Assertion-based Analysis via Slicing with ABETS∗ *(System Description)*

M. ALPUENTE, F. FRECHINA, J. SAPIÑA
DSIC-ELP, Universitat Politècnica de València

D. BALLIS
DIMI, University of Udine

Abstract

We present ABETS, an assertion-based, dynamic analyzer that helps diagnose errors in Maude programs. ABETS implements trace and program slicing to automatically create reduced versions of the executed programs and runs in which any information that is not relevant to the bug currently being diagnosed is removed. In addition, ABETS employs runtime assertion checking to automate the identification of bugs so that whenever an assertion is violated, the system automatically infers accurate slicing criteria from the failure. We summarize the main services provided by ABETS, which also include a novel assertion-based facility for program repair that tries to automatically fix buggy programs when a state invariant is refuted. Finally, we provide an experimental evaluation that shows the performance and effectiveness of the system.

1 Introduction

Bug diagnosis is a time-consuming and, most often, tedious manual task that forces developers to painstakingly examine large volumes of complex execution traces while trying to locate the actual cause of observable misbehaviors. This paper describes a dynamic program analyzer called ABETS, which aims to mitigate the costs of diagnosing errors in concurrent programs that are written in Maude.

Maude is a language and a system that efficiently implements Rewriting Logic (RWL) (Meseguer 1992), which is a logic of change that seamlessly unifies a wide variety of models of concurrency. Thanks to its logical basis, Maude provides a precise mathematical model, which allows it to be used as a declarative language and as a formal verification system. Maude supports rich formal specification, equational rewriting, and logical reasoning modulo *algebraic axioms* (such as associativity, commutativity, and identity), providing tools for a number of formal techniques that include theorem proving, protocol analysis, state space exploration, deductive verification, model transformation, constraint solving, and model checking. The execution traces generated by Maude (and by Maude tools, including the standard Maude debugger) are complex objects to analyze since they may contain a huge number of compound rewrite steps that, however, omit crucial information for debugging such as the application of algebraic axioms (which is concealed within

∗ This work has been partially supported by the EU (FEDER) and Spanish MINECO grant TIN2015-69175-C4-1-R, and by Generalitat Valenciana PROMETEOII/2015/013. J. Sapiña was supported by FPI-UPV grant SP2013-0083.

Maude's *equational matching* algorithm). While this maximizes efficiency and is certainly justified during the program operation, it further complicates the debugging. The dynamic analyzer ABETS described in this paper facilitates the debugging of Maude programs. It does it by drastically simplifying the size and complexity of the analyzed programs and runs while unveiling all relevant information for debugging, which is done by a fruitful combination of runtime assertion checking and slicing that was originally formalized in (Alpuente et al. 2016). In assertion-based slicing, the Maude program to be analyzed is supplemented with a set of logical assertions that are checked at runtime. Upon an assertion failure, an accurate set of discordant positions (called symptoms) is computed by comparing the *computed* erroneous state within the *expected* state pattern (as defined by the violated assertion), with the comparison being performed by using least general generalization *modulo* the algebraic axioms of the operators involved (Alpuente et al. 2014). By filtering out everything but the distilled disagreements, a so-called *slicing criterion* is synthesized that accurately identifies the (position of the) faulty information in the erroneous last state of the trace. Then, in order to locate the source of the error, a trace slicing procedure is automatically triggered that backpropagates the anomalous information. This is done by recursively computing the origins or *antecedents* (Field and Tip 1994) of the observed positions while removing everything but the computed antecedents at each step. The given combination of runtime checking and slicing yields a self-initiating, enhanced dynamic slicing technique that traverses the program execution and makes every single computation detail explicit while revealing only and all data in the trace that contribute to the criterion observed. As a by-product of the trace slicing process, an executable, dynamic program slice is also automatically extracted that captures the program subset that is concerned with the error.

Assertion-based slicing is efficiently implemented in ABETS for both Maude and Full Maude (Clavel et al. 2007), which is a powerful extension of Maude that provides support for object-oriented specification and advanced module operations. The major strength of the system is that no criterion or error symptom must be identified in advance because the assertions (or more precisely, their runtime checks) are used to synthesize the slicing criteria. This is a significant improvement over more traditional, hand-operated slicing in which the criteria for slicing need to be manually fixed.

Contributions. The basic algorithms behind ABETS were introduced in (Alpuente et al. 2016), where we evaluated them on a prototypical implementation of the system. This work describes the latest, full-fledged ABETS implementation, which improves system efficiency as well as the generality/flexibility of the overall technique. Specifically,

- We provide a thorough description of those novel implementation details and optimizations that have boosted the system performance.
- We report a new in-depth experimental evaluation of the system that assesses critical aspects of the tool such as the assertion-checking and slicing capabilities, and its input/output performance, which is a usual weakspot of tools developed in (Full) Maude.
- We outline an experimental repair technique that automatically suggests program corrections to fix the program faults that are detected whenever an assertion that models a state invariant is refuted. The corrected rules are guarded by (a suitable instance of) the state invariant so that the repaired rule is fired only if the invariant is fulfilled.
- The ABETS dynamic analyzer is publicly available at `http://safe-tools.dsic.upv.`

es/abets, where it can be downloaded and locally installed as a novel, stand-alone console application, or it can be remotely used via a user-friendly web interface.

Plan of the paper. After some preliminaries in Section 2, the ABETS system is described in Section 3. Section 4 describes an analysis session with ABETS. Section 5 reports on those implementation choices and optimizations that have considerably improved the tool performance. Experimental results are given in Section 6. A brief discussion of related tools and concluding remarks are provided in Section 7. For optimizing our system we developed some new high-performance Maude operations that are summarized in Appendix. The appendix is not intended for publication but only meant to facilitate the review.

2 Modeling Concurrent Systems in Maude: A Leading Example

Concurrent systems can be formalized through Maude programs. A Maude program essentially consists of two components, E and R, where E is a canonical (membership) equational theory that models system states as algebraic entities (i.e., terms of an algebraic data type), and R is a set of rewrite rules that define transitions between states. Algebraic structures often involve axioms like associativity (A), commutativity (C), and/or identity (U) of function symbols, which cannot be handled by ordinary term rewriting but instead are handled implicitly by working with congruence classes of terms. Representatives of these congruence classes are chosen for practical implementation purposes, and the standard pattern matching of term rewriting is replaced by sophisticated, equational matching algorithms that are specific to the equational theories in use. This is why the membership equational theory E is decomposed into a disjoint union $E = \Delta \uplus Ax$, where the set Δ consists of (conditional) equations and membership axioms (i.e., axioms that assert the type or *sort* of some terms) that are implicitly oriented from left to right as rewrite rules (and operationally used as simplification rules), and Ax is a set of algebraic axioms (i.e., distinguished equations that define commonly occurring properties such as associativity, commutativity, and identity for some program operators). These axioms are only used for Ax-matching and implicitly expressed as function attributes.

