Declarative Debugging of Rewriting Logic Specifications*

A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet
Facultad de Informática, Universidad Complutense de Madrid, Spain
`ariesco@fdi.ucm.es, {alberto, rafa, narciso}@sip.ucm.es`

Abstract. Declarative debugging is a semi-automatic technique that starts from an incorrect computation and locates a program fragment responsible for the error by building a tree representing this computation and guiding the user through it to find the wrong statement. This paper presents the fundamentals for the declarative debugging of rewriting logic specifications, realized in the Maude language, where a wrong computation can be a reduction, a type inference, or a rewrite. We define appropriate debugging trees obtained as the result of collapsing in proof trees all those nodes whose correctness does not need any justification. Since these trees are obtained from a suitable semantic calculus, the correctness and completeness of the debugging technique can be formally proved. We illustrate how to use the debugger by means of an example and succinctly describe its implementation in Maude itself thanks to its reflective and metalanguage features.

1 Introduction

In this paper we present a declarative debugger for *Maude specifications*, including equational functional specifications and concurrent systems specifications. Maude [10] is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Maude modules correspond to specifications in *rewriting logic* [14], a simple and expressive logic which allows the representation of many models of concurrent and distributed systems. This logic is an extension of equational logic; in particular, Maude *functional modules* correspond to specifications in *membership equational logic* [1, 15], which, in addition to equations, allows the statement of *membership axioms* characterizing the elements of a sort. In this way, Maude makes possible the faithful specification of data types (like sorted lists or search trees) whose data are not only defined by means of constructors, but also by the satisfaction of additional properties. Rewriting logic extends membership equational logic by adding rewrite rules, that represent transitions in a concurrent system. Maude *system modules* are used to define specifications in this logic.

* Research supported by MEC Spanish projects *DESAFIOS* (TIN2006-15660-C02-01) and *MERIT-FORMS* (TIN2005-09027-C03-03), and Comunidad de Madrid program *PROMESAS* (S-0505/TIC/0407).

The Maude system supports several approaches for debugging Maude programs: tracing, term coloring, and using an internal debugger [10, Chap. 22]. The tracing facilities allow us to follow the execution of a specification, that is, the sequence of applications of statements that take place. The same ideas have been applied to the functional paradigm by the tracer *Hat* [9], where a graph constructed by graph rewriting is proposed as suitable trace structure. Term coloring consists in printing with different colors the operators used to build a term that does not fully reduce. The Maude debugger allows to define break points in the execution by selecting some operators or statements. When a break point is found the debugger is entered. There, we can see the current term and execute the next rewrite with tracing turned on. The Maude debugger has as a disadvantage that, since it is based on the trace, it shows to the user every small step obtained by using a single statement. Thus the user can lose the general view of the *proof* of the incorrect inference that produced the wrong result. That is, when the user detects an unexpected statement application it is difficult to know where the incorrect inference started. Here we present a different approach based on declarative debugging that solves this problem for Maude specifications.

Declarative debugging, also known as algorithmic debugging, was first introduced by E. Y. Shapiro [23]. It has been widely employed in the logic [12, 16, 25], functional [19, 18, 20], multi-paradigm [7, 3, 13], and object-oriented [4] programming languages. Declarative debugging starts from a computation considered incorrect by the user (error symptom) and locates a program fragment responsible for the error. The declarative debugging scheme [17] uses a *debugging tree* as logical representation of the computation. Each node in the tree represents the result of a computation step, which must follow from the results of its child nodes by some logical inference. Diagnosis proceeds by traversing the debugging tree, asking questions to an external oracle (generally the user) until a so-called *buggy node* is found. A buggy node is a node containing an erroneous result, but whose children have all correct results. Hence, a buggy node has produced an erroneous output from correct inputs and corresponds to an erroneous fragment of code, which is pointed out as an error. From an explanatory point of view, declarative debugging can be described as consisting of two stages, namely the debugging tree *generation* and its *navigation* following some suitable strategy [24].

The application of declarative debugging to Maude functional modules was already studied in our previous papers [5, 6]. The executability requirements of Maude functional modules mean that they are assumed to be confluent, terminating, and sort-decreasing[1] [10]. These requirements are assumed in the form of the questions appearing in the debugging tree. In this paper, we considerably extend that work by also considering system modules. Now, since the specifications described in this kind of modules can be non-terminating and non-confluent, their handling must be quite different.

[1] All these requirements must be understood *modulo* some axioms such as associativity and commutativity that are associated to some binary operations.

The debugging process starts with an incorrect computation from the initial term to an unexpected one. The debugger then builds an appropriate debugging tree which is an abbreviation of the corresponding proof tree obtained by applying the inference rules of membership equational logic and rewriting logic. The abbreviation consists in collapsing those nodes whose correctness does not need any justification, such as those related with transitivity or congruence. Since the questions are located in the debugging tree, the answers allow the debugger to discard a subset of the questions, leading and shortening the debugging process. In the case of functional modules, the questions have the form "Is it correct that T fully reduces to T'?", which in general are easier to answer. However, in the absence of confluence and termination, these questions do not make sense; thus, in the case of system modules, we have decided to develop two different trees whose nodes produce questions of the form "Is it correct that T is rewritten to T'?" where the difference consists in the number of steps involved in the rewrite. While one of the trees refers only to one-step rewrites, which are often easier to answer, the other one can also refer to many-steps rewrites that, although may be harder to answer, in general discard a bigger subset of nodes. The user, depending on the debugged specification or his "ability" to answer questions involving several rewrite steps, can choose between these two kinds of trees.

