downarrow_{\Delta,B}}, L)$ and returns the backward trace slicing of $t \to^+_{\Delta,B} t{\downarrow_{\Delta,B}}$ with respect to 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 (which is the strategy followed by the ABETS tool) by selecting the first symptom in Err. On the other hand, an 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 T is delivered that encodes the first *depth* levels of the computation tree $T_{\mathcal{R}}(s_0)$; 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: i.e., we can (manually) run *analyze* on a sequence of mutations of the original program that fix (if possible) the violations progressively encountered. The automatization of this strategy is a forthcoming work.

An example of our methodology at work is shown in Section 6.2.

Theorem 5.8 (Correctness) *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $T_{\mathcal{R}}(s_0)$ be the computation tree for initial state $s_0 \in \mathcal{T}(\Sigma, \mathcal{V}ar)$ in $\mathcal{R}$, and $\mathcal{A}$ be an assertional specification for $\mathcal{R}$. Let depth be a natural number. Then, analyze$(s_0, \mathcal{R}, \mathcal{A}, depth)$ terminates and*

1. *if there exists a computation $\mathcal{C}$ in $T_{\mathcal{R}}(s_0)$ such that $\mathcal{C} \not\models \mathcal{A}$ and $length(\mathcal{C}) \leq depth$, then analyze$(s_0, \mathcal{R}, \mathcal{A}, depth)$ delivers a backward trace slice $\mathcal{C}^{\bullet}_{pre}$ of a fragment $\mathcal{C}_{pre}$ of $\mathcal{C}$ that violates either a functional or a system assertion in $\mathcal{A}$. $\mathcal{C}^{\bullet}_{pre}$ is computed with respect to the term slice of the last state of $\mathcal{C}_{pre}$ that includes all subterms correlated to a chosen error symptom;*

2. *otherwise, analyze$(s_0, \mathcal{R}, \mathcal{A}, depth)$ delivers a tree T that corresponds to the expansion of the first depth levels of $T_{\mathcal{R}}(s_0)$.*

Proof. Termination of *analyze*$(s_0, \mathcal{R}, \mathcal{A}, depth)$ is trivial since the main **while**-loop is performed at most *depth* times, and at each iteration it invokes: 1) the terminating backward slicing algorithm of [13], and 2) the function *check*, which is terminating because the execution in $\mathcal{R}$ of the assertional specification is also terminating.

Now, let us prove Claim 1. Function *analyze* implements a breadth-first visit of the Maude steps in $T_{\mathcal{R}}(s_0)$ until an assertion violation occurs or the *depth* bound has been reached. Since we assume that there exists $\mathcal{C}$ in $T_{\mathcal{R}}(s_0)$ such that $\mathcal{C} \not\models \mathcal{A}$ and $length(\mathcal{C}) \leq depth$, there exists a (minimum) prefix $\mathcal{C}_{pre}$ of $\mathcal{C}$ such that either $\mathcal{C}_{pre} \not\models \mathcal{S}(\mathcal{A})$ or $\mathcal{C}_{pre} \not\models \mathcal{F}(\mathcal{A})$ that is detected by *analyze*. Let us assume that $\mathcal{C}_{pre} \not\models \mathcal{S}(\mathcal{A})$ (the proof of the case $\mathcal{C}_{pre} \not\models \mathcal{F}(\mathcal{A})$ follows an analogous argument). Hence, $\mathcal{C}_{pre} = s_0 \rightarrow^{*}_{\Delta \cup R, B} \mathcal{P}$ where $\mathcal{P}$ is obtained by checking the last expanded Maude step $\mathcal{M}$, that is, $check(\mathcal{M}, \mathcal{A}) = (\mathcal{P}, Err, sys)$. Let l be the last state in $\mathcal{C}_{pre}$. Now, the state l is a canonical form such that $l \not\models \Theta$ for some $\Theta \in \mathcal{S}(\mathcal{A})$. Therefore, $\xi_{sys}(l, \Theta) \subseteq Err$. Let $w \in \xi_{sys}(l, \Theta) \subseteq Err$ be a selected system error symptom. By Definition 4.1, $l^{\bullet} = TSlice(l, \mathcal{P}os_w(l))$ computes a term slice $l^{\bullet}$ that includes all of the symbols in the subterm $l|_w$. Thus,

$$backwardSlice(\mathcal{C}_{pre}, l^{\bullet}) = backwardSlice(s_0 \rightarrow^{*}_{R \cup \Delta, B} \mathcal{P}, l^{\bullet})$$

is a backward trace slice of $\mathcal{C}_{pre}$ that is computed with respect to a state l that includes the subterm $l|_w$ that is univocally correlated to the chosen system error symptom w.

Note that, in the case when there is no computation $\mathcal{C}$ in $T_{\mathcal{R}}(s_0)$ such that $\mathcal{C} \not\models \mathcal{A}$ and $length(\mathcal{C}) \leq depth$, Claim 2 is trivially proved by construction of the *analyze* function. In this case, there is no assertion violation, and thus the algorithm generates a tree T by unraveling all of the Maude steps of $T_{\mathcal{R}}(s_0)$ until the bound *depth* is reached. ∎

The assertion-based trace slicing methodology described in this section is a synchronous procedure that incrementally executes, checks, and possibly slices Maude computations at runtime. However, note that an offline, asynchronous procedure (that works on pre-calculated computations) can be easily derived from our synchronous algorithm with little effort. Actually, it suffices to provide the whole computation $\mathcal{C}$ to be analyzed as input and to stepwisely check its Maude steps by using the *check* function in search of assertion failures. When an assertion violation is detected on a prefix $\mathcal{C}_{pre}$ of the input computation that reaches the erroneous state e, a slicing criterion is then inferred by exploiting the error symptoms that are associated with the violation, as happened in the synchronous case; finally, a backward trace slice of $\mathcal{C}_{pre}$ is computed with respect to the considered slicing criterion. Synchronous and asynchronous modalities have been implemented in a prototypical trace analysis system that we describe in Section 6.