The concurrent system evolves by rewriting states using *equational rewriting*, i.e., rewriting with the rewrite rules in R *modulo* the equations and axioms in E (Meseguer 1992). More precisely, execution traces (i.e., system computations) correspond to rewrite sequences $t_0 \xrightarrow{r_0}_E t_1 \xrightarrow{r_1}_E \ldots$, where $t \xrightarrow{r}_E t'$ denotes a transition (modulo E) from state t to t' via the rewrite rule of R that is uniquely labeled with r. Note that each single transition $t \xrightarrow{r}_E t'$ is computed as a rewrite chain $t \to^*_\Delta (t_{\downarrow_\Delta}) \xrightarrow{r} t'$, where the prefix $t \to^*_\Delta (t_{\downarrow_\Delta})$ is an equational simplification sequence that rewrites t into its canonical (i.e., irreducible) form $(t_{\downarrow_\Delta})$ using the oriented equations in Δ. Although advisedly omitted in our notation, all rewrites in the chain (either applying r or any of the equations in Δ) are performed *modulo Ax*.

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

Example 2.1
Let us introduce a (faulty) rewrite theory that specifies a simplified[1]stock exchange concurrent system, in which traders operate on stocks via limit orders, that is, orders that set the upper bound

[1] Maude's syntax is hopefully self-explanatory. Due to space limitations and for the sake of clarity, we only highlight those details of the system that are relevant to this work. A complete Maude specification of the stock exchange model is available at the ABETS website at `http://safe-tools.dsic.upv.es/abets`.

(price *limit*) at which traders want to buy stocks. When the stock price drops until it equals the price limit, the associated order is *opened* and the trader buys the stocks at the price limit. An order is automatically *closed* and the associated stocks are sold when the stock price P exceeds the purchase price limit L in a predetermined *profit target* PT (i.e., $P - L \geq PT$) or $L - P$ exceeds a predetermined *stop loss* SL (i.e., $L - P \geq SL$).

```
eq  [prefT] : PreferredTraders = 'T2 .
cmb [premT] : tr(TID,C) : PremiumTrader if TID in PreferredTraders .
rl  [next-rnd] : R : SS | TS | OS => R + 1 : updP(R+1,reSeed(R+1),SS) | TS | OS .
crl [open-ord] :
          R : (st(SID,P),SS) | (tr(TID,C),TS) | (ord(OID,TID,SID,L,PT,SL,close),OS) =>
          R : (st(SID,P),SS) | (tr(TID,C - P),TS) | (ord(OID,TID,SID,L,PT,SL,open),OS)
          if P == L .
crl [close-ord-SL] :
          R : (st(SID,P),SS) | (tr(TID,C),TS) | (ord(OID,TID,SID,L,PT,SL,open),OS) =>
          R : (st(SID,P),SS) |  (tr(TID,C + L - SL),TS) | OS
          if P <= L - SL .
crl [close-ord-PT] :
          R : (st(SID,P),SS) | (tr(TID,C),TS) | (ord(OID,TID,SID,L,PT,SL,open),OS) =>
          R : (st(SID,P),SS) | (tr(TID,C + L + PT),TS) | OS
          if P >= L + PT .
eq [updP] : updP(R,S,(st(SID,P),SS)) =
          if (rndDelta(R * S) rem 2) == 0
          then st(SID,S + rndDelta(R * S)),updP(R,S + 1,SS)
          else st(SID,S - rndDelta(R * S)),updP(R,S + 1,SS)
          fi .
eq [updP-owise] : updP(R,S,empty) = empty [owise] .
```

Fig. 1. (Conditional) rewrite rules and equations modeling the stock exchange system.

Within our system model, variable names are fully capitalized, while names that begin with the symbol ' are constant identifiers for traders, stocks and orders. System states have the form `R : SS | TS | OS`, where R is a natural number (called round) that models the market time evolution, and SS, TS, and OS are sets[2] of stocks, traders, and orders, respectively.

Stocks are modeled as terms `st(SID,P)` with SID being the stock identifier and P being the current stock price. Traders are modeled as `tr(TID,C)`, where TID is the trader identifier and C is the trader's available capital. We consider two classes of traders: premium traders and ordinary (or non-premium) traders. Premium traders are allowed to buy even if they run out of capital. Premium traders are identified by the conditional membership axiom `premT` (see Figure 1) that simply checks whether the trader identifier belongs to the (hard-coded) list `PreferredTraders`, which in this example just contains the premium trader 'T2.

Orders are specified by terms of the form `ord(OID,TID,SID,L,PT,SL,ST)`, which record the order identifier OID, the trader identifier TID, the stock identifier SID, the stock price limit L, the profit target PT, the stop loss SL, and the order status (which can be either `open` or `close`). For simplicity, an order allows a single stock to be traded at a time. This is not a limitation since multiple stocks can be managed by multiple orders.

Basic operations of the stock exchange model (i.e., market time evolution, opening and closure of orders) are implemented via the rules and equations of Figure 1. The `open-ord` rule

[2] To specify sets of X-typed elements, we instantiate the Maude parameterized sort `Set{X}`, which defines sets as associative, commutative, and idempotent lists of elements that is simply written as $(e_1,\ldots,e_n)$. The empty set is denoted by the constant symbol `empty`.

opens a trader order only if the stock price `P` equals the order price limit `L`. Once the order has been opened, the stock price is subtracted from the trader's capital, thereby updating the capital. Furthermore, the order status changes from `close` to `open`. Note that, in the set of stocks (`st(SID,P),SS`), the stock `st(SID,P)` is distinguished from all other stocks `SS` in the system. Similarly, the `close-ord-SL` rule closes a trader order for the stock `SID` and removes it from the current state when the `SID` stock price drops under the `L` − `SL` stop loss threshold. The trader's capital is then updated by adding the amount paid by the trader when the order was issued (i.e., the price limit `L`). The capital is also decreased by the predetermined stop loss `SL`. The `close-ord-PT` rule is similar and closes an order when its stock price satisfies the profit target. Finally, the `next-rnd` rule models the time evolution by simply increasing the round number by one and then automatically updating the stock prices by means of the function `updP`, which randomly increases or decreases the stock prices via the naïve pseudo-random number generator `rndDelta` that is re-seeded at the beginning of each round with the round tick `R+1`.

Note that the specification given in Figure 1 contains two sources of error. First, the function `updP` is flawed because it could generate negative stock prices, which are meaningless and should be disallowed. Second, the rule `open-ord` does not check if the available capital of a non-premium trader is enough to cover the order price limit. For instance, for the ordinary Trader `'T`, the following reachability goal (which can be solved in Maude via the `search` command[3])

```
(1 : st('S,8) | tr('T,9) | ord('O,'T,'S,12,4,3,close)) =>* R : SS | tr('T,C) | OS .
```

computes (among other solutions) the substitution `{R/3, SS/st('S, 12), C/-3, OS/ord('O, 'T, 'S, 12, 4, 3, open)}` that witnesses the existence of an execution trace that starts from the specified initial state and ends in a final state with a faulty, negative capital `C=-3`.

