# A Complete Declarative Debugger for Maude*

Adrián Riesco, Alberto Verdejo, and Narciso Martí-Oliet

Facultad de Informática, Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es, {alberto,narciso}@sip.ucm.es

**Abstract.** We present a declarative debugger for Maude specifications that allows to debug wrong answers (a *wrong* result is obtained) and missing answers (a correct but *incomplete* result is obtained) due to both wrong and missing statements and wrong search conditions. The debugger builds a tree representing the computation and guides the user through it to find the bug. We present the debugger's latest commands and features, illustrating its use with several examples.

**Keywords:** Declarative debugging, Maude, missing answers, wrong answers.

## 1 Introduction

Declarative debugging [8] is a semi-automatic debugging technique that focuses on results, which makes it specially suited for declarative languages, whose operational details may be hard to follow. Declarative debuggers represent the computation as a tree, called *debugging tree*, where each node must be logically inferred from the results in its children. In our case, these trees are obtained by abbreviating proof trees obtained in a formal calculus [4,5]. Debugging progresses by asking questions related to the nodes of this tree (i.e., questions related to subcomputations of the wrong result being debugged) to an external oracle (usually the user), discarding nodes in function of the answers until a *buggy node*—a node with an erroneous result and with all its children correct—is located. Since each node in the tree has associated a piece of code, when this node is found, the bug, either a wrong or a missing statement,[1] is also found.

Maude [2] is a declarative language based on both equational and rewriting logic for the specification and implementation of a whole range of models and systems. Functional modules define data types and operations on them by means of *membership equational logic* theories that support multiple sorts, subsort relations, equations, and assertions of membership in a sort, while system modules specify rewrite theories that also support *rules*, defining local concurrent transitions that can take place in a system. As a programming language, a distinguishing feature of Maude is its use of reflection, that allows many metaprogramming applications. Moreover, the debugger is implemented on top of Full Maude [2, Chap. 18], a tool completely written in Maude which

* Research supported by MICINN Spanish project *DESAFIOS10* (TIN2009-14599-C03-01) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC-1465).

[1] It is important not to confuse wrong and missing answers with wrong and missing statements. The former are the initial symptoms that indicate the specifications fails, while the latter is the error that generated this misbehavior.

M. Johnson and D. Pavlovic (Eds.): AMAST 2010, LNCS 6486, pp. 216–225, 2011.

includes features for parsing and pretty-printing terms, improving the input/output interaction. Thus, our declarative debugger, including its user interactions, is implemented in Maude itself.

We extend here the tool presentation in [7], based on [1,4], for the debugging of wrong answers (wrong results, which correspond in our case to erroneous reductions, sort inferences, and rewrites) with the debugging of missing answers (incomplete results, which correspond here to not completely reduced normal forms, greater than expected least sorts, and incomplete sets of reachable terms), showing how the theory introduced in [5,6] is applied. The reasons the debugger is able to attribute to these errors are wrong and missing statements and, since missing answers in system modules are usually found with the `search` command [2, Chap. 6] that performs a reachability analysis, wrong search conditions.

With this extension we are able to present a state-of-the-art debugger, with several options to build, prune, and traverse the debugging tree. Following the classification in [9], these are the main characteristics of our debugger. Although we have not implemented all the possible strategies to shorten and navigate the debugging tree, like the latest tree compression technique or the answers "maybe yes," "maybe not," and "inadmissible," our trees are abbreviated (in our case, since we obtain the trees from a formal calculus, we are able to prove the correctness and completeness of the technique), statements can be trusted in several ways, a correct module can be used as oracle, `undo` and `don't know` commands are provided, the tree can be traversed with different strategies, and a graphical interface is available. Furthermore, we have developed a new technique to build the debugging tree: before starting the debugging process, the user can choose between different debugging trees, one that leads to shorter sessions (from the point of view of the number of questions) but with more complex questions, and another one that presents longer, although easier, sessions. We have successfully applied this approach to both wrong and missing answers.

Complete explanations about our debugger, including a user guide [3] that describes the graphical user interface, the source files for the debugger, examples, and related papers, are available in the webpage `http://maude.sip.ucm.es/debugging`.

## 2 Using the Debugger

We make explicit first what is assumed about the modules introduced by the user; then we present the new available commands.

**Assumptions.** A rewrite theory has an underlying equational theory, containing equations and memberships, which is expected to satisfy the appropriate executability requirements, namely, it has to be terminating, confluent, and sort decreasing. Rules are assumed to be coherent with respect to the equations; for details, see [2].

The tool allows to shorten the debugging trees in several ways: statements and complete modules can be trusted, a correct module can be used as oracle, constructed terms (terms built only with constructors, indicated with the attribute `ctor`) are considered to be in normal form, and constructed terms of some sorts or built with some operators can be pointed out as *final* (they cannot be further rewritten). This information, as well

as the answers given during the debugging process and the signature of the module, is assumed to be correct.