In the following subsection, we improve our basic methodology by providing a practical strategy that delivers finer slicing criteria by reducing the number of positions that are worth tracking to those that appear in selected subformulas of the (post)conditions.

5.3. Improving the inference of the slicing criteria

Without loss of generality, we assume that any logic formula ϕ in a system assertion $S\{\varphi\}$ or in a functional assertion $I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$ is written in conjunctive normal form $\varphi_1 \wedge \ldots \wedge \varphi_n$, where $\wedge$ does not occur in any φ_i, $i = 1, \ldots, n$. Then, a refined strategy for inferring accurate slicing criteria for an erroneous state e can be formulated as follows:

1. When a system assertion $S\{\varphi\}$ with $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_n$ is refuted, this is because e matches S (modulo the enriched equational theory) with matching substitution σ, while $\varphi\sigma$ is not satisfied. Hence, we sequentially examine the conjuncts φ_i, $i = 1, \ldots, n$, and for the first failing conjunct φ_j, a slicing criterion is synthesized by instantiating the pattern S with $\sigma_{\lceil \mathcal{V}ar(\varphi_j)}$, where $\sigma_{\lceil \mathcal{V}ar(\varphi_j)}$ is the restriction of σ to the variables that occur in φ_j.

2. In the case when a functional assertion $I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$ is violated, an instance $I\sigma_{in}$ of the input template I reduces to a canonical form O_e in e, and one of the following two cases occurs:
   (a) O_e does not equationally match the associated instance $O(\sigma_{in}\sigma_{out})$ of the output template O; in this case, the slicing criterion is computed by applying the modular order-sorted least general generalization algorithm of [13] to gather up all of the mismatches (modulo the considered equational theory) between the erroneous canonical form O_e and $O(\sigma_{in}\sigma_{out})$ (see [20] for full details).
   (b) O_e does equationally match $O(\sigma_{in}\sigma_{out})$, but all of the corresponding matchers falsify the formula φ_{out}; in this case, the methodology proceeds analogously to case 1 (system assertions) and synthesizes the slicing criterion by examining the failing conjuncts of φ_{out} systematically.

In the following section, we describe the assertion-based, dynamic analyzer ABETS that implements the slicing methodology for (Full) Maude theories proposed in this article.

6. Implementation

Our assertion-based, dynamic trace analysis methodology has been implemented in the prototype tool ABETS (*Assertion-BasEd Trace Slicer*), which is publicly available at `http://safe-tools.dsic.upv.es/abets`. For implementing the exploration capabilities of ABETS, we reused (part of) the inspection machinery of the dynamic exploration framework ANIMA [49] that was developed in previous work [15]. Likewise, the slicing-based analysis capabilities of ABETS (and ANIMA) are rooted in the trace (and program) slicing procedures developed in [13, 48], which were first implemented in the dynamic slicing tool *i*Julienne [50]. One of the main novelties of ABETS with respect to previous (ANIMA and *i*Julienne) systems is that it has been implemented to run at the Core Maude and Full Maude levels, while the need to invoke Full Maude is automatically inferred so that high-performance analyses can be achieved for theories that do not require Full Maude capabilities.

The architecture of ABETS is depicted in Figure 4 and consists of the following components: (i) a Maude-based slicer and constraint-checker core that consists of about 400 Maude function definitions (approximately 3500 lines of source code) that can run at both Core Maude and Full Maude levels interchangeably; (ii) a scalable, high-performance NoSQL database powered by MongoDB that endows the tool with *memoization* capabilities in order to improve the response time for complex and recurrent executions; (iii) a RESTful Web service written in Java that is executed by means of the Jersey JAX-RS API; and (iv) an intuitive user interface that is based on AJAX technology and written in HTML5 canvas and Javascript.

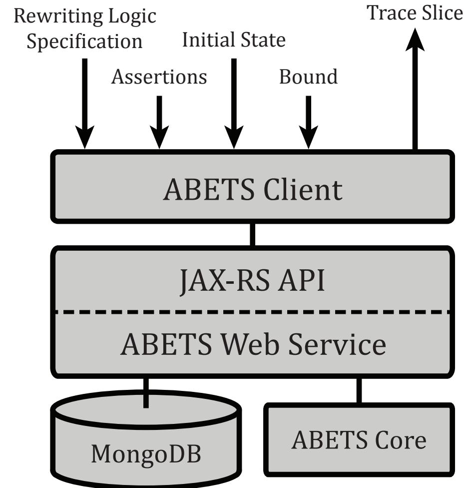

Figure 4: ABETS architecture.

Given a rewrite theory $\mathcal{R}$, runtime assertion checking is performed in ABETS by first wrapping $\mathcal{R}$, via inclusion, in a system module PRED($\mathcal{R}$) that also contains the extra predicates the user may need to define new formulas. Functional and system assertions are given sorts in the functional module ASSERTION

```
fmod ASSERTION is
  sorts sAssertion fAssertion Assertion .
  subsorts sAssertion fAssertion < Assertion .

  op _/\_ : Bool Bool -> Bool [ ctor assoc prec 125 gather (e e) ] .
  op _‘{_‘} : Universal Bool -> sAssertion [ ctor poly (1) ] .
  op _‘{_‘}->_‘{_‘} : Universal Bool Universal Bool -> fAssertion
                                                    [ ctor poly (1 3) ] .
endfm
```

which is also included in PRED($\mathcal{R}$). This hierarchical setup allows assertions specifications in PRED($\mathcal{R}$) to be directly parsed by means of Maude's `metaParse` command, resulting in a list $\mathcal{A}$ of (system and functional) assertions.