3 Assertion-based Program Analysis with ABETS

ABETS implements an automated trace slicing technique based on (Alpuente et al. 2014) that facilitates the analysis of Maude programs by drastically reducing the size and complexity of entangled, textually-large execution traces. The technique first uncovers data dependences within the execution trace $\mathcal{T}$ w.r.t. a slicing criterion (i.e., a set of selected symbols in the last state of $\mathcal{T}$) and then produces a trace slice $\mathcal{T}^\bullet$ of $\mathcal{T}$ in which pointless information that is detected to be irrelevant w.r.t. the chosen criterion (i.e., symbols in $\mathcal{T}$ that are not origins or *antecedents* of the observed symbols) is replaced with the special variable symbol •.

Unlike the original trace slicing methodology of (Alpuente et al. 2014) where the slicing criterion must be *manually* determined in advance by the user, ABETS encompasses a runtime assertion-checking mechanism (which is built on top of the slicing engine) that was originally formalized in (Alpuente et al. 2016) and preserves the program semantics. This mechanism allows the slicing criteria to be *automatically* inferred from falsified assertions, thereby offering a more automatic support to the analysis of erroneous programs and traces.

While we employ equational *unification modulo* axioms to implement the origin-tracking procedure that properly backtracks the data dependencies along the trace, we use the *generalization*

[3] Given a (possibly) non–ground term `s`, Maude's `search` command checks whether a reduct of t is an instance (modulo the program equations and axioms) of `s` and delivers the corresponding (equational) matcher as the computed solution.

(i.e., anti-unification) algorithm *modulo* axioms of (Alpuente et al. 2014) to identify semantic disagreements of the program behavior w.r.t. the assertions. Also, the new program autofix capability of ABETS, described in Section 3.2, employs *unification modulo* axioms to generate adequate conditions for the buggy program rules.

3.1 Assertion-based Slicing in ABETS*: Core Functionality*

ABETS supports two types of assertions: system assertions and functional assertions.

i) *System* assertions: Their general syntax is $S\{\varphi\}$, where S is a term (called *state template*), and φ is a logic formula in conjunctive normal form $\varphi_1 \wedge \ldots \wedge \varphi_n$. Roughly speaking, a system assertion $S\{\varphi\}$ defines a state invariant that must be satisfied by all system states that match (modulo the equational theory E) the state template S.

Example 3.1
The following system assertion specifies that the capital of ordinary traders must be non-negative in every system state of the trace:

```
  tr(TID:TraderID,C:Int) { ordinary(tr(TID:TraderID,C:Int)) implies C:Int >= 0 }
```

where the new predicate `ordinary(T)` checks whether `T` is a non-premium trader in the Maude program of Example 2.1:

```
op ordinary : Trader -> Bool .
eq ordinary(tr(TID:TraderID,C:Int)) = not(tr(TID:TraderID,C:Int) :: PremiumTrader).
```

ii) *Functional* assertions: Their general form is $I\,\{\varphi_{in}\} \to O\,\{\varphi_{out}\}$ where I, O are terms, and $\varphi_{in}, \varphi_{out}$ are logic formulas in conjunctive normal form. Intuitively, functional assertions specify pre- and post-conditions over the equational simplification $t \to^{*}_{\Delta} (t_{\downarrow_\Delta})$ that heads the rewriting $t \xrightarrow{r}_{E} t'$ of any term t in the system trace by providing: (i) an input template I that t can match and a pre-condition φ_{in} that t can meet; (ii) an output template O that the canonical form $(t_{\downarrow_\Delta})$ of t has to match and a post-condition φ_{out} that $(t_{\downarrow_\Delta})$ has to meet (whenever the input term t matching I meets φ_{in}).

Example 3.2
Consider again the Maude program of Example 2.1. The functional assertion

```
      updP(R:Nat,S:Nat,(st(SID:StockID, P:Int),SS:Set{Stock})) { P:Int >= 0 }
                  -> (st(SID:StockID, P':Int),SS':Set{Stock}) { P':Int >= 0 }
```

specifies that stock market fluctuations modeled by function `updP` generate non-negative stock prices provided that the input stock prices are also non-negative.

The satisfiability of the provided assertions can be checked in two different modes, either as a *synchronous* (and trace-storing) procedure that incrementally executes, checks, and potentially slices execution traces at runtime, or as an *asynchronous* (off-line) procedure that processes a previously computed execution trace against the set of provided assertions. In ABETS, system traces can be easily generated by providing both an initial and a final reachable state. As for equational simplification traces, they can be generated by simply providing the initial term, which is then simplified to its irreducible form.

Synchronous as well as asynchronous assertion checking is implemented via equational rewriting that automatically reduces all matched assertions to Boolean truth values.

Example 3.3
Consider the Maude program of Example 2.1 and the execution trace $\mathcal{T} = s_0 \xrightarrow{\texttt{next-rnd}} s_1 \xrightarrow{\texttt{open-ord}} s_2$ that starts in the initial state

$s_0 =$ `1 : (st('S1,23), st('S2,8)) | (tr('T1,9), tr('T2,20)) | ord('O1,'T1,'S2,12,4,3,close)`

and ends in the state

$s_2 =$ `2 : (st('S1,4), st('S2,12)) | (tr('T1,-3), tr('T2,20)) | ord('O1,'T1,'S2,12,4,3,open)`

The negative capital of the ordinary trader `'T1` in the state s_2 is demonstrably wrong by the refutation of the system assertion of Example 3.1. Hence, ABETS automatically computes the slicing criterion `tr('T1,-3)` that pinpoints this faulty information and produces the trace slice $\mathcal{T}^\bullet$ of Figure 2, which represents a partial view of the system evolution that focuses on `T1`'s trading actions and exposes the erroneous behaviour of the `open-ord` rule to user inspection.

```
             1 : st(●,●),st(●,●) | tr('T1,9),● | (ord(●,'T1,●,12,●,●,close))
next-rnd
   ●→        ● : ●,st(●,12)) | tr('T1,9),● | ord(●,'T1,●,12,●,●,close)
open-ord
   ●→        ● : ● | tr('T1,-3),● | ●
```

Fig. 2. Trace slice for automatically synthesized criterion `tr('T1,-3)`.

