Document downloaded from:

http://hdl.handle.net/10251/79350

This paper must be cited as:

Alpuente Frasnedo, M.; Frechina, F.; Sapiña Sanchis, J.; Ballis, D. (2016). Assertion-based Analysis via Slicing with ABETS. Theory and Practice of Logic Programming. 16(5):515-532. doi:10.1017/S1471068416000375.

The final publication is available at

http://dx.doi.org/10.1017/S1471068416000375

Copyright Cambridge University Press (CUP)

Additional Information

Assertion-based Analysis via Slicing with ABETS*

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 uses slicing to automatically create reduced versions of both a run's execution trace and executed program, reduced versions 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 generates suitable program fixes when a state invariant is violated. Finally, we provide an experimental evaluation that shows the performance and effectiveness of the system.

KEYWORDS: Runtime Assertion Checking, Dynamic Program and Trace Slicing, Program Diagnosis and Debugging, Rewriting Logic, Maude

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 ("Assertion-BasEd Trace Slicer"), 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 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 Maude's *equational matching* algorithm). While this

* 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.

maximizes efficiency and is certainly justified during the program operation, it further complicates debugging. The dynamic analyzer ABETS described in this paper facilitates the debugging of Maude programs. It does this by drastically simplifying the size and complexity of the analyzed programs and runs while still showing 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 user supplements the Maude program to be analyzed with a set of logical assertions that are checked at runtime. Upon an assertion failure, an accurate set of discordant positions (called symptoms) is *automatically* calculated by ABETS by comparing the *computed* erroneous program state with the *expected* pattern for the state (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 by ABETS 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 propagates 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 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 not just for Maude, but also for 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 the user needs not identify criteria or error symptoms 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 provided by the user.

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

- We explain the functionality of ABETS in Section 3. In Section 3.1, we describe the assertion-based trace slicing facility. In Section 3.2, we outline a new repair technique that automatically suggests program corrections to fix the program faults that are signalled by the violation of a state invariant property. The corrected rules are guarded by a suitable instance of the invariant so that the repaired rule is fired only if the invariant is fulfilled. In Section 3.3, we present some novel extra analysis features that complement the ABETS core functionality.
- We provide a description of those novel implementation details and optimizations that have boosted the system performance in Section 4. Also, we report a new in-depth experimental evaluation of the system in Section 5 that assesses critical aspects such as the assertion-

checking and slicing capabilities, and the system input/output performance, which is a usual weak spot of tools developed in (Full) Maude.

- The ABETS system is available at `http://safe-tools.dsic.upv.es/abets`. It can be downloaded and locally installed as a stand-alone console application, or it can be remotely used via a user-friendly web interface. A brief discussion of related tools and concluding remarks are provided in Section 6.

2 Modeling Concurrent Systems in Maude: Our Running 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 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 (a.k.a unity) (U) of function symbols, which cannot be handled by ordinary term rewriting but instead are handled implicitly by working with congruence classes of terms. 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, implicitly expressed as function attributes, that are only used for Ax-matching.

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 $t_0 \stackrel{r_0}{\longrightarrow}_E t_1 \stackrel{r_1}{\longrightarrow}_E \ldots$, where $t \stackrel{r}{\longrightarrow}_E t'$ denotes a transition (modulo E) from state t to t' via the rewrite rule of R that is uniquely labeled with r. Assuming that the initial term t is normalized (this assumption is not essential, but it will simplify the exposition), each single transition $t \stackrel{r}{\longrightarrow}_E t'$ (or *Maude step)* is computed as a rewrite chain $t \stackrel{r}{\longrightarrow} t'' \to^*_\Delta (t''_{\downarrow_\Delta}) = t'$, where $t'' \to^*_\Delta (t''_{\downarrow_\Delta})$ 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*. When a rewrite step from term t to term t' via a rule $r \in R$ must be fully characterized, we will write $t \stackrel{r,\sigma,w}{\longrightarrow} t'$ where w is the position in t where the rewrite occurred and σ is the computed substitution obtained by pattern matching modulo E. As usual, term positions are defined by means of sequences of natural numbers (Λ denotes the empty sequence, i.e., the root position). The result of *replacing the subterm* of t at position w by the term s is denoted by $t[s]_w$.

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 (price *limit*) at which traders want to buy stocks.