Moreover, exploiting the fact that rewriting logic is *reflective* [11], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [10, Chap. 14], a feature that makes Maude remarkably extensible and powerful, and that allows many advanced metaprogramming and metalanguage applications. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. Therefore, we are able to generate and navigate the debugging tree of a Maude computation using operations in Maude itself. In addition, the Maude system provides another module, `LOOP-MODE` [10, Chap. 17], which can be used to specify input/output interactions with the user. However, instead of using this module directly, we extend Full Maude [10, Chap. 18], that includes features for parsing, evaluating, and pretty-printing terms, improving the input/output interaction. Moreover, Full Maude allows the specification of concurrent object-oriented systems, that can also be debugged. Thus, our declarative debugger, including its user interactions, is implemented in Maude itself.

The rest of the paper is structured as follows. Sect. 2 provides a summary of the main concepts of both membership equational logic and rewriting logic, and how their specifications are realized in Maude functional and system modules, respectively. Sect. 3 describes the theoretical foundations of the debugging trees for inferences in both logics. Sect. 4 shows how to use the debugger by means of an example, while Sect. 5 comments some aspects of the Maude implementation. Finally, Sect. 6 concludes and mentions some future work.

Detailed proofs of the results, additional examples, and much more information about the implementation can be found in the technical report [21], which, together with the Maude source files for the debugger, is available from the webpage `http://maude.sip.ucm.es/debugging`.

2 Rewriting Logic and Maude

As mentioned in the introduction, Maude modules are executable rewriting logic specifications. Rewriting logic [14] is a logic of change very suitable for the specification of concurrent systems that is parameterized by an underlying equational logic, for which Maude uses membership equational logic (*MEL*) [1, 15], which, in addition to equations, allows the statement of membership axioms characterizing the elements of a sort.

2.1 Membership Equational Logic

A *signature* in *MEL* is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{k_1 \dots k_n, k}\}_{(k_1 \dots k_n, k) \in K^* \times K}$ a many-kinded signature, and $S = \{S_k\}_{k \in K}$ a pairwise disjoint K-kinded family of sets of *sorts*. The kind of a sort s is denoted by $[s]$. We write $T_{\Sigma,k}$ and $T_{\Sigma,k}(X)$ to denote respectively the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \dots, x_n : k_n\}$ is a set of K-kinded variables. Intuitively, terms with a kind but without a sort represent undefined or error elements.

The atomic formulas of *MEL* are either *equations* $t = t'$, where t and t' are Σ-terms of the same kind, or *membership axioms* of the form $t : s$, where the term t has kind k and $s \in S_k$. *Sentences* are universally-quantified Horn clauses of the form $(\forall X)\, A_0 \Leftarrow A_1 \wedge \dots \wedge A_n$, where each A_i is either an equation or a membership axiom, and X is a set of K-kinded variables containing all the variables in the A_i. A *specification* is a pair (Σ, E), where E is a set of sentences in *MEL* over the signature Σ.

Models of *MEL* specifications are Σ*-algebras* $\mathcal{A}$ consisting of a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times \dots \times A_{k_n} \longrightarrow A_k$ for each operator $f \in \Sigma_{k_1 \dots k_n, k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$. The meaning $[\![t]\!]_{\mathcal{A}}$ of a term t in an algebra $\mathcal{A}$ is inductively defined as usual. Then, an algebra $\mathcal{A}$ satisfies an equation $t = t'$ (or the equation holds in the algebra), denoted $\mathcal{A} \models t = t'$, when both terms have the same meaning: $[\![t]\!]_{\mathcal{A}} = [\![t']\!]_{\mathcal{A}}$. In the same way, satisfaction of a membership is defined as: $\mathcal{A} \models t : s$ when $[\![t]\!]_{\mathcal{A}} \in A_s$.

A *MEL* specification (Σ, E) has an initial model $\mathcal{T}_{\Sigma/E}$ whose elements are E-equivalence classes of terms $[t]$. We refer to [1, 15] for a detailed presentation of (Σ, E)-algebras, sound and complete deduction rules (that we adapt to our purposes in Fig. 1 in Sect. 3.1), as well as the construction of initial and free algebras. Since the *MEL* specifications that we consider are assumed to satisfy the executability requirements of confluence, termination, and sort-decreasingness, their equations $t = t'$ can be oriented from left to right, $t \to t'$. Such a statement holds in an algebra, denoted $\mathcal{A} \models t \to t'$, exactly when $\mathcal{A} \models t = t'$, i.e., when $[\![t]\!]_{\mathcal{A}} = [\![t']\!]_{\mathcal{A}}$. Moreover, under those assumptions an equational condition $u = v$ in a conditional equation can be checked by finding a common term t such that $u \to t$ and $v \to t$. The notation we will use in the inference rules studied in Sect. 3 for this situation is $u \downarrow v$.

2.2 Rewriting Logic

Rewriting logic extends equational logic by introducing the notion of *rewrites* corresponding to transitions between states; that is, while equations are interpreted as equalities and therefore they are symmetric, rewrites denote changes which can be irreversible. A rewriting logic specification, or *rewrite theory*, has the form $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational specification and R is a set of *rules* as described below. From this definition, one can see that rewriting logic is built on top of equational logic, so that rewriting logic is parameterized with respect to the version of the underlying equational logic; in our case, Maude uses *MEL*, as described in the previous section. A rule in R has the general conditional form[2]

$$(\forall X)\ t \Rightarrow t' \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w'_k$$

where the head is a rewrite and the conditions can be equations, memberships, and rewrites; both sides of a rewrite must have the same kind. From these rewrite rules, one can deduce rewrites of the form $t \Rightarrow t'$ by means of general deduction rules introduced in [14, 2], that we have adapted to our purposes.