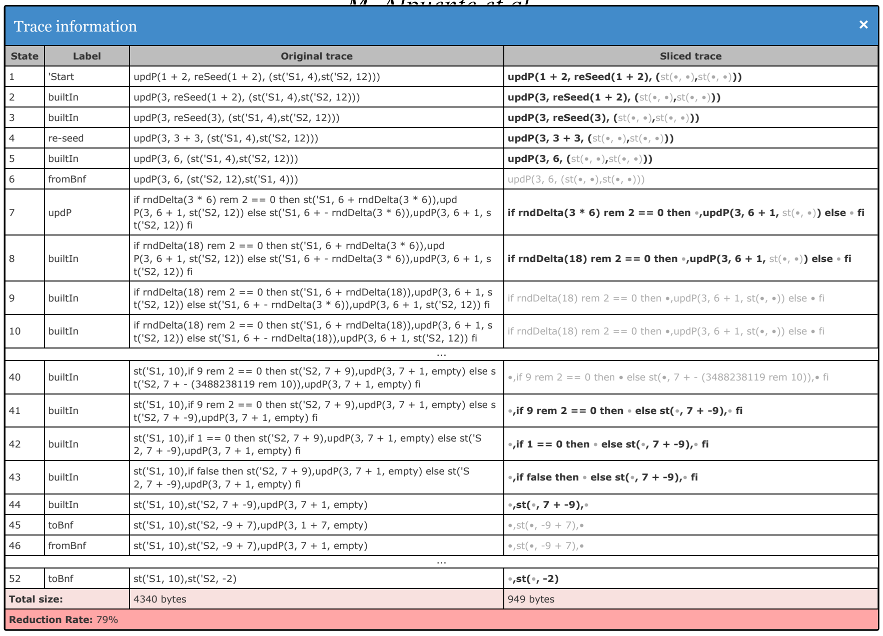
ABETS also provides a handy way to automatically synthesize refined slicing criteria by means of special variables (whose name begins with ♯) that can be used in the assertions to indicate pieces of the matched term that the user does not want to observe along the generated trace slice. For instance, if we replace `TID:TraderID` with `♯TID:TraderID` in the system assertion of Example 3.1, we compute the refined criterion `tr(●,-3)` for the trace $\mathcal{T}$ of Example 3.3. A more detailed analysis session with ABETS can be found in Section 4.

3.2 Additional Analysis Features

In addition to the described automatic slicing technique, ABETS provides the following additional services and features:

Backward/forward incremental trace slicing and trace inspection. ABETS supports stepwise user-directed, backward and forward incremental refinements of the slicing. This allows the computed slices to be further sliced (downwards or upwards) to create smaller and smaller slices of both program and trace, which helps isolate the meandering antecedents or descendants *modulo* the equations and axioms of any given expression in the trace.

Within ABETS, the user can easily inspect traces in full detail by expanding each state transition into its complete rewrite chain that contains equational simplification and algebraic axioms and built-in operator applications, which are usually hidden in the traces delivered by existing Maude tools, including the Maude interpreter. The recursive inspection of the conditions that are

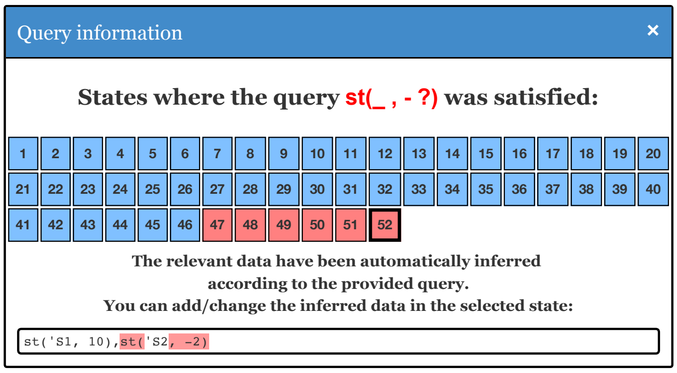


Trace information ×

State	Label	Original trace	Sliced trace
1	'Start	updP(1 + 2, reSeed(1 + 2), (st('S1, 4),st('S2, 12)))	**updP(1 + 2, reSeed(1 + 2), (**st(•, •)**,**st(•, •)**))**
2	builtIn	updP(3, reSeed(1 + 2), (st('S1, 4),st('S2, 12)))	**updP(3, reSeed(1 + 2), (**st(•, •)**,**st(•, •)**))**
3	builtIn	updP(3, reSeed(3), (st('S1, 4),st('S2, 12)))	**updP(3, reSeed(3), (**st(•, •)**,**st(•, •)**))**
4	re-seed	updP(3, 3 + 3, (st('S1, 4),st('S2, 12)))	**updP(3, 3 + 3, (**st(•, •)**,**st(•, •)**))**
5	builtIn	updP(3, 6, (st('S1, 4),st('S2, 12)))	**updP(3, 6, (**st(•, •)**,**st(•, •)**))**
6	fromBnf	updP(3, 6, (st('S2, 12),st('S1, 4)))	updP(3, 6, (st(•, •),st(•, •)))
7	updP	if rndDelta(3 * 6) rem 2 == 0 then st('S1, 6 + rndDelta(3 * 6)),updP(3, 6 + 1, st('S2, 12)) else st('S1, 6 + - rndDelta(3 * 6)),updP(3, 6 + 1, st('S2, 12)) fi	**if rndDelta(3 * 6) rem 2 == 0 then** •**,updP(3, 6 + 1,** st(•, •)**) else** • **fi**
8	builtIn	if rndDelta(18) rem 2 == 0 then st('S1, 6 + rndDelta(3 * 6)),updP(3, 6 + 1, st('S2, 12)) else st('S1, 6 + - rndDelta(3 * 6)),updP(3, 6 + 1, st('S2, 12)) fi	**if rndDelta(18) rem 2 == 0 then** •**,updP(3, 6 + 1,** st(•, •)**) else** • **fi**
9	builtIn	if rndDelta(18) rem 2 == 0 then st('S1, 6 + rndDelta(18)),updP(3, 6 + 1, st('S2, 12)) else st('S1, 6 + - rndDelta(3 * 6)),updP(3, 6 + 1, st('S2, 12)) fi	if rndDelta(18) rem 2 == 0 then •,updP(3, 6 + 1, st(•, •)) else • fi
10	builtIn	if rndDelta(18) rem 2 == 0 then st('S1, 6 + rndDelta(18)),updP(3, 6 + 1, st('S2, 12)) else st('S1, 6 + - rndDelta(18)),updP(3, 6 + 1, st('S2, 12)) fi	if rndDelta(18) rem 2 == 0 then •,updP(3, 6 + 1, st(•, •)) else • fi
...			
40	builtIn	st('S1, 10),if 9 rem 2 == 0 then st('S2, 7 + 9),updP(3, 7 + 1, empty) else st('S2, 7 + - (3488238119 rem 10)),updP(3, 7 + 1, empty) fi	•,if 9 rem 2 == 0 then • else st(•, 7 + - (3488238119 rem 10)),• fi
41	builtIn	st('S1, 10),if 9 rem 2 == 0 then st('S2, 7 + 9),updP(3, 7 + 1, empty) else st('S2, 7 + -9),updP(3, 7 + 1, empty) fi	•**,if 9 rem 2 == 0 then** • **else st(**•**, 7 + -9),**• **fi**
42	builtIn	st('S1, 10),if 1 == 0 then st('S2, 7 + 9),updP(3, 7 + 1, empty) else st('S2, 7 + -9),updP(3, 7 + 1, empty) fi	•**,if 1 == 0 then** • **else st(**•**, 7 + -9),**• **fi**
43	builtIn	st('S1, 10),if false then st('S2, 7 + 9),updP(3, 7 + 1, empty) else st('S2, 7 + -9),updP(3, 7 + 1, empty) fi	•**,if false then** • **else st(**•**, 7 + -9),**• **fi**
44	builtIn	st('S1, 10),st('S2, 7 + -9),updP(3, 7 + 1, empty)	•**,st(**•**, 7 + -9),**•
45	toBnf	st('S1, 10),st('S2, -9 + 7),updP(3, 1 + 7, empty)	•,st(•, -9 + 7),•
46	fromBnf	st('S1, 10),st('S2, -9 + 7),updP(3, 7 + 1, empty)	•,st(•, -9 + 7),•
...			
52	toBnf	st('S1, 10),st('S2, -2)	•**,st(**•**, -2)**
Total size:		4340 bytes	949 bytes
Reduction Rate: 79%			

Fig. 3. Extended view of (a fragment of) the computed trace slice (making explicit the application of built-in operators and algebraic axioms) after refuting the functional assertion of Example 3.2.

evaluated in conditional rewrite steps is also available for thorough exploration.