When the stock price equals or drops below the price limit `L`, the associated order is *opened* and the trader buys the stocks at the current stock price. An order is automatically *closed* and the

[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`.

associated stocks are sold either (a) when the stock price P exceeds the purchase price limit L plus a predetermined *profit target* PT (i.e., $P - L \geq PT$), or (b) when $L - P$ 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,closed),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 + P),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 + P),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 ST (which can be either `open` or `closed`). For simplicity, an order allows only 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 opens a trader order only if the stock price P falls below or is equal to the order price limit L. When

[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`.

the order is opened, the stock price is subtracted from the trader's capital, thereby updating the capital. 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 an order for the stock `SID` and removes it from the current state when the `SID` stock price `P` falls below or is equal to the `L` − `SL` stop loss threshold. The trader's capital then increases by the price `P` that the trader gets for the sold stocks. 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 non-positive 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,closed)) =>* 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 and Repair 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 $\bullet$.

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 more automatic support to the analysis of erroneous programs and traces.

The slicing algorithm employs *unification* to implement the origin-tracking procedure that properly tracks back the data dependences along the trace, and the *generalization* (i.e., anti-unification) algorithm *modulo* axioms of (Alpuente et al. 2014) to automatically identify semantic disagreements of the program behavior w.r.t. the assertions (Alpuente et al. 2016).

[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.

ABETS is also provided with an automatic program repair facility, which is described in Section 3.2, that suggests fixes to potentially buggy rewrite rules whenever it 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 causes (a piece of) the transformed state to match the state pattern S). This fix is done by adding a constrained instance of the logic formula φ into the conditional part of the rule, which is computed by using Maude's built-in E-unification (Durán et al. 2016).

3.1 Assertion-based Slicing in ABETS

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 $\varphi_1 \wedge \ldots \wedge \varphi_n$.

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. When a system state s does not satisfy a system assertion $S\{\varphi\}$, the position p in s, which is called *bug* position, precisely indicates the subterm of s that matches S and is responsible for the assertion violation.

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:

```
R:Nat : SS:Set{Stock} | tr(TID:TraderID,C:Int),TS:Set{Trader} | OS:Set{Order}
                        {ordinary(tr(TID:TraderID,C:Int)) implies C:Int >= 0}
```

where the user-specified predicate `ordinary(T)` simply checks whether `T` is a non-premium trader in the Maude program of Example 2.1.

ii) *Functional* assertions: Their general form is $I\,\{\varphi_{in}\} \rightarrow O\,\{\varphi_{out}\}$ where I, O are terms, and $\varphi_{in}, \varphi_{out}$ are logic formulas. Intuitively, functional assertions specify pre- and post-conditions over the equational simplification $t \rightarrow^*_\Delta (t_{\downarrow_\Delta})$ 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` should generate positive stock prices provided that the input stock prices are also positive.

The satisfiability of the provided assertions can be checked in two different modalities, 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 $\mathcal{T} = s_0 \stackrel{\texttt{next-rnd}}{\longrightarrow} s_1 \stackrel{\texttt{open-ord}}{\longrightarrow} s_2$ 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,closed)`

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 violation 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,•,•,closed))
next-rnd
   •→          • : •,st(•,12)) | tr('T1,9),• | ord(•,'T1,•,12,•,•,closed)
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.

3.2 Automatic Repair of Program Rules in ABETS

Given an equational theory $E = \Delta \cup Ax$ and two terms t_1 and t_2, an E-unifier for t_1 and t_2 is a substitution σ such that $t_1\sigma =_E t_2\sigma$. In Maude, E-unifiers are not represented as a single substitution, but as a pair of substitutions (σ_1, σ_2), one for left unificands and the other for right unificands (i.e., $t_1\sigma_1 =_E t_2\sigma_2$). Also, Maude's E-unification algorithm may generate new (fresh) unification variables, denoted by `%n`, with `n` being a natural number. The set of all such variables contained in a given term t is denoted by $\mathit{UnifVar}(t)$. Let us see an example.