Models of rewrite theories are called $\mathcal{R}$-*systems* in [14]. Such systems are defined as categories that possess a (Σ, E)-algebra structure, together with a natural transformation for each rule in the set R. More intuitively, the idea is that we have a (Σ, E)-algebra, as described in Sect. 2.1, with transitions between the elements in each set A_k; moreover, these transitions must satisfy several additional requirements, including that there are identity transitions for each element, that transitions can be sequentially composed, that the operations in the signature Σ are also appropriately defined for the transitions, and that we have enough transitions corresponding to the rules in R. Then, if we keep in this context the notation $\mathcal{A}$ to denote an $\mathcal{R}$-system, a rewrite $t \Rightarrow t'$ is satisfied by $\mathcal{A}$, denoted $\mathcal{A} \models t \Rightarrow t'$, when there is a transition $[\![t]\!]_{\mathcal{A}} \Rightarrow_{\mathcal{A}} [\![t']\!]_{\mathcal{A}}$ in the system between the corresponding meanings of both sides of the rewrite, where $\Rightarrow_{\mathcal{A}}$ will be our notation for such transitions. The rewriting logic deduction rules introduced in [14] are sound and complete with respect to this notion of model. Moreover, they can be used to build initial and free models; see [14] for details.

2.3 Maude Modules

Maude functional modules [10, Chap. 4], introduced with syntax `fmod ... endfm`, are executable membership equational specifications and their semantics is given by the corresponding initial membership algebra in the class of algebras satisfying the specification. In a functional module we can declare sorts

[2] Note that we use the notation $\Rightarrow$ for rewrites (as in Maude) and $\rightarrow$ for *oriented* equations and reductions using such equations. Other papers on rewriting logic use instead the notation $\rightarrow$ for rewrites.

(by means of keyword `sort(s)`); subsort relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`).

Maude system modules [10, Chap. 6], introduced with syntax `mod ... endm`, are executable rewrite theories and their semantics is given by the initial system in the class of systems corresponding to the rewrite theory. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`).

The executability requirements for equations and memberships are confluence, termination, and sort-decreasingness. With respect to rules, the satisfaction of all the conditions in a conditional rewrite rule is attempted sequentially from left to right, solving rewrite conditions by means of search; for this reason, we can have new variables in such conditions but they must become instantiated along this process of solving from left to right (see [10] for details). Furthermore, the strategy followed by Maude in rewriting with rules is to compute the normal form of a term with respect to the equations before applying a rule. This strategy is guaranteed not to miss any rewrites when the rules are *coherent* with respect to the equations [26, 10].

The following section describes an example of a Maude system module with both equations and rules.

2.4 An Example: Knight's Tour Problem

A knight's tour is a journey around the chessboard in such a way that the knight lands on each square exactly once. The legal move for a knight is two spaces in one direction, then one in a perpendicular direction. We want to solve the problem for a 3×4 chessboard with the knight initially located in one corner.

We represent positions in the chessboard as pairs of integers and journeys as lists of positions.

```
(mod KNIGHT is
  protecting INT .
  sorts Position Movement Journey Problem .
  subsort Position < Movement .
  subsorts Position < Journey < Problem .
  op [_,_] : Int Int -> Position .
  op nil : -> Journey .
  op __ : Journey Journey -> Journey [assoc id: nil] .
  vars N X Y : Int .  vars P Q : Position .  var  J : Journey .
```

The term `move P` represents a position reachable from position `P`. Since the reachable positions are not unique, this operation is defined by means of rewrite rules, instead of equations. The reachable positions can be outside the chessboard, so we define the operation `legal`, that checks if a position is inside the 3×4 chessboard.

```
  op move_ : Position -> Movement .
  rl [mv1] : move [X, Y] => [X + 2, Y + 1] .
  ...
  rl [mv8] : move [X, Y] => [X - 1, Y - 2] .
  op legal : Position -> Bool .
  eq [leg] : legal([X, Y]) = X >= 1 and Y >= 1 and X <= 3 and Y <= 4 .
```

The function `contains(J, P)` checks if position `P` occurs in the journey `J`.

```
  op contains : Journey Position -> Bool .
  eq [con1] : contains(P J, P) = true .
  eq [con2] : contains(J, P) = false [otherwise] .
```

`knight(N)` represents a journey where the knight has performed `N` hops. When no hops are taken, the knight remains at the first position `[1, 1]`. When `N > 0` the problem is recursively solved (using backtracking in an implicit way) as follows: first a legal journey of `N - 1` steps is found, then a new hop from the last position of that journey is performed, and finally it is checked that this last hop is legal and compatible with the other ones.

```
  op knight : Nat -> Problem .
  rl [k1] : knight(0) => [1, 1] .
  crl [k2] : knight(N) => J P Q
   if N > 0
   /\ knight(N - 1) => J P
   /\ move P => Q
   /\ legal(Q)
   /\ not(contains(J P, Q)) .
endm)
```

The solution to the 3×4 chessboard can be found by looking for a journey with 11 hops, but we obtain the following unexpected, wrong result, where the journey contains repeated positions. We will show how to debug it in Sect. 4.

```
Maude> (rew knight(11) .)
result Journey :
   [1,1][2,3][3,1][2,3][3,1][2,3][3,1][2,3][3,1][2,3][3,1][2,3]
```

3 Debugging Trees for Maude Specifications

Now we will describe debugging trees for both *MEL* specifications and rewriting logic specifications. Since a *MEL* specification coincides with a rewrite theory with an empty set of rules, our treatment will simply be at the level of rewrite theories. Our proof and debugging trees will include statements for reductions $t \to t'$, memberships $t : s$, and rewrites $t \Rightarrow t'$, and in the following sections we will describe how to build the debugging trees from the proof trees taking into account each kind of statement.