Given a computation $\mathcal{C}$ in $\mathcal{R}$, for asynchronous checking ABETS proceeds by incrementally consuming Maude steps (of $\mathcal{C}$) while checking the assertions in $\mathcal{A}$ that are relevant to the step. In contrast, for the synchronous case, we distinguish two operation modes, depending on how computations are dynamically generated. The first mode consists of a step-by-step generation of a computation $\mathcal{C}$ that follows Maude's internal strategy. Each time a Maude step is generated, the satisfaction of $\mathcal{A}$ is checked, which is similar to the asynchronous case. The second mode allows a fragment of the whole computation tree to be deployed up to a given depth d that is measured in Maude steps. Also in this case, the satisfaction of $\mathcal{A}$ is checked at each Maude step.

In the event that an assertion is falsified at state s_n, the dynamic checking is immediately stopped and ABETS delivers a (simplified) counter-example trace. As explained in Section 4, for system assertions the simplified trace is computed as the backward slicing of the trace from the initial state s_0 to s_n (with respect to a slicing criterion that is automatically inferred by matching the discordant subterm of s_n with the state pattern of the falsified assertion). As for functional assertions, the delivered counter-examples consist of the equational simplification trace for the outermost term that is responsible for the falsification. In this case, the slicing criteria is automatically obtained as the discrepancy between the normal form that is expected and the normal form that is actually computed. This discrepancy is calculated by using the least general generalization algorithm of [18] that was first implemented in ACUOS [51], which has been coupled into the ABETS core.

The asynchronous mode is preferable for the debugging of previously-identified faulty executions. Since this mode avoids having to reexecute the program, it is the lightest of the three checking modes. While the synchronous tree-checking mode is useful for analyzing all the non-deterministic paths of a non-confluent program execution at the same time, the synchronous trace-checking mode is the best to debug deterministic executions or (arbitrarily chosen) non-deterministic computations of non-confluent programs. An upper bound is necessary to finitize the analyses in the synchronous checking modes, which is typically based on counting the elapsed time or the number of rewrite steps. For the synchronous trace-checking mode, we have set 250 rewrite steps as the upper bound, mainly because trace formatting is rather time-consuming (the instrumentation of a trace with 250 rewrite steps can result in thousands of instrumented rewrite steps that also need to be properly formatted to be output). Nevertheless, offline (console) checking can deal with much higher bounds, especially when formatting is dispensed. As for the synchronous tree-checking mode, we have set two different bounds: the first one limits the depth of the deployed computation trees to 10, while the second one limits the number of nodes that can be checked to 100K.

Finally, as we mentioned in Section 2, object-oriented modules are just syntactic sugar in Maude and are internally transformed into system modules for execution purposes. In object notation, object attributes do not need to be explicitly written in the rules when they remain unchanged, which overcomes the classical annoyance of expressing invariance or *frame properties* in algebraic specifications (i.e., that those parts of a state that are not affected by a change remain unchanged). However, these attributes do appear in (desugared) program states and computation traces. In order to simplify object trace slices to the fullest and to deal effectively with frame properties by mimicking attribute hiding, ABETS is endowed with refined matching and filtering procedures that are automatically activated when dealing with object modules and are transparent to the user.

6.1. ABETS *Features*

These are the main features provided by ABETS:

1. *Constraint checking.* ABETS implements the analysis technique in this paper in both the asynchronous modality and the two synchronous modes, previously described. Whenever an assertion fails to be satisfied, in all cases, the user is given an automatically generated counterexample trace slice that he/she can fully inspect, query, and slice further.
2. *Automatic slicing.* The tool is endowed with a (forward and backward) incremental, interactive trace slicer, which allows the user to greatly simplify any execution trace that falsifies at least one of the constraints. In order to incrementally unmask the bugs that are responsible for the errors, at each run, the slicing criterion is automatically inferred with respect to the first non-satisfied constraint.
3. *Interactive navigation.* Computations and computation slices can be easily and thoroughly inspected by navigating the traces and by accessing all of their available information, which includes the details of the instrumentation of each Maude step. Specifically, for each instrumented Maude step $\mathcal{M}$, ABETS shows the rule and (possible) equations applied in $\mathcal{M}$ together with their computed matching substitutions, *redexes*, and *contractums.*
   All this information is accessible in both source and meta-level representations. Moreover, for conditional rewrite steps, an in-depth analysis of the condition proofs can be accessed through the *inspect condition* option of the context menu.
4. *Program slicing.* In addition to the simplification achieved by slicing execution traces, ABETS offers the user the possibility to compute a dynamic *program slice* that only contains the potentially faulty rules or equations [13]. This feature is particularly useful in the case when a functional assertion fails, since the relevant equations are isolated from a presumably large, complex program that consists of many modules.

6.2. ABETS *in action*

Maude programs can be uploaded in ABETS as simple `.maude` or `.fm` files. Some predefined specifications are provided with the tool for demonstration purposes, including the car rental system of Example 2.1, and an assertional specification $\mathcal{A}_{\texttt{rent}}$ that contains the system assertion Θ of Example 3.4 and the functional assertion Φ of Example 3.7. Let us consider the synchronous checking modality that non-deterministically expands all Maude steps that originate from the initial state:

```
s0 = < 'A1 : EconomyCar | available : true , rate : 30 >
     < 'A3 : MidSizeCar | available : true , rate : 45 >
     < 'A5 : FullSizeCar | available : true , rate : 70 >
     < 'C1 : Customer | credit : 50 , suspended : false >
     < 'C2 : PreferredCustomer | credit : 100 , suspended : false >
     < 'RG : Register | date : 0 , rentals : 0 >
```

This is achieved in ABETS by calling *analyze*$(\mathcal{R}_{\texttt{rent}}, \mathcal{A}_{\texttt{rent}}, 1)$, where $\mathcal{R}_{\texttt{rent}}$ is the rewrite theory specified by the `RENT-A-CAR-ONLINE-STORE` object module. The screenshots shown in

PROVIDE THE MAUDE INPUT PROGRAM AND INPUT STATE OR TRACE ?