Example 3.4
Consider a simple Maude program whose signature consists of two unary operators, `m` and `c`, and one commutative, binary operator `f`. The program includes a single equation `m(X)` $=$ `c(X)`. Then, $\sigma = (\sigma_1, \sigma_2) = (\{\texttt{X/\%1}\}, \{\texttt{Z/\%1}\})$ is an E-unifier for the terms $\texttt{t}_1 =$ `f(m(X),0)` and $t_2 =$

Trace information (trusted mode)

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
...			
34	rnd-delta	st('S1, 10),if random(21) rem 10 rem 2 == 0 then st('S2, 7 + random(21) rem 10),updP(3, 7 + 1, empty) else st('S2, 7 + - rndDelta(21)),updP(3, 7 + 1, empty) fi	•,if random(21) rem 10 rem 2 == 0 then • else st(•, 7 + - rndDelta(21)),• fi
35	rnd-delta	st('S1, 10),if random(21) rem 10 rem 2 == 0 then st('S2, 7 + random(21) rem 10),updP(3, 7 + 1, empty) else st('S2, 7 + - (random(21) rem 10)),updP(3, 7 + 1, empty) fi	• **,if random(21) rem 10 rem 2 == 0 then** • **else st(** • **, 7 + - (random(21) rem 10)),** • **fi**
36	builtIn	st('S1, 10),if 3488238119 rem 10 rem 2 == 0 then st('S2, 7 + random(21) rem 10),updP(3, 7 + 1, empty) else st('S2, 7 + - (random(21) rem 10)),updP(3, 7 + 1, empty) fi	•,if 3488238119 rem 10 rem 2 == 0 then • else st(•, 7 + - (random(21) rem 10)),• fi
...			
50	builtIn	st('S1, 10),st('S2, -2),updP(3, 8, empty)	•,st(•, -2),•
51	updP-owise	st('S1, 10),st('S2, -2),(empty).Set{Stock}	•,st(•, -2),•
52	toBnf	st('S1, 10),st('S2, -2)	•,st(•, -2)
Total size:		4340 bytes	279 bytes
Reduction Rate: 94%			

Fig. 3. Extended view of the computed trace slice after refuting the functional assertion of Example 3.2 (trusted mode).

f(0,c(Z)). The new, unification variable %1 is used to establish that X and Z represent the same value, and it is the only common variable shared by $t_1\sigma_1$ and $t_2\sigma_2$.

Our repair technique is based on a two-phase algorithm that takes as input: (i) the last Maude step $t \xrightarrow{\mathtt{r},\sigma,w} t' \to^*_\Delta t'_{\downarrow\Delta}$ of the execution trace that violates $S\{\varphi\}$, with r being λ => ρ if C, (ii) the violated system assertion $S\{\varphi\}$, and iii) the bug position p in the last trace state $t'_{\downarrow\Delta}$.

Phase 1 [Semantic unification of the failing assertion and rule]. First we E-unify the terms $t'[\rho]_w$ (that is, a more general version of $t' = t[\rho\sigma]_w$ that does not apply the substitution σ to the reduced term) and $t'_{\downarrow\Delta}[S]_p$ (that is, a more general version of $t'_{\downarrow\Delta}$ where the subterm at the bug position p is replaced by the assertion pattern S itself) in order to relate the variables in the right-hand side ρ of r with the variables that appear in the state template S. Since there may be several E-unifiers, we just select an E-unifier (σ_ρ, σ_S) such that the bindings in σ_ρ do not clash with the bindings in the computed substitution σ. This is done by performing a standard consistency check through the parallel composition of σ_ρ and σ, which computes the most general unifier (mgu) of the set of all the equations $x = t$ that represent a binding x/t in either σ_ρ or σ. If such an mgu exists, σ_ρ is consistent w.r.t. σ, and the corresponding E-unifier (σ_ρ, σ_S) is selected.

As a side note, observe that we cannot simply E-unify ρ with S because the state template S could include operators that are not in ρ but in t', and, hence, the two terms could be not E-unifiable and lead to no repair. This is the reason why we need to E-unify ρ and S within their corresponding state contexts, that is, $t'[\rho]_w$ and $t'_{\downarrow\Delta}[S]_p$.

Example 3.5

Consider a Maude program that contains the rewrite rule rl [r] f(X) => g(X) and no equations, together with the execution trace a & f(0) $\xrightarrow{\mathtt{r}}$ a & g(0) and the system assertion (a & g(Z)) {Z>0}, which is violated in the state a & g(0). Observe that there is no E-unifier between the right-hand side g(X) of r and the state template a & g(Z), whereas the pair ({X/%1},{Z/%1}) is an E-unifier for the terms a & g(X) and a & g(Z), which include g(X) and a & g(Z) in their corresponding state context. More importantly, the bindings in the computed E-unifier en-

force X and Z to bind the very same value. This suggests to us that we can achieve a repair by forcing the rewrite rule argument X to inherit the constraints on Z.