3.1 Proof Trees

Before defining the debugging trees employed in our declarative debugging framework we introduce the semantic rules defining the semantics of a rewrite theory $\mathcal{R}$. The inference rules of the calculus can be found in Fig. 1, where θ denotes a substitution. The rules allow to deduce statements of the three kinds and are an adaptation of the rules presented in [1, 15] for *MEL* and in [14, 2] for rewriting logic. With respect to *MEL*, because of the executability assumptions, we have a more operational interpretation of the equations, which are oriented from left to right. With respect to rewriting logic, we work with terms (as in [2]) instead of equivalence classes of terms (as in [14]); moreover, unlike [2], replacement is not nested. Both changes make the logical representation closer to the way the Maude system operates. As usual, we represent deductions in the calculus as *proof trees*, where the premises are the child nodes of the conclusion at each inference step. We assume that the inference labels ($\mathit{Rep}_{\Rightarrow}$), ($\mathit{Rep}_{\rightarrow}$), and ($\mathit{Mb}$) decorating the inference steps contain information about the particular rewrite rule, equation, and membership axiom, respectively, applied during the inference. This information will be used by the debugger in order to present to the user the incorrect fragment of code causing the error.

In our debugging framework we assume the existence of an *intended interpretation* $\mathcal{I}$ of the given rewrite theory $\mathcal{R} = (\Sigma, E, R)$. The intended interpretation must be an $\mathcal{R}$-system corresponding to the model that the user had in mind while writing the specification $\mathcal{R}$. Therefore the user expects that $\mathcal{I} \models t \Rightarrow t'$, $\mathcal{I} \models t \rightarrow t'$, and $\mathcal{I} \models t : s$ for each rewrite $t \Rightarrow t'$, reduction $t \rightarrow t'$, and membership $t : s$ computed w.r.t. the specification $\mathcal{R}$. We will say that a statement $t \Rightarrow t'$ (respectively $t \rightarrow t'$, $t : s$) is *valid* when it holds in $\mathcal{I}$, and *invalid* otherwise. Declarative debuggers rely on some external oracle, normally the user, in order to obtain information about the validity of some nodes in the debugging tree. The concept of validity can be extended to distinguish *wrong rules*, *wrong equations*, and *wrong membership axioms*, which are those specification pieces that can deduce something invalid from valid information.

Definition 1. *Let* $r \equiv (af \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w'_k)$ *where* af *denotes an atomic formula, that is,* r *is either a rewrite rule, an oriented equation, or a membership axiom (in the last two cases* $l = 0$*) in some rewrite theory* $\mathcal{R}$*. Then:*

- $\theta(r)$ *is a* wrong rewrite rule instance *(respectively* wrong equation instance *and* wrong membership axiom instance*) w.r.t. an intended interpretation* $\mathcal{I}$ *when*
  1. *There exist terms* $t_1, \ldots, t_n$ *such that* $\mathcal{I} \models \theta(u_i) \rightarrow t_i$, $\mathcal{I} \models \theta(u'_i) \rightarrow t_i$ *for* $i = 1 \ldots n$.
  2. $\mathcal{I} \models \theta(v_j) : s_j$ *for* $j = 1 \ldots m$.
  3. $\mathcal{I} \models \theta(w_k) \Rightarrow \theta(w'_k)$ *for* $k = 1 \ldots l$.
  4. $\theta(af)$ *does not hold in* $\mathcal{I}$.
- r *is a* wrong rewrite rule *(respectively,* wrong equation *and* wrong membership axiom*) if it admits some wrong instance.*

(**Reflexivity**)

$$\frac{}{t \Rightarrow t}\ {\scriptstyle (Rf_{\Rightarrow})} \qquad\qquad \frac{}{t \rightarrow t}\ {\scriptstyle (Rf_{\rightarrow})}$$

(**Transitivity**)

$$\frac{t_1 \Rightarrow t' \qquad t' \Rightarrow t_2}{t_1 \Rightarrow t_2}\ {\scriptstyle (Tr_{\Rightarrow})} \qquad\qquad \frac{t_1 \rightarrow t' \qquad t' \rightarrow t_2}{t_1 \rightarrow t_2}\ {\scriptstyle (Tr_{\rightarrow})}$$

(**Congruence**)

$$\frac{t_1 \Rightarrow t'_1 \quad \ldots \quad t_n \Rightarrow t'_n}{f(t_1, \ldots, t_n) \Rightarrow f(t'_1, \ldots, t'_n)}\ {\scriptstyle (Cong_{\Rightarrow})} \qquad \frac{t_1 \rightarrow t'_1 \quad \ldots \quad t_n \rightarrow t'_n}{f(t_1, \ldots, t_n) \rightarrow f(t'_1, \ldots, t'_n)}\ {\scriptstyle (Cong_{\rightarrow})}$$

(**Replacement**)

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n}\ \{\theta(v_j) : s_j\}_{j=1}^{m}\ \{\theta(w_k) \Rightarrow \theta(w'_k)\}_{k=1}^{l}}{\theta(t) \Rightarrow \theta(t')}\ {\scriptstyle (Rep_{\Rightarrow})}$$

if $t \Rightarrow t' \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w'_k$

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(t) \rightarrow \theta(t')}\ {\scriptstyle (Rep_{\rightarrow})} \text{ if } \ t \rightarrow t' \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j$$

(**Equivalence Class**) (**Subject Reduction**)

$$\frac{t \rightarrow t' \quad t' \Rightarrow t'' \quad t'' \rightarrow t'''}{t \Rightarrow t'''}{\scriptstyle (EC)} \qquad\qquad \frac{t \rightarrow t' \quad t' : s}{t : s}{\scriptstyle (SRed)}$$

(**Membership**)

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(t) : s}\ {\scriptstyle (Mb)} \qquad \text{if } \ t : s \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j$$

Fig. 1. Semantic calculus for Maude modules

The general schema of [17] presents declarative debugging as the search of *buggy nodes* (invalid nodes with all children valid) in a debugging tree representing an erroneous computation. In our scheme instance, the proof trees constructed by the inferences of Fig. 1 seem natural candidates for debugging trees. Although this is a possible option, we will use instead a suitable abbreviation of these trees. This is motivated by the following result:

Proposition 1. *Let N be a buggy node in some proof tree in the calculus of Fig. 1 w.r.t. an intended interpretation $\mathcal{I}$. Then:*

1. *N is the result of either a* membership *or a* replacement *inference step.*
2. *The statement associated to N is either a wrong rewrite rule, a wrong equation, or a wrong membership axiom.*

Both points are a consequence of the definition of the semantic calculus. The first result states that all the inference steps different from *membership* and *replacement* are logically sound w.r.t. the definition of $\mathcal{R}$-system, i.e., they always produce valid results from valid premises. The second result can be checked by observing that any *membership* or *replacement* buggy node satisfies the requirements of Def. 1: the valid premises correspond to the points 1-3 of the definition, while the invalid conclusion fulfills the last point.