Rent-a-car

Upload

```
1  (omod RENT-A-CAR-ONLINE-STORE is
2    pr CONVERSION .
3    pr QID .
4
5    subsort Qid < Oid .
6
7    class Register | rentals : Nat , date : Nat .
8    class Customer | credit : Int, suspended : Bool .
9    class Car | available : Bool, rate : Nat .
10   class Rental | deposit : Nat, dueDate : Nat, pickUpDate : Nat, customer : Oid, ca
11   class PreferredCustomer .
12   subclass PreferredCustomer < Customer .
13
14   class EconomyCar .
15   class MidSizeCar .
16   class FullSizeCar .
17   subclasses EconomyCar MidSizeCar FullSizeCar < Car .
18
19   vars U C R RG : Oid .
20   vars CREDIT AMNT : Int .
21   vars TODAY PDATE DDATE RATE DPST RNTLS : Nat .
22
23   rl [new-day] : < RG : Register | date : TODAY >
24               => < RG : Register | date : TODAY + 1 > .
25
26   crl [3-day-rental] :
```

Synchronous checking

Generate

```
< 'A1 : EconomyCar | available : true , rate : 30 > < 'A3 : MidSizeCar | available :
true , rate : 45 > < 'A5 : FullSizeCar | available : true , rate : 70 > < 'C1 : Cust
omer | credit : 50 , suspended : false > < 'C2 : PreferredCustomer | credit : 100 ,
suspended : false > < 'RG : Register | rentals : 0 , date : 0 >
```

Figure 5: Input Phase I.

PROVIDE THE EXTRA PREDICATES AND SET OF ASSERTIONS TO CHECK ?

Add the extra predicates used in your assertions:

```
(mod RENT-A-CAR-ONLINE-STORE-PRED is
   inc RENT-A-CAR-ONLINE-STORE .
   sorts sAssertion fAssertion Assertion .
   subsorts sAssertion fAssertion < Assertion .
   op _/\_ : Bool Bool -> Bool [ ctor assoc prec 125 gather (e e) ] .
   op _`{_`} : Universal Bool -> sAssertion [ ctor poly (1) ] .
   op _`{_`}->_`{_`} : Universal Bool Universal Bool -> fAssertion [ ctor poly (1 3) ] .

    op isPreferredCustomer : Cid -> Bool .
      eq isPreferredCustomer(PreferredCustomer) = true .
      eq isPreferredCustomer(U:Cid) = false [owise] .

endm)
```

Based on your program and predicates, specify your assertions:

```
< O:Oid : C:Cid | credit : B:Int , suspended : S:Bool > { not(isPreferredCustome
r(C:Cid)) implies B:Int >= 0 }
updateSuspension(< U:Oid : PreferredCustomer | credit : B:Int , suspended : false >)
{ B:Int < 0 } -> < U:Oid : PreferredCustomer | credit : B:Int , suspended : false >
{ true }
```

Starting from the provided input state, check your assertions in the dynamically computed:

execution tree, up to the following tree depth (in Maude steps): 10

Check

Figure 6: Input Phase II.

Figure 5 and Figure 6 illustrate the initial, input phase for the parameters $\mathcal{R}_{\texttt{rent}}$ and $\mathcal{A}_{\texttt{rent}}$, respectively.

By pressing the CHECK button, the assertion checking algorithm starts and immediately

discovers that Θ is not satisfied in the following Maude step $\mathcal{M}$

$$\mathtt{s}_0 \xrightarrow{\texttt{3-day-rental}}$$

```
< 'A1 : EconomyCar | available : false , rate : 30 >
< 'A3 : MidSizeCar | available : true , rate : 45 >
< 'A5 : FullSizeCar | available : true , rate : 70 >
< 'C1 : Customer | credit : - 40 , suspended : false >
< 'C2 : PreferredCustomer | credit : 100 , suspended : false >
< 'R0 : Rental | car : A1 , customer : 'C1 , deposit : 90 ,
dueDate : 3 , pickUpDate : 0 , rate : 30 >
< 'RG : Register | date : 0 , rentals : 1 >
```

since `'C1`'s credit becomes negative after the application of the `3-day-rental` rule in $\mathtt{s}_0$ that allows `'C1` to rent car `'A1`.

Then, a system error symptom is automatically computed by the tool, which unambiguously signals the anomalous subterm

```
< 'C1 : Customer | credit : - 40 , suspended : false >
```

of the last state of $\mathcal{M}$, and produces the associated term slice

$$l^\bullet = \bullet_1\ \bullet_2\ \bullet_3\ \texttt{<}\ \bullet_4\ \texttt{:}\ \bullet_5\ \texttt{| credit : - 40 ,}\ \bullet_6\ \texttt{>}\ \bullet_7\ \bullet_8\ \bullet_9$$

Finally, the algorithm automatically generates the backward trace slice of $\mathcal{M}$ with respect to $l^\bullet$, that is,

$$\texttt{<}\ \bullet_{10}\ \texttt{:}\ \bullet_{11}\ \texttt{| available : true , rate : 30 >}\ \bullet_{15}\ \bullet_{14}\ \texttt{<}\ \bullet_4\ \texttt{:}\ \bullet_5\ \texttt{| credit : 50 ,}$$
$$\texttt{suspended : false >}\ \bullet_{13}\ \bullet_{12}$$
$$\bullet\!\!\to$$
$$\bullet_1\ \bullet_2\ \bullet_3\ \texttt{<}\ \bullet_4\ \texttt{:}\ \bullet_5\ \texttt{| credit : - 40 ,}\ \bullet_6\ \texttt{>}\ \bullet_7\ \bullet_8\ \bullet_9$$

which suggests an erroneous implementation of the `3-day-rental` rule. Indeed, `3-day-rental` authorizes any car rental to all customers, even when the requested deposit exceeds the residual customer credit, which contradicts the property asserted by the system constraint Θ.