Phase 2 [Strengthening the rule condition]. Given the computed E-unifier (σ_ρ, σ_S), first we split σ_ρ into two sets σ_{rule} and σ_{new} such that $\sigma_{rule} = \{x/t \in \sigma_\rho \mid x \in Var(\rho) \wedge UnifVar(t) = \emptyset\}$, and $\sigma_{new} = \sigma_\rho \setminus \sigma_{rule}$. Note that σ_{new} contains all those σ_ρ bindings that introduce new unification variables, while the bindings of σ_{rule} only use the original variables of ρ. Then, we replace the faulty rule r with the following corrected rule whose condition is strengthened by adding a constrained version (that is built by using σ_S and σ_{rule}) of the violated logic formula φ:

$$\texttt{crl [rfix] : } \lambda\sigma_{new} \texttt{ => } \rho\sigma_{new} \texttt{ if C}\sigma_{new} \texttt{ /\textbackslash}\Big(\big(\bigwedge_{x/t\in\sigma_{rule}} x\texttt{==}t\big) \texttt{ implies } \varphi\sigma_S\Big).$$

The corrected rule rfix is produced by instantiating the original rule r with the substitution σ_{new} that introduces in rfix the fresh variables generated during the unification process of Phase 1 and by adding the instance $\varphi\sigma_S$ of the falsified logical formula φ. The variables of such an instance are constrained via a logical implication whose premise is the conjunction of all the bindings x/t in σ_{rule} interpreted as Boolean expressions $x\texttt{==}t$[4] . In the case when σ_{rule} is empty, the logical implication corresponds to (true implies $\varphi\sigma_S$), and thus simply reduces to the term $\varphi\sigma_S$.

Example 3.6
Consider a Maude program that includes the following rewrite rule r and equation e

```
crl [r] : f(X,Y) => c(2,g(X,Y)) if X =/= Y .
eq  [e] : g(X,Y) = m(X,Y) .
```

and assume that the operator m is declared commutative. Let us consider the system assertion c(2,m(Z,5)) {even(Z)}, where even(Z) checks if Z is an even natural number.

The execution trace $\texttt{f(5,3)} \xrightarrow{\texttt{r},\sigma} \texttt{c(2,g(5,3))} \xrightarrow{\texttt{e}} \texttt{c(2,m(5,3))}$, with computed substitution $\sigma = \{\texttt{X/5}, \texttt{Y/3}\}$, is erroneous since the formula even(Z) does not hold for the binding Z/3 that is computed by matching modulo *commutativity* the state c(2,m(5,3)) in the assertion state template c(2,m(Z,5)).

The repair proceeds by first performing Phase 1, which computes two E-unifiers of the terms c(2,g(X,Y)) and c(2,m(Z,5)), namely,

$$(\sigma_{\rho_1}, \sigma_{S_1}) = (\{\texttt{X/\%1}, \texttt{Y/5}\}, \{\texttt{Z/\%1}\}) \qquad (\sigma_{\rho_2}, \sigma_{S_2}) = (\{\texttt{X/5}, \texttt{Y/\%1}\}, \{\texttt{Z/\%1}\})$$

Now, observe that the E-unifier $(\sigma_{\rho_1}, \sigma_{S_1})$ is discarded since σ_{ρ_1} is not consistent w.r.t. σ. Actually, there is no mgu of σ_{ρ_1} and σ because of the clash between the bindings $\texttt{Y/5} \in \sigma_{\rho_1}$ and $\texttt{Y/3} \in \sigma$. The E-unifier $(\sigma_{\rho_2}, \sigma_{S_2})$ is consistent w.r.t. σ and thus is used to infer the repair in Phase 2 of the algorithm.

Phase 2 generates the partition $\sigma_{\rho_2} = \sigma_{rule} \cup \sigma_{new} = \{\texttt{X/5}\} \cup \{\texttt{Y/\%1}\}$ and uses it together with σ_{S_2} to yield the following corrected version of the rule r:

```
crl [rfix] : f(X,%1) => c(2,g(X,%1)) if (X =/= %1 /\ (X == 5 implies even(%1)) .
```

Note that the generated condition of a repaired rule rfix might not be satisfiable, which makes rfix not applicable. This is not bad since the non-applicability of the corrected rule prevents the system from reaching the faulty state signaled by the assertion violation. This therefore has the

[4] A binding x/t in σ_{rule} can always be interpreted as an executable, Boolean expression $x\texttt{==}t$, since all the variables included in x/t appear in the rewrite rule as well and thus take concrete values when the rule is applied.

inherent effect of reducing the number of erroneous runs in the system, which is of primary importance in the repair of critical systems as first advocated by (Logozzo and Ball 2012).