Query information ×

States where the query st(_ , - ?) was satisfied:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52								

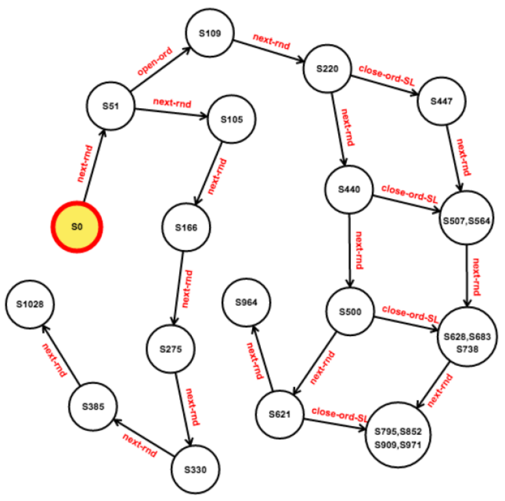
The relevant data have been automatically inferred according to the provided query. You can add/change the inferred data in the selected state:

```
st('S1, 10),st('S2, -2)
```

Fig. 4. Result of the trace query `st(_, - ?)`.

Trace querying and manipulation. This feature allows information of interest to be searched in huge execution traces by undertaking a query that specifies a template for the search (see Figure 3). This query is a filtering pattern with wildcards that define irrelevant symbols by means of the underscore character (_) and relevant symbols by means of the question mark character (?). In addition, traces and trace slices can be manipulated using their meta-level representation to be exported to other Maude tools. The meta-representation of terms can be visually displayed, which is particularly useful for the analysis of object-oriented computations where some object attributes can only be unambiguously visualized in the meta-level (desugared) states.

Computation graph exploration. To help identify traces of interest for asynchronous checking, ABETS supports two different representations of the computation space for a given initial term: the (standard) tree



representation that is provided by default and a graph representation of the state space that can improve user's understanding of the program behavior (see Figure 5).

Autofix of program rules. ABETS is provided with an automatic program repair facility that suggests fixes to potentially buggy rewrite rules whenever ABETS detects a faulty system state of a trace $\mathscr{T}$ that does not satisfy a system assertion $S\{\varphi\}$. Roughly speaking, the technique transforms the rewrite rule that is responsible for the system assertion failure (i.e., the last applied rule in $\mathscr{T}$ that "embeds" the state pattern S), by adding a constrained instance of the logic formula φ into the conditional part of the rule. Such an instance is computed by using Maude's built-in variant-based unification modulo axioms (Durán et al. 2016) to unify S with (a renamed apart version of) the right-hand side of the rewrite rule under examination. Let us see a simple example.

Example 3.4
Consider a Maude program that includes the following rewrite rule `r` together with the system assertion `c(2,g(1,Z)) {Z < 3}`. Assume that the symbol `c` is declared commutative.

```
crl [r] : f(X,Y) => c(g(X,Y),2) if X > 0 .
```

The execution trace $\texttt{f}(1,3) \xrightarrow{\texttt{r}} \texttt{c}(\texttt{g}(1,3),2)$ is erroneous since the formula `Z < 3` does not hold in the state `c(g(1,3),2)`. Indeed, the second argument of `g` is not lower than `3`.

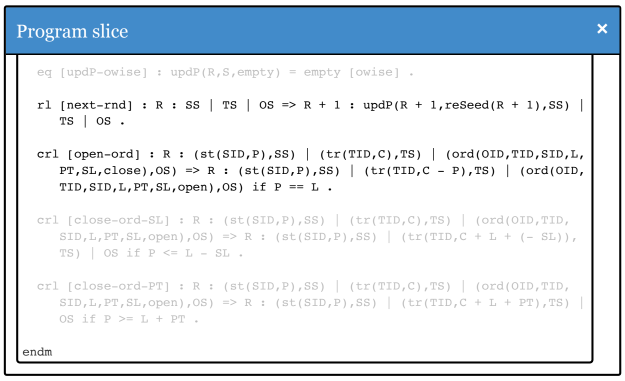
The repair proceeds by first computing a unifier (modulo commutativity of the operator `c`) between the state pattern `c(2,g(1,Z))` and the right-hand side of `r` (i.e., `c(g(X,Y),2)`), which is $\theta =$`{X/1, Z/Y}`; then, the condition of `r` is augmented by adding the formula `{Z < 3}`, which is constrained by the bindings of θ, thereby computing the following fix:

```
crl [rfix] : f(X,Y) => c(g(X,Y),2) if X > 0 /\ ((X := 1 and Z := Y) implies Z < 3) .
```

which is then simplified into

```
crl [rfix] : f(X,Y) => c(g(X,Y),2) if X > 0 /\ ((X := 1 implies Y < 3) .
```

Note that the generated condition $\texttt{C}_{\texttt{fix}}$ of a repaired rule `rfix` might not be satisfiable, which makes the rule completely useless since it cannot be applied. Hence, the original rule can be simply removed and a warning message be issued to the user, who can try to fix the bug by writing a new rule or by relaxing the refuted assertion. This is not implemented in ABETS yet because there



```
eq [updP-owise] : updP(R,S,empty) = empty [owise] .

rl [next-rnd] : R : SS | TS | OS => R + 1 : updP(R + 1,reSeed(R + 1),SS) |
   TS | OS .

crl [open-ord] : R : (st(SID,P),SS) | (tr(TID,C),TS) | (ord(OID,TID,SID,L,
   PT,SL,close),OS) => R : (st(SID,P),SS) | (tr(TID,C - P),TS) | (ord(OID,
   TID,SID,L,PT,SL,open),OS) if P == L .

crl [close-ord-SL] : R : (st(SID,P),SS) | (tr(TID,C),TS) | (ord(OID,TID,
   SID,L,PT,SL,open),OS) => R : (st(SID,P),SS) | (tr(TID,C + L + (- SL)),
   TS) | OS if P <= L - SL .

crl [close-ord-PT] : R : (st(SID,P),SS) | (tr(TID,C),TS) | (ord(OID,TID,
   SID,L,PT,SL,open),OS) => R : (st(SID,P),SS) | (tr(TID,C + L + PT),TS) |
   OS if P >= L + PT .

endm
```

Fig. 6. Program slice computed after the system assertion refutation.