If we re-execute the analysis after correcting the buggy `3-day-rental` rule in the `RENT-A-CAR-ONLINE-STORE` module, we can also discover a violation of the functional assertion Φ that detects an anomalous behaviour of function `updateSuspension`: in fact, `updateSuspension` suspends *every* customer with debts (i.e., a negative credit), while preferred customers should never be suspended. Details of the refuted assertion are attained by selecting the *falsified assertion* option from the tool menu, as shown in Figure 7. The delivered trace slice is shown in Figure 8, which displays a tabular view of the trace (also provided by our tool), where the achieved reduction is shown (97%). In order to simplify the displayed view of the trace, we note that subindices of $\bullet$-variables are hidden in our implementation; they can be shown by selecting the *detailed trace* view. After the diagnosis, runtime checks can be turned off to avoid any execution overheads.

Finally, by running the *program slice* option of ABETS, all program statements that can (potentially) cause the erroneous program behavior are automatically identified (see Figure 9).

Falsified assertion information ×

Assertion

```
updateSuspension(< U : PreferredCustomer | credit : B , suspended :
false >) { B < 0 } -> < U : PreferredCustomer | credit : B , suspen
ded : false > { true }
```

Type

```
Functional
```

Input Substitution (as obtained from input/pre)

```
B / 90 + 10 - 126
U / 'C2
```

Input Substitution (normalized, as applied to output/post)

```
B / - 26
U / 'C2
```

Figure 7: Description of the falsified assertion Φ.

6.3. Experimental evaluation

To evaluate the performance of the ABETS system, we benchmarked the system on the following collection of (Core and Full) Maude programs, which are all available within the ABETS Web platform (each program has been coupled with a suitable assertional specification, which is also available at the system's website):

- *Bank model*, a conditional Maude specification that models a faulty, distributed banking system.
- *Blocks World*, the typical AI planning problem, which consists of producing one or more vertical stacks of blocks (placed on a table) that can be moved by means of a robot arm.
- *BRP*, the Bounded Retransmission Protocol (BRP) [52], which is a data link protocol developed and used by Philips Electronics that can be thought of as a variant of the alternating bit protocol.
- *Dekker*, a Maude specification that models a faulty version of Dekker's algorithm, one of the earliest solutions to the mutual exclusion problem which appeared in [53].
- *Maze*, a non-deterministic Maude specification that defines a maze game in which multiple players must reach a given exit point by walking or jumping, where colliding players are eliminated from the game [15].

Trace information ×

<table>
<tr><th>State</th><th>Label</th><th>Trace</th><th>Trace Slice</th></tr>
<tr><td>1</td><td>'Start</td><td>< 'A1 : EconomyCar | available: true , rate: 30 > < 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > < 'C2 : PreferredCustomer | credit: 100 , suspended: false > < 'RG : Register | rentals: 0 , date: 0 ></td><td>•</td></tr>
<tr><td>4</td><td>3-day-rental</td><td>< 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > < 'C2 : PreferredCustomer | credit: 100 - 90 , suspended: false , none > < 'A1 : EconomyCar | available: false , rate: 30 , none > < 'RG : Register | rentals: 0 + 1 , date: 0 , none > < qid("R" + string(0,10)) : Rental | pickUpDate: 0 , dueDate: 0 + 3 , car: 'A1 , deposit: 90 , customer: 'C2 , rate: 30 ></td><td>•</td></tr>
<tr><td>19</td><td>new-day</td><td>< 'A1 : EconomyCar | available: false , rate: 30 > < 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > < 'C2 : PreferredCustomer | credit: 10 , suspended: false > < 'R0 : Rental | car: 'A1 , customer: 'C2 , deposit: 90 , dueDate: 3 , pickUpDate: 0 , rate: 30 > < 'RG : Register | date: 0 + 1 , rentals: 1 , none ></td><td>•</td></tr>
<tr><td>23</td><td>new-day</td><td>< 'A1 : EconomyCar | available: false , rate: 30 > < 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > < 'C2 : PreferredCustomer | credit: 10 , suspended: false > < 'R0 : Rental | car: 'A1 , customer: 'C2 , deposit: 90 , dueDate: 3 , pickUpDate: 0 , rate: 30 > < 'RG : Register | date: 1 + 1 , rentals: 1 , none ></td><td>•</td></tr>
<tr><td>27</td><td>new-day</td><td>< 'A1 : EconomyCar | available: false , rate: 30 > < 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > < 'C2 : PreferredCustomer | credit: 10 , suspended: false > < 'R0 : Rental | car: 'A1 , customer: 'C2 , deposit: 90 , dueDate: 3 , pickUpDate: 0 , rate: 30 > < 'RG : Register | date: 2 + 1 , rentals: 1 , none ></td><td>•</td></tr>
<tr><td>33</td><td>new-day</td><td>< 'A1 : EconomyCar | available: false , rate: 30 > < 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > < 'C2 : PreferredCustomer | credit: 10 , suspended: false > < 'R0 : Rental | car: 'A1 , customer: 'C2 , deposit: 90 , dueDate: 3 , pickUpDate: 0 , rate: 30 > < 'RG : Register | date: 3 + 1 , rentals: 1 , none ></td><td>•</td></tr>
<tr><td>39</td><td>late-return</td><td>< 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > updateSuspension(< 'C2 : PreferredCustomer | credit: 10 - 126 + 90 , suspended: false , none >) < 'A1 : EconomyCar | available: true , rate: 30 , none > < 'RG : Register | date: 4 , rentals: 1 , none ></td><td>• updateSuspension(< • : • | credit: 10 - 126 + 90 , suspended: false , • >) •</td></tr>
<tr><td>44</td><td>suspend</td><td>< 'A3 : MidSizeCar | available: true , rate: 45 > < 'A5 : FullSizeCar | available: true , rate: 70 > < 'C1 : Customer | credit: 50 , suspended: false > < 'C2 : PreferredCustomer | credit: - 26 , none , suspended: true > < 'A1 : EconomyCar | available: true , rate: 30 , none > < 'RG : Register | date: 4 , rentals: 1 , none ></td><td>• < • : • | • , • , suspended: true > •</td></tr>
<tr><td colspan="2">Total size:</td><td>2410</td><td>76</td></tr>
<tr><td colspan="4">Reduction Rate: 97%</td></tr>
</table>