3.2 Abbreviated Proof Trees

Our goal is to find a buggy node in any proof tree T rooted by the initial error symptom detected by the user. This could be done simply by asking questions to the user about the validity of the nodes in the tree according to the following *top-down* strategy:

Input: A tree T with an invalid root.
Output: A buggy node in T.
Description: Consider the root N of T. There are two possibilities: if all the children of N are valid, then finish pointing out at N as buggy; otherwise, select the subtree rooted by any invalid child and use recursively the same strategy to find the buggy node.

Proving that this strategy is complete is straightforward by using induction on the height of T.

However, we will not use the proof tree T as debugging tree, but a suitable abbreviation which we denote by $APT(T)$ (from *Abbreviated Proof Tree*), or simply APT if the proof tree T is clear from the context. The reason for preferring the APT to the original proof tree is that it reduces and simplifies the questions that will be asked to the user while keeping the soundness and completeness of the technique. In particular the APT essentially contains only nodes related to the *replacement* and *membership* inferences using statements included in the specification, which are the only possible buggy nodes as Prop. 1 indicates. Thus, in order to minimize the number of questions asked to the user the debugger should consider the validity of $(Rep_{\Rightarrow})$, $(Rep_{\rightarrow})$, or (Mb). The APT rules can be seen in Fig. 2.

The rules are assumed to be applied top-down: if several APT rules can be applied at the root of a proof tree, we must choose the first one, that is, the rule of least number. As a matter of fact, the figure includes rules for two different possible $APTs$, which we call *one-step* abbreviated proof tree (in short $APT^o(T)$), defined by all the rules in the figure excluding $(\mathbf{APT}_4^m)$, and *many-steps* abbreviated proof tree (in short $APT^m(T)$), defined by all the rules in the figure excluding $(\mathbf{APT}_4^o)$. Analogously, we will use the notation $APT'^o(T)$ (resp. $APT'^m(T)$) for the subset of rules of APT' excluding $(\mathbf{APT}_4^m)$ (resp. $(\mathbf{APT}_4^o)$).

The one-step debugging tree follows strictly the idea of keeping only nodes corresponding to the *replacement* and *membership* inference rules. However, the many-steps debugging tree also keeps nodes corresponding to the *transitivity* inference rule for rewrites. The user will choose which debugging tree (one-step or many-steps) will be used for the declarative debugging session, taking into account that the many-steps debugging tree usually leads to shorter debugging

$$(\mathbf{APT}_1)\quad APT\left(\frac{T_1 \ldots T_n}{af}{\scriptstyle (R)}\right) = \frac{APT'\left(\frac{T_1 \ldots T_n}{af}{\scriptstyle (R)}\right)}{af}$$

$$(\mathbf{APT}_2)\quad APT'\left(\frac{}{t \Rrightarrow t}{\scriptstyle (Rf_{\Rrightarrow})}\right) = \emptyset$$

$$(\mathbf{APT}_3)\quad APT'\left(\frac{\frac{T_1 \ldots T_n}{t_1 \to t'}{\scriptstyle (Rep_{\to})} \quad T'}{t_1 \to t_2}{\scriptstyle (Tr_{\to})}\right) =$$

$$\left\{\frac{APT'(T_1) \ldots APT'(T_n)\ APT'(T')}{t_1 \to t_2}{\scriptstyle (Rep_{\to})}\right\}$$

$$(\mathbf{APT}^o_4)\quad APT'\left(\frac{T_1 \quad T_2}{t_1 \Rightarrow t_2}{\scriptstyle (Tr_{\Rightarrow})}\right) = APT'(T_1) \bigcup APT'(T_2)$$

$$(\mathbf{APT}^m_4)\quad APT'\left(\frac{T_1 \quad T_2}{t_1 \Rightarrow t_2}{\scriptstyle (Tr_{\Rightarrow})}\right) = \left\{\frac{APT'(T_1)\ APT'(T_2)}{t_1 \Rightarrow t_2}{\scriptstyle (Tr_{\Rightarrow})}\right\}$$

$$(\mathbf{APT}_5)\quad APT'\left(\frac{T_1 \ldots T_n}{t_1 \Rrightarrow t_2}{\scriptstyle (Cong_{\Rrightarrow})}\right) = APT'(T_1) \bigcup \ldots \bigcup APT'(T_n)$$

$$(\mathbf{APT}_6)\quad APT'\left(\frac{T_1 \quad T_2}{t : s}{\scriptstyle (SRed)}\right) = APT'(T_1) \bigcup APT'(T_2)$$

$$(\mathbf{APT}_7)\quad APT'\left(\frac{T_1 \ldots T_n}{t : s}{\scriptstyle (Mb)}\right) = \left\{\frac{APT'(T_1) \ldots APT'(T_n)}{t : s}{\scriptstyle (Mb)}\right\}$$

$$(\mathbf{APT}_8)\quad APT'\left(\frac{T_1 \ldots T_n}{t_1 \Rrightarrow t_2}{\scriptstyle (Rep_{\Rrightarrow})}\right) = \left\{\frac{APT'(T_1) \ldots APT'(T_n)}{t_1 \Rrightarrow t_2}{\scriptstyle (Rep_{\Rrightarrow})}\right\}$$

$$(\mathbf{APT}_9)\quad APT'\left(\frac{T' \quad \frac{T_1 \ldots T_n}{t \Rightarrow t'}{\scriptstyle (Rep_{\Rightarrow})} \quad T''}{t_1 \Rightarrow t_2}{\scriptstyle (EC)}\right) =$$

$$\left\{\frac{APT'(T')\ APT'(T_1) \ldots APT'(T_n)\ APT'(T'')}{t_1 \Rightarrow t_2}{\scriptstyle (Rep_{\Rightarrow})}\right\}$$

$$(\mathbf{APT}_{10})\quad APT'\left(\frac{T_1 \ldots T_n}{t_1 \Rightarrow t_2}{\scriptstyle (EC)}\right) = APT'(T_1) \bigcup \ldots \bigcup APT'(T_n)$$