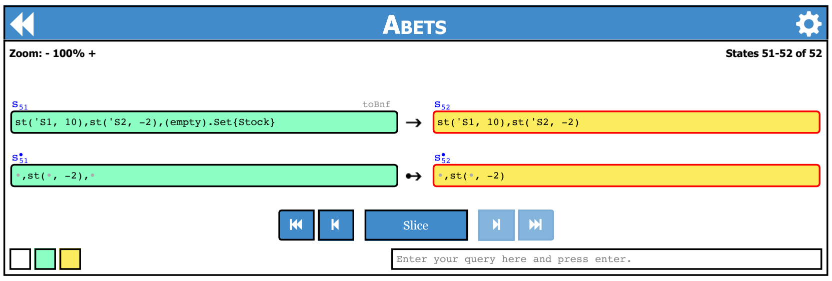
is no SMT solver currently linked to Maude 2.7. We are currently working on providing Mau-Dev with a SMT backend.

Program Slicing. ABETS delivers a program slice that includes all and only the rules/equations applied in the computed trace slice which exposes the error (see Figure 6).

4 An Analysis Session with ABETS

Let us analyze the running stock exchange Maude specification described in Section 2 w.r.t. the assertions formalized in Examples 3.1 and 3.2, and the initial system state s_0 of Example 3.3. We consider the synchronous checking modality that non-deterministically expands all Maude steps that originate from the initial state s_0. In this mode, ABETS allows the analysis coverage to be tuned by choosing to check either a single branch of the execution tree, or all of the branches up to a given depth. By default, the depth of the deployed computation tree is set to 10 and the number of nodes to be checked is set to 100K. After the example code is loaded, we start this session by choosing to analyze all branches of the computation tree for s_0. By pressing the CHECK button, ABETS immediately discovers within the tree an erroneous equational simplification trace that leads to the canonical form `st('S1,10),st('S2,-2)`, where the functional assertion of Example 3.1 is falsified due to the negative price for stock `'S2`. A functional error symptom that unambiguously signals the anomalous stock price `-2` for `'S2` is then issued by the tool, and the slicing criterion `•,st(•,-2)` for the identified faulty trace is automatically synthesized. Then, the trace slice shown in Figure 3 is delivered, unequivocally signaling the malfunction of the stock price update function `updP` and its ancillary functions.

ABETS also allows us to querying the trace (see Figure 4) and to interactively navigate back and forth through the delivered trace slice to thoroughly inspect the origins (or influences upon) and the impact (or consequences) of the malfunction (see Figure 7). Moreover, by selecting the *Show trace information* option in the main menu, we can access: (i) an instrumented version of the input trace that shows each rule/equation/axiom transformation step in a fine-grained way; (ii) several views of the computed trace slice that support different program comprehension levels (e.g., it is possible to hide/expose the state transformations given by algebraic axiom applications); (iii) a Maude meta-representation of the computed trace slice. On the other hand, all rewrite steps, including equational simplification steps and *built-in* operator application steps, can be easily inspected by accessing the *show transition information* option in the context menu of each state. The accessible information includes: (i) the rule or equation applied,



s_{51}
```
st('S1, 10),st('S2, -2),(empty).Set{Stock}
```
$\rightarrow$ s_{52}
```
st('S1, 10),st('S2, -2)
```
$s^\bullet_{51}$
```
•,st(•, -2),•
```
$\leftrightarrow$ $s^\bullet_{52}$
```
•,st(•, -2)
```

Fig. 7. Navigable trace slice after refuting the functional assertion.

(ii) the computed substitution, and (iii) the position in the state where the rule was applied. Moreover, for conditional rewrite steps, an in-depth analysis of the condition proofs can be accessed through the *inspect condition* option of the same menu. A convenient *explore computation space* option of ABETS's context menu provide access to tree and graph representations of (the generated fragment of) the computation space for initial state s_0 (see Figure 5).
After correcting the faulty `updP` equational definition, if we re-execute the analysis, we can also discover the system assertion violation described in Example 3.3. Also, the following fix for the `open-ord` rule is suggested:

```
crl [oo-fix] : R : (st(SID,P),SS) | (tr(TID,C),TS) | (ord(OID,TID,SID,L,PT,SL,close),OS) =>
    R : (st(SID,P),SS) | (tr(TID,C - P),TS) | (ord(OID,TID,SID,L,PT,SL,open),OS)
        if P == L /\ ordinary(tr(TID,C) implies ((C - P) >= 0)) .
```

Note that the rule `oo-fix` corrects the buggy original rule since it allows ordinary traders to open orders only if their capital `C` covers the stock price `P`.

Finally, by running the *program slice* option, the rewrite rules `next-round` and `open-ord` (and dependent statements) are automatically isolated and highlighted (see Figure 6).

5 Implementation Details and Optimizations

The architecture of ABETS consists of the following: (i) a Maude-based slicer and constraint-checker core that can run at both 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 (v) an intuitive user interface that is based on AJAX technology and written in HTML5 canvas and Javascript. ABETS contains about 3500 lines of Maude code, 1000 lines of C++ code, 1000 lines of Java code, and 3000 lines of Javascript code. The system has been (re-)implemented by primarily focusing on its performance, including improvements for both the analysis and for the input and output operations.

Analysis optimizations. One of the many features of ABETS is its ability to manipulate all the relevant information regarding the application of equations, algebraic axioms, and built-in operators at the meta-level, which is a feature that is not supported by Maude. We implemented this extension in a new developer version of the Maude system called Mau-Dev[4] without affecting the efficiency of the latest Maude 2.7 release. Also, to boost the system performance, the functions that are more frequently used in ABETS have been reimplemented in C++ as new, highly efficient, built-in Mau-Dev (meta-level) operations that are listed in the appendix.

I/O optimizations. Maude's efficient parser allows very large initial calls to be efficiently parsed in just a few milliseconds. In contrast, Full Maude's parser is entirely developed in Maude itself; hence, its efficiency can be seriously penalized when dealing with mixfix operator definitions due to extensive backtracking. As a result, ABETS initial calls that contain large and complex execution traces as arguments typically took some minutes to be loaded into our previous system (Alpuente et al. 2016). We have overcome this drawback by dynamically creating

[4] Mau-Dev is publicly available at `http://safe-tools.dsic.upv.es/maudev`

a *devoted module* that defines unique *placeholder* constants that are subsequently reduced to the actual arguments of the initial (Full Maude) call. This module is loaded prior to starting the Full Maude's *execution loop*. Thus, by taking advantage of the ability of Full Maude to access previously loaded Maude modules, the entire call can be parsed directly in Maude, except for its top-most operator.