3.3 New Additional Analysis Features

The system functionality of ABETS has been extended by introducing the following, new additional features.

Trusted/Untrusted modes. ABETS encompasses two slicing modes: trusted and untrusted. In trusted mode, Maude built-in operators are considered to be trusted (i.e., not to have bugs) and are therefore ignored in the trace slice (See Figure 3), which further reduces its size. In untrusted mode, all relevant operators are traced. The trusted mode is set to true by default and can be switched to untrusted mode by choosing the `Trace Information` option in the main menu and then clicking the `Trusted/Untrusted mode` button. To help the user compare the original, extended trace and the trace slice when they are shown side-by-side (e.g., in the table view), trusted reduction steps (as well as duplicate states *modulo axioms*) are not omitted but are depicted in light gray.

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 novel graph representation of the state space that can improve user's understanding of the program behavior (see Figure 4). It is possible to switch between the two representations by left-clicking on any node of the tree or graph. In the case when the user left-clicks on a node in the graph, the topmost leftmost node in the tree that is associated with the considered graph node is highlighted.

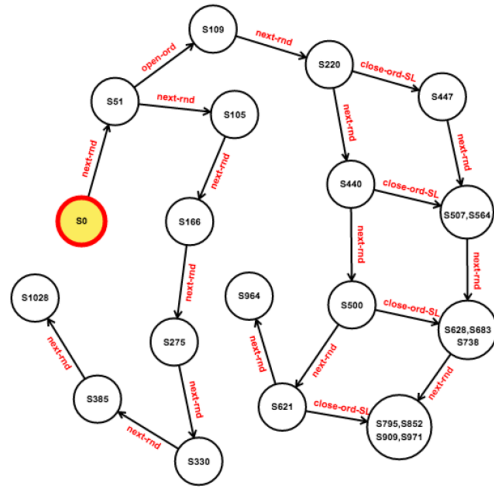

Fig. 4. Computation graph generated from initial state s_0 of Example 3.3 (partial view).

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 5). This query is a filtering pattern with wildcards that define irrelevant terms by means of the underscore character (_) and relevant terms 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.

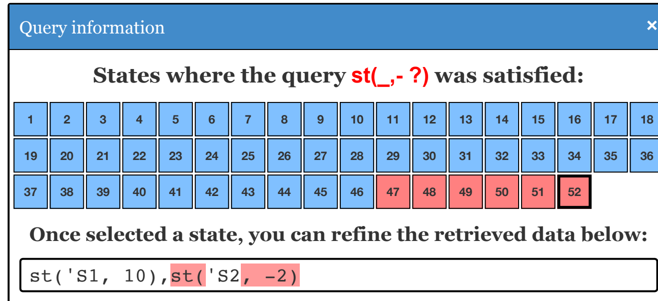

Fig. 5. Result of the trace query `st(_, - ?)`.

Several extra features, described in (Alpuente et al. 2016), are: (i) an *incremental trace slicing* capability that allows the computed trace slices to be further simplified by automatically applying backward as well as forward trace slicing w.r.t. user-provided slicing criteria refinements (Alpuente et al. 2015); (ii) a *program slicing* feature for delivering program fragments that include all and only the rules/equations responsible for the detected error.

A starting guide that contains a typical analysis session with ABETS can be found at `http://safe-tools.dsic.upv.es/abets/quickstart.pdf`.

4 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 (iv) 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 (available at `http://safe-tools.dsic.upv.es/maudev`) 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 available at Mau-Dev's website.

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 a devoted module that defines unique *placeholder* terms that are subsequently reduced to the actual

arguments of the initial (Full Maude) call. For example, to encode a Full-Maude, source-level representation of the state s_2 of Example 3.3, ABETS defines the 0-ary operator aState:

```
op aState : -> String .
eq aState = "1 : (st('S1,23), st('S2,8)) | (tr('T1,10), tr('T2,20)) |
 (ord('O1,'T2,'S1,10,6,4,open), ord('O2,'T1,'S2,12,4,3,close))" .
```

This greatly reduces the size of the initial Full-Maude call since it only contains the aState placeholder but not the actual state data. These data are later brought back by applying the aState equation. A similar encoding is used for user-defined assertions and execution traces that are to be asynchronously checked.