Figure 8: Compact view of the computed Trace Slice after refuting the functional assertion Φ.

- *Philosophers*, the classical Dijkstra dining philosophers concurrency example that deals with resource access synchronization.
- *Rent-a-car (fm)*, the leading example of this article, a *Full Maude* object-oriented system that models the logic of the faulty, distributed, object-oriented, online car-rental store of Example 2.1.

Program slice ×

```
crl [late-return] :
 < U : Customer | credit : CREDIT >
 < C : Car | rate : RATE >
 < R : Rental | customer : U, car : C, pickUpDate : PDATE,
  dueDate : DDATE, deposit : DPST >
 < RG : Register | date : TODAY >
 =>
 updateSuspension(< U : Customer | credit : (CREDIT - AMNT) + DPST >)
 < C : Car | available : true >
 < RG : Register | >
 if DDATE < TODAY /\ AMNT := RATE * (DDATE - PDATE) +
   (120 * RATE * (TODAY - DDATE)) quo 100 .

op updateSuspension : Object -> Object .
 ceq [suspend] : updateSuspension(
   < U : Customer | credit : CREDIT , suspended : false >) =
   < U : Customer | credit : CREDIT , suspended : true >
  if (CREDIT < 0) .

  eq [maintainSuspension] : updateSuspension(
   < U : Customer | suspended : B:Bool >) =
   < U : Customer | suspended : B:Bool > [owise] .
endom)
```

Figure 9: Computed Program Slice.

- *Stock Exchange*, a rewrite theory that specifies a simplified stock exchange concurrent system in which traders operate on stocks via limit orders that are used to set a trading threshold (price limit).
- *Stock Exchange (fm)*, a *Full Maude*, object-oriented version of the *Stock Exchange* example.
- *Webmail*, a Maude specification borrowed from [54] that models a webmail application that provides typical login/logout functionality, email management, system administration capabilities, etc.

In this section, we focus on evaluating the performance of the assertion-checking capabilities of ABETS, since the empirical evaluation of the underlying trace slicing process has been already described in [13]. Table 1 summarizes the preliminary results that we achieved.

Experiments have been conducted on a 3.3GHz Intel Xeon E5-1660 with 64GB ROM by applying the following *modus operandi*.

1. For each program, by using the `metaRewrite` command we force (Full or Core) Maude to generate four execution traces of increasing length ($k = 10, 50, 100, 500$ rewrite steps)

using its internal, default rewrite strategy, and we record the corresponding computation times.

2. For each execution trace, we record the number of assertion checks performed by the ABETS assertion-checking engine when synchronously checking the assertional specification of the corresponding program (the column `checks` in Table 1).
3. We compute the average `slowdown` (in seconds) of five independent measurements of the execution time (in seconds) required for Point 2 with respect to Maude's computation times of Point 1.

	Trace length $k = 10$		Trace length $k = 50$		Trace length $k = 100$		Trace length $k = 500$	
	checks	slowdown (in s)	checks	slowdown (in s)	checks	slowdown (in s)	checks	slowdown (in s)
Bank model	44	0.004	204	0.018	404	0.03	2004	0.098
Blocks World	19	0.001	59	0.002	109	0.004	509	0.041
BRP	22	0.001	102	0.003	202	0.006	1002	0.023
Dekker	22	0.002	102	0.01	202	0.021	1002	0.1
Maze	88	0.003	687	0.034	1437	0.073	7437	0.384
Philosophers	22	0.001	102	0.004	202	0.008	811	0.035
Rent-a-car *(fm)*	33	0.004	153	0.011	303	0.024	1503	0.11
Stock Ex.	33	0.002	153	0.01	303	0.019	1503	0.093
Stock Ex. *(fm)*	44	0.004	204	0.027	404	0.071	2004	1.248
Webmail App	22	0.006	102	0.027	202	0.056	1002	0.275

Table 1: ABETS synchronous assertion-checking performance analysis.

Our experimental results indicate that the overhead due to assertion checking is reasonably low. Actually, our figures reveal very small slowdowns (0.1 ms/checking on average). However, the incurred overhead obviously depends on the number of assertions that are contained in the specification and, specially, the degree of instantiation of their associated patterns: patterns that are too general can result in a large number of (often) unprofitable evaluations of the formula since the number of possible B-matchings with the system's states can grow very quickly. This is evident in the `Maze` example of Table 1, where one of the assertions contains a very general pattern that matches a term that commonly appears in the trace, thus reaching up to 7,437 evaluations of the corresponding formula. Of course, the more instantiated the assertions, the better the performance.