**Commands.** The debugger is started by loading the file `dd.maude`, which starts an input/output loop that allows the user to interact with the tool. Since the debugger is implemented on top of Full Maude, all modules and commands must be introduced enclosed in parentheses. Debugging of missing answers uses all the features already described for wrong answers: use of a correct module as oracle, trusting of statements and modules, and different types of debugging tree; see [3,7] for details about the corresponding commands.

When debugging missing answers we can select some sorts as final (i.e., they cannot be further rewritten) with `(final [de]select SORTS .)`, that only works once the final mode has been activated with `(set final select on/off .)`.

When debugging missing answers in system modules, two different trees can be built: one whose questions are related to one-step searches and another one whose questions are related to many-steps searches; the user can switch between these trees with the commands `(one-step missing-tree .)`, which is the default one, and `(many-steps missing-tree .)`, taking into account that the many-steps debugging tree usually leads to shorter debugging sessions but with likely more complicated questions.

The user can prioritize questions related to the fulfillment of the search condition from questions involving the statements defining it with `(solutions prioritized on/off .)`.

The debugging tree can be navigated by using two different strategies: the more intuitive *top-down strategy*, 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 the more efficient *divide and query strategy*, that each time selects the node whose subtree's size is the closest one to half the size of the whole tree, discarding the whole subtree if the inference in this node is correct and continuing the process with this subtree in other case. The user can switch between them with the commands `(top-down strategy .)` and `(divide-query strategy .)`, being divide and query the default strategy.

The debugging process is started with:

```
(missing [in MODULE-NAME :] INIT-TERM -> NF .)
(missing [in MODULE-NAME :] INIT-TERM : LS .)
(missing [[depth]] [in MODULE-NAME :] INIT-TERM ARROW PATTERN [s.t. COND] .)
```

The first command debugs normal forms, where `INIT-TERM` is the initial term and `NF` is the obtained unexpected normal form. Similarly, the second command starts the debugging of incorrect least sorts, where `LS` is the computed least sort. The last command refers to incomplete sets found when using search, where `depth` indicates the bound in the number of steps, which is considered unbounded when omitted; `ARROW` is `=>*` for searches in zero or more steps, `=>+` for searches in one or more steps, and `=>!` for final terms; and `COND` is the optional condition to be fulfilled by the results. Finally, when no module name is specified in a command, the default one is used.

When the divide and query strategy is selected, one question, that can be either correct or wrong (w.r.t. the intended behavior the user has in mind), is presented in each step. The different answers are transmitted with the commands `(yes .)` and `(no .)`. Instead of just answering `yes`, we can also *trust* some statements on the fly if, once the process has started, we decide the bug is not there. To trust the current statement we type the command `(trust .)`. If a question refers to a set of reachable terms and one of these terms is not reachable, the user can point it out with the answer `(I is wrong .)` where `I` is the index of the wrong term in the set; in case the question is related to a set of reachable *solutions*, if one of the terms should not be a solution the user can indicate it with `(I is not a solution .)`. Information about final terms can also be given on the fly with `(its sort is final .)`, which indicates that the least sort of the term currently displayed is final. If the current question is too complicated, it can be skipped with the command `(don't know .)`, although this answer can, in general, introduce incompleteness in the debugging process. When the top-down strategy is used, several questions will be displayed in each step. The answer in this case is transmitted with `(N : ANSWER)`, where `N` is the number of the question and `ANSWER` the answer the user would give in the divide and query strategy. As a shortcut to answer `yes` to all nodes, the tool also provides the answer `(all : yes .)`. Finally, we can return to the previous state by using the command `(undo .)`.

**Graphical User Interface.** The graphical user interface allows the user to visualize the debugging tree and navigate it with more freedom. More advanced users can take advantage of visualizing the whole tree to select the questions that optimize the debugging process (keeping in mind that the user is looking for incorrect nodes with all its children correct, he can, for example, search for erroneous nodes with few children or even erroneous leaves), while average users can just follow the implemented strategies and answer the questions in a friendlier way. Moreover, the interface eases the trusting of statements by providing information about the statements in each module and the subsort relations between sorts; provides three different navigation strategies (top-down, divide and query, and free); and implements two different modes to modify the tree once the user introduces an answer: in the *prune* mode only the minimum amount of relevant information is depicted in each step, while in the *keep* mode the complete tree is kept through the whole debugging process, coloring the nodes depending on the information given by the user. The first mode centers on the debugging process, while the second one allows the user to see how different answers modify the tree.

## 3 Debugging Sessions

We illustrate how to use the debugger with two examples, the first one shows how to debug an erroneous normal form due to a missing statement and the second one how to debug an incomplete set of reachable terms due to a wrong statement. Complete details about both examples are available in the webpage.