(R) any inference rule $\qquad \Rrightarrow$ either $\to$ or $\Rightarrow$
af either $t_1 \to t_2,\ t : s$ or $t_1 \Rightarrow t_2$

Fig. 2. Transforming rules for obtaining abbreviated proof trees

sessions (in terms of the number of questions) but with likely more complicated questions. The number of questions is usually reduced because keeping the transitivity nodes for rewrites shapes some parts of the debugging tree as a balanced binary tree (each transitivity inference has two premises, i.e., two child subtrees), and this allows the debugger to use very efficient navigation strategies [23, 24]. On the contrary, removing the transitivity inferences for rewrites (as rule ($\mathbf{APT}^o_4$) does) produces flattened trees where this strategy is no longer efficient. On the other hand, in rewrites $t \Rightarrow t'$ appearing as conclusion of the transitivity inference rule, the term t' can contain the result of rewriting several subterms of t, and determining the validity of such nodes can be complicated, while in the one-step debugging tree each rewrite node $t \Rightarrow t'$ corresponds to a single rewrite applied at t and checking its validity is usually easier. The user must balance the pros and cons of each option, and choose the best one for each debugging session.

The rules ($\mathbf{APT}_3$) and ($\mathbf{APT}_9$) deserve a more detailed explanation. They keep the corresponding label ($Rep_{\Rightarrow}$) but changing the conclusion of the replacement inference in the lefthand side. For instance, ($\mathbf{APT}_3$) replaces $t_1 \to t'$ by the conclusion of the next transitivity inference $t_1 \to t_2$. We do this as a pragmatic way of simplifying the structure of the APTs, since t_2 is obtained from t' and hence likely simpler (the root of the tree T' in ($\mathbf{APT}_3$) must be necessarily of the form $t' \to t_2$ by the structure of the inference rule for transitivity in Fig. 1). A similar reasoning explains the form of ($\mathbf{APT}_9$). We will formally state now that these changes are safe from the point of view of the debugger.

Theorem 1. *Let T be a finite proof tree representing an inference in the calculus of Fig. 1 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$ such that the root of T is invalid in $\mathcal{I}$. Then:*

- *Both $APT^o(T)$ and $APT^m(T)$ contain at least one buggy node (completeness).*
- *Any buggy node in $APT^o(T)$, $APT^m(T)$ has an associated wrong statement in $\mathcal{R}$ (correctness).*

The theorem states that we can safely employ the abbreviated proof tree as a basis for the declarative debugging of Maude system and functional modules: the technique will find a buggy node starting from any initial symptom detected by the user. Of course, these results assume that the user answers correctly all the questions about the validity of the APT nodes asked by the debugger.

4 A Debugging Session

The debugger is initiated in Maude by loading the file `dd.maude` (available from `http://maude.sip.ucm.es/debugging`). This starts an input/output loop that allows the user to interact with the tool. Then, the user can enter Full Maude modules and commands, as well as commands for the debugger. The current version supports all kinds of modules. When debugging a rewrite computation, two different debugging trees can be built: one whose questions are related to one-step rewrites and another whose questions are related to several steps. The latter tree is partially built so that any node corresponding to a one-step rewrite is expanded only when the navigation process reaches it.

The debugger provides two strategies to traverse the debugging tree: *top-down*, that traverses the tree from the root asking each time for the correctness of all the children of the current node, and then continues with one of the incorrect children; and *divide and query*, that each time selects the node whose subtree's size is the closest one to half the size of the whole tree, keeping only this subtree if its root is incorrect, and deleting the whole subtree otherwise. Note that, although the navigation strategy can be changed during the debugging session, the construction strategy is selected before the tree is built and cannot be changed.

The user can select a module containing only correct statements. By checking the correctness of the inferences with respect to this module (i.e., using this module as oracle) the debugger can reduce the number of questions. The debugger allows us to debug specifications where some statements are suspicious and have been labeled. Only these labeled statements generate nodes in the proof tree, being the user in charge of this labeling. The user can decide to use all the labeled statements as suspicious or can use only a subset by trusting labels and modules. Moreover, the user can answer that he trusts the statement associated with the currently questioned inference; that is, statements can be trusted "on the fly." The user can also give the answer "don't know," that postpones the answer to that question by asking alternative questions. An `undo` command, allowing the user to return to the previous state, is also provided. We refer the reader to [22, 21] for further information.