The output of ABETS executions typically consists of a Maude term of sort `String`, represented in JSON (JavaScript Object Notation) format, that collects all the computed information (e.g., the source-level and meta-level representation of the original trace and the sliced trace, the associated program slice, and transition information between subsequent trace states). This output string is later processed by the ABETS front-end to offer a more friendly, visual representation. A naïve handling of the output string can be particularly time-consuming when dealing with huge execution traces that contain a large number of (complex) states. Since efficient output handling is crucial not to penalize the overall performance of the system, (meta) string conversion has also been implemented in C++, which achieves (on average) one order of magnitude speedup for the output operations w.r.t. (Alpuente et al. 2016).

Some experiments that highlight the efficiency gain of the optimized system w.r.t. (Alpuente et al. 2016) are shown in Section 6.

6 Experimental evaluation

To evaluate the performance of the ABETS system, we introduced defects in several Maude programs endowed with assertions and used the system to detect assertion violations. We benchmarked ABETS on the following collection of Maude programs, which are all available and fully described within the ABETS Web platform: *Bank model*, a conditional Maude specification that models a distributed banking system; *Blocks World*, a Maude encoding of the classical AI planning problem that consists of setting one or more vertical stacks of blocks on a table using a robotic arm; *BRP*, a Maude implementation of the Bounded Retransmission Protocol; *Dekker*, a Maude specification of Dekker's mutual exclusion algorithm; *Maze*, the nondeterministic Maude specification of a maze game where multiple players walk, jump, or collide while trying to reach a given exit point; *Philosophers*, a Maude specification of the classical Dijkstra concurrency example; *Rent-a-car (fm)*, a *Full Maude* program that models the logic of a distributed, object-oriented, online car-rental store; *Stock Exchange*, the leading example of this article; *Stock Exchange (fm)*, a *Full Maude*, object-oriented version of the *Stock Exchange* example; *Webmail*, a Maude specification of a rich webmail application that provides typical login/logout functionality, system administration capabilities, email management, etc. We note that ABETS automatically identifies theories that do not require Full Maude capabilities so that the highest possible analysis performance is achieved without incurring unnecessary costs.

In our experiments, we empirically evaluate both the effectiveness and performance of ABETS by (synchronously) checking each program against an assertional specification that contains at least one failing assertion. This way, an erroneous execution trace $\mathcal{T}_\varepsilon$ is delivered and subsequently simplified into a trace slice $\mathcal{T}_\varepsilon^\bullet$ w.r.t. slicing criteria that are automatically inferred. The experiments were conducted on a 3.3GHz Intel Xeon E5-1660 with 64GB RAM.

The achieved results are summarized in Table 1. For each benchmark program, we show the (average) `slowdown` per assertion check introduced by checking assertions during the trace generation process; the `inference time` for synthesizing the slicing criterion (in ms) the `size` of

Table 1. *Synchronous assertion-checking performance analysis*

Program	Slowdown (ms/check)	Inference Time (ms)	Size $\mathscr{T}_\varepsilon$ (kB)	Size $\mathscr{T}_\varepsilon^\bullet$ (kB)	Reduction
Bank Model	0.05	2	9.536	1.236	87%
Blocks World	0.04	1	0.279	0.046	84%
BRP	0.03	1	0.792	0.269	67%
Dekker	0.09	2	8.268	0.286	97%
Maze	0.05	1	2.747	0.423	85%
Philosophers	0.04	1	5.244	1.990	62%
Rent-a-car (fm)	0.09	5	5.507	0.115	98%
Stock Ex.	0.06	3	46.423	4.153	91%
Stock Ex. (fm)	0.09	5	195.397	20.862	89%
Webmail app	0.18	9	133.460	7.823	94%

the detected, erroneous execution trace $\mathscr{T}_\varepsilon$ (in kilobytes); the `size` of the sliced execution trace $\mathscr{T}_\varepsilon^\bullet$ (in kilobytes); and the `reduction` rate achieved (which quantitatively measures the amount of irrelevant information that has been removed from the original trace $\mathscr{T}_\varepsilon$ in $\mathscr{T}_\varepsilon^\bullet$).

Obviously, the slowdown of the whole checking process depends on the number of assertions that are contained in the specification and particularly on 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 logic formulas involved since the number of possible matchings (modulo axioms) with the system's states can grow very quickly. The slowdown can also be affected by the complexity of the user-defined predicates involved in the functional and system assertions to be checked. Of course, the more instantiated and "computationally light" the assertions are, the better the assertion checking performance is. Our experimental results indicate that the overhead due to assertion checking is reasonably low. Actually, our figures reveal very small slowdowns (0.07 ms/check on average), which is 70% of the average slowdown of (Alpuente et al. 2016) (0.1 ms/check) for the very same benchmark programs.

Our experiments also show very small synthesis times for the slicing criteria that grow linearly with the size of the erroneous state. This is particularly evident in the case of Webmail App, whose states are quite large (about 20 times the size of the Stock Ex. states). With regard to the time required to perform the slicing, our implementation is quite time efficient; the elapsed times are small even for very complex traces and also scale linearly. For example, running the slicer for a 50kB trace of a Maude program with about 150 rules and equations with ACU rewrites takes less than a few milliseconds. As for the trace slices that are *automatically* delivered by ABETS the reduction rates range from 98% to 62%, with an average reduction rate of 85%, which greatly facilitates the isolation and analysis of the faulty code.

Finally, the generation, parsing and output of traces (and trace slices) has been greatly improved in the current version of ABETS. This is not evident in synchronous mode because the input/output operations handle single state transitions. But when we run our benchmarks in asynchronous mode, which performs I/O offline on the whole input execution trace, the average I/O cost goes down drastically w.r.t. our previous implementation (from 5 minutes to 0.6s, for input/output sizes of about 5 Mb in the case of the `Webmail` specification).

7 Conclusion and Related Work

ABETS combines run-time assertion checking and automated (program and execution trace) transformations for improving the debugging of programs that are written in (Full) Maude, an expressive rule-based language that supports: 1) functional, concurrent, logic, and object-oriented computations; 2) rich type structures with sorts, subsorts and overloading; 3) equational rewriting and reasoning modulo axioms such as commutativity, associativity-commutativity, and associativity-commutativity-identity. As future work, we plan to augment the system capabilities by first extending its foundations to deal with *(folding variant) narrowing*, a recent addition to Maude that efficiently supports (built-in) semantic unification and symbolic reachability analysis of terms with logical variables, where suitable substitutions must be computed for the variables in both, the origin and the destination terms of reachability goals (Durán et al. 2016).

Assertions have been considered in (constraint) logic programming, functional programming, and functional-logic programming (see (Hermenegildo et al. 2012; Chitil 2011; Antoy and Hanus 2012) and references therein). However, we are not aware of any assertion-based, dynamic slicing system that is comparable to ABETS for either declarative or imperative languages. Actually, none of the correctness tools in the related literature integrate trace slicing and assertion-based reasoning to automatically identify, simplify, inspect, and repair faulty code and runs.