7. Conclusions and Further Work

Checking logical assertions is a popular approach to error discovery. We have formalized a framework that integrates dynamic slicing and runtime assertion checking to help diagnose programming errors in rewriting logic theories. Our methodology smoothly blends in with the general framework for the analysis and exploration of rewriting logic computations that we developed in previous research [15]. The main improvement obtained is that no error symptom must be separately identified because the assertions (or more precisely, their runtime checks) are used to synthesize deft slicing criteria. In other words, false assertions not only flag error symptoms, but, more importantly, they are used as the starting point for automated backward slicing.

The proposed methodology has been implemented in the ABETS prototype tool, which provides a skillful and highly dynamic environment for the runtime assertion-checking of RWL theories. Our preliminary experience has shown that the synergistic capabilities of ABETS can provide a very powerful Swiss Army knife in error diagnosis and debugging by abetting the analyst's attention to suspicious (but otherwise possibly overlooked) aspects of the code.

The techniques we have developed are adequately fast and usable, with a performance that is comparable to Maude itself when applied to programs of several hundred lines, yet there are certainly several ways that our prototype implementation can be improved. For instance, two issues of interest would be to refine the inferred slicing criteria by enhancing the processing of postconditions to reduce the number of variables that are worth observing and also to add further flexibility to the selection of the violated assertion(s) to consider. Actually, our methodology can be straightforwardly adapted to infer all slicing criteria for all failing assertions; however, from our own experience, we find it is overwhelming for the user to receive all (alternative) criteria together at once. Instead, we are thinking of a kind of 'best fit' notion that allows us to prioritize the criterion(a) that will most likely lead to fixing a given error with less effort.

References

[1] L. Clarke, D. Rosenblum, A Historical Perspective on Runtime Assertion Checking in Software Development, ACM SIGSOFT Software Engineering Notes 31 (3) (2006) 25–37.

[2] B. Korel, J. Laski, Dynamic Program Slicing, Information Processing Letters 29 (3) (1988) 155–163.

[3] B. Korel, J. Rilling, Application of Dynamic Slicing in Program Debugging, in: Proceedings of the 3rd International Workshop on Automated Debugging (AADEBUG 1997), Linköping University Electronic Press, 43–58, 1997.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude: A High-Performance Logical Framework, Springer-Verlag, ISBN 978-35-4071-940-3, 2007.

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

[6] J. Meseguer, Membership Algebra as a Logical Framework for Equational Specification, in: Proceedings of the 12th International Workshop on Algebraic Development Techniques (WADT 1997), vol. 1376 of *Lecture Notes in Computer Science*, Springer-Verlag, 18–61, 1998.

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

[8] M. Alba-Castro, M. Alpuente, S. Escobar, Abstract Certification of Global Non-Interference in Rewriting Logic, in: Proceedings of the 8th International Symposium on Formal Methods for Components and Objects (FMCO 2009), vol. 6286 of *Lecture Notes in Computer Science*, Springer-Verlag, 105–124, 2010.

[9] M. Alpuente, D. Ballis, D. Romero, Specification and Verification of Web Applications in Rewriting Logic, in: Proceedings of the 16th International Symposium on Formal Methods (FM 2009), vol. 5850 of *Lecture Notes in Computer Science*, Springer-Verlag, 790–805, 2009.

[10] M. Alpuente, D. Ballis, J. Espert, D. Romero, Model-checking Web Applications with Web-TLR, in: Proceedings of the 8th International Symposium on Automated Technology for Verification and Analysis (ATVA 2010), vol. 6252 of *Lecture Notes in Computer Science*, Springer-Verlag, 341–346, 2010.

[11] M. Alpuente, D. Ballis, M. Baggi, M. Falaschi, A Fold/Unfold Transformation Framework for Rewrite Theories extended to CCT, in: Proceedings of the 19th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2010), Association for Computing Machinery, 43–52, 2010.

[12] N. Martí-Oliet, M. Palomino, A. Verdejo, Rewriting Logic Bibliography by Topic: 1990–2011, The Journal of Logic and Algebraic Programming 81 (7–8) (2012) 782–815.

[13] M. Alpuente, D. Ballis, F. Frechina, D. Romero, Using Conditional Trace Slicing for improving Maude Programs, Science of Computer Programming 80, Part B (2014) 385 – 415.

[14] M. Alpuente, D. Ballis, F. Frechina, J. Sapiña, Inspecting Rewriting Logic Computations (in a Parametric and Stepwise Way), in: Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi (SAS 2014), vol. 8373 of *Lecture Notes in Computer Science*, Springer-Verlag, 229–255, 2014.

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

[16] G. T. Leavens, Y. Cheon, Design by Contract with JML, Available at: `http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf`, 2005.

[17] B. Meyer, Object-Oriented Software Construction, 2nd Edition, Prentice-Hall, ISBN 978-01-3629-155-8, 1997.

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

[19] ABETS, The ABETS Web site, Available at: `http://safe-tools.dsic.upv.es/abets`, 2015.

[20] M. Alpuente, D. Ballis, F. Frechina, J. Sapiña, Combining Runtime Checking and Slicing to improve Maude Error Diagnosis, in: Logic, Rewriting, and Concurrency - Festschrift Symposium in Honor of José Meseguer, vol. 9200 of *Lecture Notes in Computer Science*, Springer-Verlag, 72–96, 2015.

[21] C. C. Din, O. Owe, R. Bubel, Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems, in: Proceedings of the 2nd International Conference

on Model-Driven Engineering and Software Development (MODELSWARD 2014), IEEE Computer Society Press, 480–487, 2014.