In Sect. 2.4 we described a system module that simulates a knight's tour. However, this system module contains a bug and the knight repeats some positions in its tour. This error is also obtained when looking for a 3 steps journey:

```
Maude> (rew knight(3) .)
result List : [1,1][2,3][3,1][2,3]
```

Thus, we debug this smaller computation. Moreover, after inspecting the rewrite rules describing the eight possible moves, we are sure that they are not responsible for the error; therefore, we trust them by using commands that allow us to select the suspicious statements.

```
Maude> (set debug select on .)
Maude> (debug select con1 con2 leg k1 k2 .)
Maude> (debug knight(3) =>* [1,1][2,3][3,1][2,3] .)
```

The default one-step tree construction strategy is used and the tree shown below is built, where every operation has been abbreviated with its first letter.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{k(0)} \Rightarrow_1 \texttt{[1,1]}}\,\textsf{k1} \quad \dfrac{}{\texttt{l([2,3])} \to \texttt{t}}\,\textsf{leg} \quad \dfrac{}{\texttt{c([1,1],[2,3])} \to \texttt{f}}\,\textsf{con2}}{\texttt{k(1)} \Rightarrow_1 \texttt{J2}}\,\textsf{k2} \quad \dfrac{}{\texttt{l([3,1])} \to \texttt{t}}\,\textsf{leg} \quad \dfrac{}{\texttt{c(J2,[3,1])} \to \texttt{f}}\,\textsf{con2}}{\texttt{k(2)} \Rightarrow_1 \texttt{J1}}\,\textsf{k2} \quad \dfrac{}{\texttt{l([2,3])} \to \texttt{t}}\,\textsf{leg} \quad \dfrac{}{\texttt{c(J1,[2,3])} \to \texttt{f}}\,\textsf{con2}}{\texttt{k(3)} \Rightarrow_1 \texttt{[1,1][2,3][3,1][2,3]}}\,\textsf{k2}$$

where `J1` denotes the journey `[1,1][2,3][3,1]` and `J2` denotes `[1,1][2,3]`.

Since the tree is navigated by using the default divide and query strategy, the first two questions asked by the debugger are

```
Is this rewrite (associated with the rule k2) correct?
knight(1) =>1 [1,1][2,3]
Maude> (yes .)
Is this rewrite (associated with the rule k2) correct?
knight(2) =>1 [1,1][2,3][3,1]
Maude> (yes .)
```

Notice the form `=>1` of the arrow in the rewrites appearing in the questions, to emphasize that they are one-step rewrites.

In both cases the answer is `yes` because these paths are possible, legal behaviors of the knight when it can do one or two hops. These two subtrees are removed and the current tree looks as follows:

$$\dfrac{\dfrac{}{\texttt{l([2,3])} \to \texttt{t}}\,\textsf{leg} \quad \dfrac{}{\texttt{c(J1,[2,3])} \to \texttt{f}}\,\textsf{con2}}{\texttt{k(3)} \Rightarrow_1 \texttt{[1,1][2,3][3,1][2,3]}}\,\textsf{k2}$$

The next question is

```
Is this reduction (associated with the equation con2) correct?
contains([1,1][2,3][3,1],[2,3]) -> false
Maude> (no .)
```

Clearly, this is not a correct reduction, since position `[2,3]` is already in the path `[1,1][2,3][3,1]`. With this answer this subtree is selected and, since it is a single node, the bug is located:

```
The buggy node is:
contains([1,1][2,3][3,1],[2,3]) -> false
with the associated equation: con2
```

Looking at the definition of the `contains` operation, we realize that it defines the membership operation for *sets*, not for lists. A correct definition of the `contains` operation is as follows:

```
  eq [con1] : contains(nil, P) = false .
  eq [con2] : contains(Q J, P) = P == Q or contains(J, P) .
```

5 The Implementation

As mentioned in the introduction, a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [10, Chap. 14]. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. Therefore, we are able to generate and navigate the debugging tree of a Maude computation using operations in Maude itself. In addition, the Maude system provides another module, `LOOP-MODE` [10, Chap. 17], which can be used to specify input/output interactions with the user. Thus, our declarative debugger, including its user interface, is implemented in Maude itself, as an extension of Full Maude [10, Chap. 18]. Instead of creating the complete proof tree and then abbreviating it, we build the abbreviated proof tree directly. Since navigation is done by asking questions to the user, this stage has to handle the navigation strategy together with the input/output interaction with the user. The technical report [21] provides a full explanation of this implementation, including the user interaction.

The way in which the debugging trees for reductions and memberships or rewrites are built is completely different. In the first case, we use the facts that equations and membership axioms are both terminating and confluent, which allow us to build the debugging tree in a "greedy" way, selecting at each moment the first equation applicable to the current term. However, we have to use a different methodology in the construction of the debugging tree for incorrect rewrites. We use breadth-first search to find from the initial term the wrong term introduced by the user, and then we use the found path to build the debugging tree in the two possible ways described in previous sections.

The functions in charge of building the debugging trees, that correspond to the APT function from Fig. 2, have a common initial behavior. They receive the module where the wrong inference took place, a correct module (or a special constant when no such module is provided) to prune the tree, the initial term, the (erroneous) result obtained, and the set of suspicious statements labels. They keep the initial inference as the root of the tree and generate the forest of abbreviated trees corresponding to the inference with functions that, in addition to the arguments above, receive the initial module "cleaned" of suspicious statements and correspond to the APT' function from Fig. 2. This transformed module is used to improve the efficiency of the tree construction, because we can use it to check if an inference can be obtained by using only trusted statements, thus avoiding to build a tree that will be finally empty.

The function that builds debugging trees for wrong reductions works with the same innermost strategy as the Maude interpreter: it first fully reduces the subterms recursively building their debugging trees (it mimics a specific behavior of the *congruence* rule in Fig. 1), and once all the subterms have been reduced, if the result is not the final one, it tries to reduce at the top to reach the final result by *transitivity*. Reduction at the top tries to apply one equation,[3] by

[3] Since the module is assumed to be confluent, we can choose any equation and the final result should be the same.

using the *replacement* rule from Fig. 1. Debugging trees for the conditions of the equation are also built and placed as children of the replacement rule. The construction of debugging trees for wrong memberships mimics the *subject reduction* rule from Fig. 1 by computing the tree for the full reduction of the term and then computing the tree for the membership inference of its least sort by using the operator declarations and the membership axioms, which corresponds to a concrete application of the *membership* inference rule.