A detailed discussion of the literature related to this work can be found in (Alpuente et al. 2016; Alpuente et al. 2014). Here, we focus on assertion-checking tools supporting logical reasoning modulo axioms, which are the closest to our work. In (Durán et al. 2014), the validator tool mOdCL is described that checks OCL constraints on UML models encoded as Maude prototypes. If a constraint is violated, the execution is aborted and an error is reported that signals the state and the constraint involved. In contrast to ABETS, mOdCL does not simplify (either manually or automatically) the execution trace that reaches the erroneous state or the program itself in any way. The (rewriting logic) semantic framework $\mathbb{K}$ (Roşu 2015) supports assertion-based analysis and runtime verification based on Reachability Logic (RL), a particular class of first-order formulas with equality that consist of (Boolean) terms with logical variables and constraints over them. These formulas, called *patterns*, specify those concrete configurations matching the pattern algebraic structure and satisfying its constraints. Pattern formulas, written $\pi \wedge \phi$, are used to express (and reason about) static state properties, similarly to our system assertions $S\{\varphi\}$. As for our functional assertions $I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$, they slightly remind more sophisticated RL formulas $\mathscr{P} \Rightarrow \mathscr{P}'$, where $\mathscr{P}, \mathscr{P}'$ are patterns. However, the formula $\mathscr{P} \Rightarrow \mathscr{P}'$ specifies that any state satisfying $\mathscr{P}$ transits (in zero or more steps) into a state satisfying $\mathscr{P}'$; hence it is evaluated on system computations, in contrast to our functional assertions, which predicate on equational simplifications. While our formulas are quantifier-free and intended for assertion-based debugging of Maude computations, RL formulas are used in $\mathbb{K}$ for deductive and algorithmic verification. To the best of our knowledge, no $\mathbb{K}$ tool has been devised to support trace slicing and slicing-based program debugging and correction. For execution, debugging, and model checking, $\mathbb{K}$ semantics has been traditionally compiled into Maude. A textual debugger that works with different $\mathbb{K}$ backends is currently under development.

References

ALPUENTE, M., BALLIS, D., FRECHINA, F., AND ROMERO, D. 2014. Using Conditional Trace Slicing for improving Maude Programs. *SCP 80, Part B*, 385 – 415.

ALPUENTE, M., BALLIS, D., FRECHINA, F., AND SAPIÑA, J. 2016. Debugging Maude Programs via Runtime Assertion Checking and Trace Slicing. *JLAMP*. To appear.

ALPUENTE, M., ESCOBAR, S., ESPERT, J., AND MESEGUER, J. 2014. A Modular Order-Sorted Equational Generalization Algorithm. *Inf. and Comp. 235*, 98–136.

ANTOY, S. AND HANUS, M. 2012. Contracts and Specifications for Functional Logic Programming. In *PADL 2012*. LNCS, vol. 7149. Springer, 33–47.

BARRETT, C., CONWAY, C. L., DETERS, M., HADAREAN, L., JOVANOVIĆ, D., KING, T., REYNOLDS, A., AND TINELLI, C. 2011. CVC4. In *CAV 2011*. LNCS, vol. 6806. Springer, 171–177.

CHITIL, O. 2011. A Semantics for Lazy Assertions. In *PEPM 2011*. ACM, 141–150.

CLAVEL, M., DURÁN, F., EKER, S., LINCOLN, P., MARTÍ-OLIET, N., MESEGUER, J., AND TALCOTT, C. 2007. *All About Maude: A High-Performance Logical Framework*. Springer.

DURÁN, F., EKER, S., ESCOBAR, S., MARTÍ-OLIET, N., MESEGUER, J., AND TALCOTT, C. 2016. Built-in Variant Generation and Unification, and their Applications in Maude 2.7 (System Description). In *IJCAR 2016*. To appear.

DURÁN, F., ROLDÁN, M., MORENO-DELGADO, A., AND ÁLVAREZ, J. M. 2014. Dynamic Validation of Maude Prototypes of UML Models. In *SAS 2014 - Essays Dedicated to Kokichi Futatsugi*. LNCS, vol. 8373. Springer, 212–228.

FIELD, J. AND TIP, F. 1994. Dynamic Dependence in Term rewriting Systems and its Application to Program Slicing. In *PLILP 1994*. LNCS, vol. 844. Springer, 415–431.

HERMENEGILDO, M. V., BUENO, F., CARRO, M., LÓPEZ-GARCÍA, P., MERA, E., MORALES, J. F., AND PUEBLA, G. 2012. An overview of Ciao and its design philosophy. *TPLP 12,* 1-2, 219–252.

MESEGUER, J. 1992. Conditional Rewriting Logic as a Unified Model of Concurrency. *TCS 96,* 1, 73–155.

ROŞU, G. 2015. From Rewriting Logic, to Programming Language Semantics, to Program Verification. In *LRC 2015 - Festschrift Symposium in Honor of José Meseguer*. LNCS, vol. 9200. Springer, 598–616.

Appendix

This appendix summarizes the Maude (meta-level) operations that we developed in C++ to further optimize the execution of ABETS. The appendix is only intended to facilitate the review: a description of the operations can be found at `http://safe-tools.dsic.upv.es/maudev` and the code is publicly available in Mau-Dev as standard Maude meta-level operations.

`op metaMap : Module Term -> String .`
Given M and t, `metaMap` delivers a sophisticated string representation of the term t where each string character in the representation has a shortcut to the corresponding subterm of t. This is key for efficiently dealing with term slices, particularly in the presence of mixfix operators.

`op metaReducePath : Module Term Bool -> ITrace .`
Given a Maude module M, a term t and a Boolean expression b, `metaReducePath` delivers an instrumented trace of sort `ITrace` that contains the precise equational simplification sequence for t in M augmented with the computed substitutions and contexts. For the case when b is `true`, the applied membership axioms are also included in the trace.

`op metaAssociative : Module Term -> Bool .`
Given M and t, `metaAssociative` returns `true` if the topmost operator of t is associative or `false` otherwise. This operation dramatically improves the `isAssociative` function defined in Full Maude by directly inspecting a specific flag in the C++ term representation of t (while `isAssociative` matches t with the possibly large list of operator definitions in M.)

`op metaCommutative : Module Term -> Bool .`
It is the analogous of the `metaAssociative` operation for the `comm` attribute.

`op metaConstructor : Module Term -> Bool .`
Given M and t, `metaConstructor` returns `true` if the topmost operator of t is a constructor symbol of M that is identified by means of the `ctor` attribute.

`op metaIdentity : Module Term ~> Term? .`
Given M and t, `metaIdentity` delivers the identity element of the topmost symbol of t or `noIdentity` if no such term exists. Analogously, the associated right (resp. left) identity element can be obtained by means of the `metaRightIdentity` (resp. `metaLeftIdentity`) operation. For terms with symbols obeying both left and right identity, all three operations deliver the same result.

`op metaString : Module Term Bool -> String .`
Given M, t, and the Boolean expression b, `metaString` returns a term of sort `String` that provides the source-level (resp. meta-level) representation of t for the case when b is `true` (resp. `false`), which highly outspeeds its Maude counterpart.