The added module is loaded prior to starting the Full Maude's *execution loop* (Clavel et al. 2007). 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 trace slice, the associated program slice that can be computed as described in (Alpuente et al. 2016), 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. Since efficient output handling is crucial not to penalize the overall performance of the system, (meta) string conversion has also been implemented in C++.

Some experiments that highlight the efficiency gain of the optimized system w.r.t. (Alpuente et al. 2016) are shown in Section 5.

5 Experimental evaluation

To evaluate the performance of the ABETS system, we introduced defects in several Maude programs endowed with assertions and we 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 running 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 email management, system administration capabilities, login/logout functionality, etc. 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 evaluate both the effectiveness and the 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

Table 1. *Synchronous assertion-checking performance analysis*

Program	T_{Ex}	T_{ExChk}	*#Chk*	*OV*	OV_{jlamp}	T_{synth}	T_{fix}	$T_{I/O}$	*Size* $\mathcal{T}_\varepsilon$	*Size* $\mathcal{T}_\varepsilon^\bullet$	*%Red.*
Bank Model	17	101	2004	4.94	5.76	2	2	10	9.536	1.236	87%
Blocks World	19	37	509	0.95	2.16	1	1	2	0.279	0.046	84%
BRP	5	23	1002	3.6	4.6	1	2	9	0.792	0.269	67%
Dekker	40	98	1002	1.45	2.5	2	14	55	8.268	0.286	97%
Maze	128	409	7437	2.2	3	1	3	13	2.747	0.423	85%
Philosophers	12	36	811	2	2.92	1	3	47	5.244	1.990	62%
Rent-a-car (fm)	178	263	1503	0.48	0.52	5	9	247	5.507	0.115	98%
Stock Ex.	36	103	1503	1.86	2.58	3	12	263	46.423	4.153	91%
Stock Ex. (fm)	726	1310	2004	0.8	1.72	5	43	4688	195.397	20.862	89%
Webmail app	138	271	1002	0.96	1.99	9	20	541	133.460	7.823	94%

simplified into a trace slice $\mathcal{T}_\varepsilon^\bullet$ w.r.t. slicing criteria that are automatically inferred. The experiments were conducted on a PC with 3.3GHz Intel Xeon E5-1660 CPU with 64GB RAM.

Obviously, the slowdown of the entire 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 quickly. The slowdown can also be affected by the complexity of the predicates involved in the functional and system assertions to be checked.

Table 1 summarizes our results. The T_{Ex} and T_{ExChk} columns measure the execution times (in ms) with and without assertion checking for traces that apply 500 rewrite rules (which expands to 8292 rewrites —i.e., rule, equation, built-in operator, and axiom applications— on average). *#Chk* represents the total number of assertion checks performed when assertion checking was enabled. *OV* is the overhead, i.e., the ratio $= (T_{ExChk} - T_{Ex})/T_{Ex}$ which indicates the relative slowdown due to assertion-checking. The results obtained are quite satisfactory and comparable with similar logic assertion checking frameworks such as (Mera et al. 2009). The average overhead is 1.92, which is 69% of the average value (2.78) of the overhead of (Alpuente et al. 2016) that are shown in column OV_{jlamp} for the very same benchmark programs.

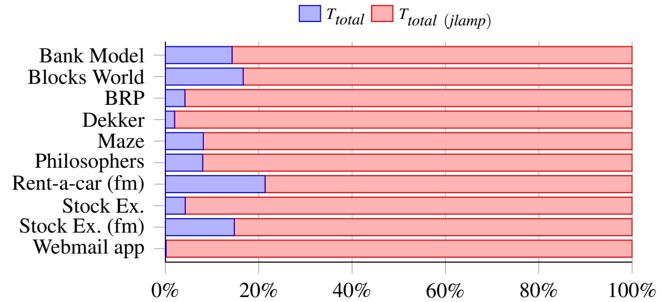

Fig. 6. Total speedup with respect to (Alpuente et al. 2016).

The figures in the T_{synth} and T_{fix} columns respectively measure the times for synthesizing the slicing criterion and for inferring the repairs (in ms). Our experiments 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 2.5Kb, which is 20 times the size of the Stock Ex. states). The time for inferring the repairs is also a small portion of the total execution time.