The one-step tree for wrong rewrites computes the tree for the reduction from the initial term to normal form and then computes the rest of the tree, that corresponds to a rewrite from a fully reduced term (this corresponds to a concrete application of the *equivalence class* inference rule from Fig. 1). The debugging tree for this rewrite is computed from the trace, that is obtained with the predefined function `metaSearchPath`. Each step of the trace corresponds to the application of one rule, that generates a tree, with the trees corresponding to the conditions of the rule as its children (reproducing the *replacement* rule). Note that although the information in the trace is related to the whole rewritten term, the application of a rule can be in a subterm, which corresponds with the *congruence* inference rule, so only the rewritten subterms appear in the debugging tree. Other children are generated for the reduction to normal form due to the *equivalence class* inference rule. Finally, all the steps are put together as children of the same root by using the *transitivity* inference rule. The many-steps debugging tree is built *by demand*, so that the debugging subtrees corresponding to one-step rewrites are only generated when they are pointed out as wrong. These one-step nodes are used to create a balanced binary tree, by dividing them into two forests of approximately the same size, recursively creating their trees, and then using them as children of a new binary tree that has as root the combination by *transitivity* of the rewrites in their roots.

6 Concluding Remarks

In this paper we have developed the foundations of declarative debugging of executable rewriting logic specifications, and we have applied them to implement a debugger for Maude modules. The work encompasses and extends previous presentations [5, 6] on the declarative debugging of Maude functional modules, which constitute now a particular case of a more general setting.

We have formally described how debugging trees can be obtained from Maude proof trees, proving the correctness and completeness of the debugging technique. The tool based on these ideas allows the user to concentrate on the logic of the program disregarding the operational details. In order to deal with the possibly complex questions associated to rewrite statements, the tool offers the possibility of choosing between two different debugging trees: the one-step trees, with simpler questions and likely longer debugging sessions, and the many-steps trees, which in general require fewer but more complex questions before finding the bug. The experience will show the user which one must be chosen in each case depending on the complexity of the specification.

In our opinion, this debugger provides a complement to existing debugging techniques for Maude, such as tracing and term coloring. An important advantage of our debugger is the help provided by the tool in locating the buggy statements, assuming the user answers correctly the corresponding questions.

As future work we want to provide a graphical interface, that allows the user to navigate the tree with more freedom. We are also investigating how to improve the questions done in the presence of the `strat` operator attribute, that allows the specifier to define an evaluation strategy. This can be used to represent some kind of laziness. Finally, we plan to study how to debug *missing answers* [8, 16] in addition to the wrong answers we have treated thus far.

References

1. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.
2. R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1):386–414, 2006.
3. R. Caballero. A declarative debugger of incorrect answers for constraint functional-logic programs. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05), Tallinn, Estonia*, pages 8–13. ACM Press, 2005.
4. R. Caballero, C. Hermanns, and H. Kuchen. Algorithmic debugging of Java programs. In F. J. López-Fraguas, editor, *15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid, Spain*, volume 177 of *Electronic Notes in Theoretical Computer Science*, pages 75–89. Elsevier, 2007.
5. R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. Declarative debugging of membership equational logic specifications. In P. Degano, R. De Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models. Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes for Computer Science*, pages 174–193. Springer, 2008.
6. R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. A declarative debugger for Maude functional modules. In G. Roşu, editor, *Proceedings Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008, Budapest, Hungary*. Elsevier, 2009. To appear.
7. R. Caballero and M. Rodríguez-Artalejo. DDT: A declarative debugging tool for functional-logic languages. In Y. Kameyama and P. J. Stuckey, editors, *Proceedings 7th International Symposium on Functional and Logic Programming (FLOPS'04), Nara, Japan*, volume 2998 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2004.
8. R. Caballero, M. Rodríguez-Artalejo, and R. del Vado Vírseda. Declarative diagnosis of missing answers in constraint functional-logic programming. In J. Garrigue and M. V. Hermenegildo, editors, *Proceedings of 9th International Symposium on Functional and Logic Programming, FLOPS 2008, Ise, Japan*, volume 4989 of *Lecture Notes in Computer Science*, pages 305–321. Springer, 2008.
9. O. Chitil and Y. Luo. Structure and properties of traces for functional programs. In I. Mackie, editor, *Proceedings of the Third International Workshop on Term Graph Rewriting (TERMGRAPH 2006)*, volume 176 of *Electronic Notes in Theoretical Computer Science*, pages 39–63, Amsterdam, The Netherlands, The Netherlands, 2007. Elsevier.

10. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
11. M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2):70–91, 2007.
12. J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.
13. I. MacLarty. Practical declarative debugging of Mercury programs. Master's thesis, University of Melbourne, 2005.
14. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
15. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.
16. L. Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10(3):255–286, 1992.
17. L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.
18. H. Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.
19. H. Nilsson and P. Fritzson. Algorithmic debugging of lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, 1994.
20. B. Pope. *A Declarative Debugger for Haskell*. PhD thesis, The University of Melbourne, Australia, 2006.
21. A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. Declarative debugging of Maude modules. Technical Report SIC-6/08, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2008. `http://maude.sip.ucm.es/debugging`.
22. A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. A declarative debugger for Maude. In J. Meseguer and G. Roşu, editors, *Algebraic Methodology and Software Technology — 12th International Conference, AMAST 2008 Urbana, IL, USA, July 28-31, 2008 Proceedings*, volume 5140 of *Lecture Notes in Computer Science*, pages 116–121. Springer, 2008.
23. E. Y. Shapiro. *Algorithmic Program Debugging*. ACM Distinguished Dissertation. MIT Press, 1983.
24. J. Silva. A comparative study of algorithmic debugging strategies. In G. Puebla, editor, *Logic-Based Program Synthesis and Transformation*, volume 4407 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2007.
25. A. Tessier and G. Ferrand. Declarative diagnosis in the CLP scheme. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl project)*, volume 1870 of *Lecture Notes in Computer Science*, pages 151–174. Springer, 2000.
26. P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.