[22] M. Leucker, Teaching Runtime Verification, in: Proceedings of the 2nd International Conference on Runtime Verification (RV 2011), vol. 7186 of *Lecture Notes in Computer Science*, Springer-Verlag, 34–48, 2012.

[23] M. Leucker, C. Schallhart, A Brief Account of Runtime Verification, The Journal of Logic and Algebraic Programming 78 (5) (2009) 293–303.

[24] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll, An overview of JML tools and applications), International Journal on Software Tools for Technology Transfer 7 (3) (2005) 212–232.

[25] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, H. Venter, Specification and verification: the Specˆ experience, Communications of the ACM 54 (6) (2011) 81–91.

[26] M. Blume, D. McAllester, Sound and Complete Models of Contracts, Journal of Functional Programming 16 (4–5) (2006) 375–414.

[27] D. N. Xu, Hybrid Contract Checking via Symbolic Simplification, in: Proceedings of the 21st ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2012), Association for Computing Machinery, 107–116, 2012.

[28] S. Antoy, M. Hanus, Contracts and Specifications for Functional Logic Programming, in: Proceedings of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012), vol. 7149 of *Lecture Notes in Computer Science*, Springer-Verlag, 33–47, 2012.

[29] The Maude Tools Web site, Available at: `http://maude.cs.illinois.edu/w/index.php?title=Maude_Tools`, 2015.

[30] A. Riesco, A. Verdejo, N. Martí-Oliet, R. Caballero, Declarative Debugging of Rewriting Logic Specifications, The Journal of Logic and Algebraic Programming 81 (7–8) (2012) 851–897.

[31] E. Y. Shapiro, Algorithmic Program Diagnosis, in: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1982), Association for Computing Machinery, 299–308, 1982.

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

[33] M. Roldán, F. Durán, Dynamic Validation of OCL Constraints with mOdCL, in: Proceedings of the 11th International Workshop on OCL and Textual Modelling (OCL 2011), vol. 44, Electronic Communications of the EASST, 2011.

[34] F. Chen, G. Roşu, Parametric Trace Slicing and Monitoring, in: Proceedings of the 15th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2009), vol. 5505 of *Lecture Notes in Computer Science*, Springer-Verlag, 246–261, 2009.

[35] TeReSe, Term Rewriting Systems, Cambridge University Press, ISBN 978-05-2139-115-3, 2003.

[36] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, Maude Manual (Version 2.7), Tech. Rep., SRI International Computer Science Laboratory, available at: `http://maude.cs.uiuc.edu/maude2-manual/`, 2015.

[37] F. Baader, W. Snyder, Unification Theory, in: J. A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, vol. I, Elsevier Science, 447–533, 2001.

[38] J. Klop, Term Rewriting Systems, in: S. Abramsky, D. Gabbay, T. Maibaum (Eds.), Handbook of Logic in Computer Science, vol. I, Oxford University Press, 1–112, 1992.

[39] R. Bruni, J. Meseguer, Semantic Foundations for Generalized Rewrite Theories, Theoretical Computer Science 360 (1–3) (2006) 386–414.

[40] F. Durán, J. Meseguer, On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories, The Journal of Logic and Algebraic Programming 81 (2012) 816–850.

[41] J. L. Lassez, M. J. Maher, K. Marriott, Unification Revisited, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, Los Altos, California, 587–625, 1988.

[42] M. Alpuente, S. Escobar, J. Meseguer, P. Ojeda, Order–Sorted Generalization, Electronic Notes in Theoretical Computer Science 246 (2009) 27–38.

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

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

[45] G. Roşu, From Rewriting Logic, to Programming Language Semantics, to Program Verification, in: Logic, Rewriting, and Concurrency - Festschrift Symposium in Honor of José Meseguer, vol. 9200 of *Lecture Notes in Computer Science*, Springer-Verlag, 598–616, 2015.

[46] M. Alpuente, D. Ballis, J. Espert, D. Romero, Backward Trace Slicing for Rewriting Logic Theories, in: Proceedings of the 23rd International Conference on Automated Deduction (CADE 2011), vol. 6803 of *Lecture Notes in Computer Science*, Springer-Verlag, 34–48, 2011.

[47] M. Alpuente, D. Ballis, F. Frechina, 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), vol. 7180 of *Lecture Notes in Computer Science*, Springer-Verlag, 62–76, 2012.

[48] M. Alpuente, D. Ballis, F. Frechina, J. Sapiña, Slicing-Based Trace Analysis of Rewriting Logic Specifications with *i*JULIENNE, in: Proceedings of the 22nd European Symposium on Programming (ESOP 2013), vol. 7792 of *Lecture Notes in Computer Science*, Springer-Verlag, 121–124, 2013.

[49] Anima, The Anima Web site, Available at: `http://safe-tools.dsic.upv.es/anima`, 2014.

[50] iJulienne, The *i*JULIENNE Web site, Available at: `http://safe-tools.dsic.upv.es/iJulienne`, 2012.

[51] M. Alpuente, S. Escobar, J. Espert, J. Meseguer, ACUOS: A System for Modular ACU Generalization with Subtyping and Inheritance, in: Proceedings of the 14th European Conference on Logics in Artificial Intelligence (JELIA 2014), vol. 8761 of *Lecture Notes in Computer Science*, Springer-Verlag, 573–581, 2014.

[52] L. Helmink, M. P. A. Sellink, F. W. Vaandrager, Proof-Checking a Data Link Protocol, in: Proceedings of the 4th International Workshop on Types for Proofs and Programs (TYPES 1993), vol. 806 of *Lecture Notes in Computer Science*, Springer-Verlag, 127–165, 1994.

[53] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, The Maude 2.0 System, in: Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003), vol. 2706 of *Lecture Notes in Computer Science*, Springer-Verlag, 76–87, 2003.

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