The trace slices that are automatically delivered by ABETS are evalutated by comparing the size of the detected erroneous execution trace $\mathcal{T}_\varepsilon$ (in kilobytes); the size of the sliced execution

trace $\mathcal{T}_{\varepsilon}^{\bullet}$ (in kilobytes); and the derived *reduction* rate achieved (*%Red.*), which ranges from 98% to 62% with an average reduction rate of 85%. With regard to the time required to perform the slicing, our implementation is quite time efficient despite the complex analyses and reasoning modulo axioms performed underneath; the elapsed times are small even for very complex traces and also scale linearly. For example, running the slicer for a typical 50Kb faulty trace delivered by the analyzer

Finally, the generation, parsing, and input/output of traces (and trace slices) have been greatly improved in the current version of ABETS. The input/output (I/O) times are shown in column $T_{I/O}$ of Table 1 (in ms) for I/O data sizes that range from 15 Kb (in the case of the `Blocks` program) to 7 Mb (in the case of the `Stock Ex.(fm)` program). This gives an average I/O cost of 0.6 s, whereas in our previous tool the I/O operations took minutes.

The total speedups that we achieved w.r.t. our previous implementation (including checking, slicing, and I/O costs) are represented in Figure 6, with an average speedup of 9.66 with respect to (Alpuente et al. 2016).

6 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.

Assertions have been considered in (constraint) logic programming, functional programming, and functional-logic programming (see (Mera et al. 2009; 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 assertions 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 $\mathscr{P} \Rightarrow \mathscr{P}'$, where $\mathscr{P}$ (and $\mathscr{P}'$) consists of a (Boolean) term b and a constraint φ over the logical variables of b (i.e., $b \wedge \varphi$). These formulas $\mathscr{P}$ specify those concrete configurations that match the algebraic structure of b and satisfy the constraint φ. They 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 are quantifier-free and evaluated on equational simplifications, while RL formulas assert more general properties on system computations and are used for deductive and algorithmic verification.

A different semantic approach for automatic program repair that is based on abstract interpretation can be found in (Logozzo and Ball 2012), which applies to .Net languages.

References

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

ALPUENTE, M., BALLIS, D., FRECHINA, F., AND SAPIÑA, J. 2015. Exploring Conditional Rewriting Logic Computations. *Journal of Symbolic Computation 69*, 3–39.

ALPUENTE, M., BALLIS, D., FRECHINA, F., AND SAPIÑA, J. 2016. Debugging Maude Programs via Runtime Assertion Checking and Trace Slicing. *Journal of Logical and Algebraic Methods in Programming*. To appear.

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

ANTOY, S. AND HANUS, M. 2012. Contracts and Specifications for Functional Logic Programming. In *Proc. of the 14th Int'l Symposium on Practical Aspects of Declarative Languages (PADL 2012)*. Lecture Notes in Computer Science, vol. 7149. Springer-Verlag, 33–47.

CHITIL, O. 2011. A Semantics for Lazy Assertions. In *Proc. of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2011)*. Association for Computing Machinery, 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-Verlag.

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. In *Proc. of the 8th International Joint Conference on Automated Reasoning (IJCAR 2016)*. Lecture Notes in Computer Science, vol. 9706. Springer-Verlag, 183–192.

DURÁN, F., ROLDÁN, M., MORENO-DELGADO, A., AND ÁLVAREZ, J. M. 2014. Dynamic Validation of Maude Prototypes of UML Models. In *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi (SAS 2014)*. Lecture Notes in Computer Science, vol. 8373. Springer-Verlag, 212–228.

FIELD, J. AND TIP, F. 1994. Dynamic Dependence in Term rewriting Systems and its Application to Program Slicing. In *Proc. of the 6th Int'l Symp. on Programming Language Implementation and Logic Programming (PLILP 1994)*. Lecture Notes in Computer Science, vol. 844. Springer-Verlag, 415–431.

LOGOZZO, F. AND BALL, T. 2012. Modular and Verified Automatic Program Repair. In *Proc. of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2012)*. Association for Computing Machinery, 133–146.

MERA, E., LÓPEZ-GARCÍA, P., AND HERMENEGILDO, M. V. 2009. Integrating software testing and run-time checking in an assertion verification framework. In *Proc. of the 25th Int'l Conference on Logic Programming (ICLP 2009)*. Lecture Notes in Computer Science, vol. 5649. Springer-Verlag, 281–295.

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

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