**Example 1.** We want to implement a function that, given an initial city (built with the operator `city`, that receives a natural number as argument), a number of cities, and a

cost graph (i.e., a partial function from pairs of cities to natural numbers indicating the cost of traveling between cities), returns a tour around all the cities that finishes in the initial one. First, we specify a priority queue where the states of this function will be kept:

```
sorts Node PNodeQueue .
subsort Node < PNodeQueue .

op node : Path Nat -> Node [ctor] .
op mtPQueue : -> PNodeQueue [ctor] .
op _ _ : PNodeQueue PNodeQueue -> PNodeQueue [ctor assoc id: mtPQueue] .
```

where a `Path` is just a list of cities. We use this priority queue to implement the function `travel` in charge of computing this tour:

```
op travel : City Nat Graph -> TravelResult .
ceq travel(C, N, G) = travel(C, N, G, R, node(C, 0))
 if R := greedyTravel(C, N, G) .
```

This function uses an auxiliary function `travel` that, in addition to the parameters above, receives a potential best result, created with the operator `result` and computed with the greedy function `greedyTravel`, and the initial priority queue, that only contains the node `node(C, 0)`. This auxiliary function is specified with the equations:

```
ceq [tr1] : travel(C, N, G, R, ND PQ) = travel(C, N, G, R, PQ')
 if not isResult(ND, N) /\ N' := getCost(R) /\ N'' := getCost(ND) /\
    N'' < N' /\ PQ' := expand(ND, N, G, PQ) .

ceq [tr2] : travel(C, N, G, R, node(P C', N') PQ) =
                                  travel(C, N, G, result(P C' C, UB), PQ)
 if isResult(node(P C', N'), N) /\ N'' := getCost(R) /\
    UB := N' + (G [road(C, C')]) /\ UB < N'' .

ceq [tr3] : travel(C, N, G, R, node(P C', N') PQ) = travel(C, N, G, R, PQ)
 if isResult(node(P C', N'), N) /\ N'' := getCost(R) /\
    UB := N' + (G [road(C, C')]) /\ UB >= N'' .

ceq [tr4] : travel(C, N, G, R, ND PQ) = R
 if N' := getCost(R) /\ N'' := getCost(ND)  /\ N'' >= N' .
```

However, we forget to specify the equation that returns the result when the queue is empty:

```
eq [tr5] : travel(C, N, G, R, emptyPQueue) = R .
```

Without this equation, Maude is not able to compute the desired normal form in an example for 4 cities. To debug it, we first consider that `GTRAVELER`, the module defining the greedy algorithm, is correct and can be trusted:

```
Maude> (set debug select on .)
Debug select is on.

Maude> (debug exclude GTRAVELER .)
Labels cheap1 cheap2 cheap3 gc1 gc2 gc3 gt1 gt2 in1 in2 in3 mi1 mi2 sz1
       sz2 are now trusted.
```

The command indicates that all the statements in the `GTRAVELER` module, whose labels are shown, are now trusted (unlabeled statements are trusted by default). Now, we debug a wrong reduction with the following command, where the term on the left of the arrow is the initial term we tried to reduce, the term on the right is the obtained result, and `G` abbreviates the cost graph:

```
(missing travel(city(0), 3, generateCostMatrix(3)) -> travel(city(0),3,G,
      result(city(0)city(3)city(1)city(2)city(0),15),emptyPQueue) .)
```

With this command the debugger builds the debugging tree, that is navigated with the default divide and query strategy. The first question is:

```
Is travel(city(0), 3, G, result(city(0) city(3) city(1) city(2) city(0), 15),
emptyPQueue) in normal form?
Maude> (no .)
```

Since we expected `travel` to be reduced to a result, this term is not in normal form. The next question is:

```
Is PNodeQueue the least sort of emptyPQueue ?
Maude> (yes .)
```

In fact, `emptyPQueue`—the empty priority queue—has as least sort `PNodeQueue`, the sort for priority queues. With this information the debugger locates the error, indicating that either another equation is required for the operator `travel` or that the conditions in the current equations are wrong:

```
The buggy node is:
travel(city(0), 3, G, result(city(0) city(3) city(1) city(2) city(0), 15),
emptyPQueue) is in normal form.
Either the operator travel needs more equations or the conditions of the
current equations are not written in the intended way.
```

When a missing statement is detected, the debugger indicates that either a new statement is needed or the conditions can be changed to allow more matchings (e.g., the user can change `N > 2` by `N >= 2`). Note that the error may be located in the conditions but *not* in the statements defining them, since they are also checked during the debugging process.

**Example 2.** Assume now we have specified the district in charge of a firefighters brigade. Buildings are represented by their street, avenue (New York alike coordinates), time to collapse (all of them natural numbers), and status, that can be `ok` or `fire`. When the status is `fire` the time to collapse is reduced until it reaches 0, when the fire cannot be extinguished, and the fire can be propagated to the nearest buildings:

```
sorts Building Neighborhood Status FFDistrict .
subsort Building < Neighborhood .

ops ok fire : -> Status [ctor] .
op < av :_, st :_, tc :_, sts :_> : Nat Nat Nat Status -> Building [ctor] .
```

The neighborhood is built with the empty neighborhood and the juxtaposition operator:

```
op empty : -> Neighborhood [ctor] .
op _ _ : Neighborhood Neighborhood -> Neighborhood
                                      [ctor assoc comm id: empty] .
```

Finally, the district contains the buildings in the neighborhood and two natural numbers indicating the avenue and the street where the firefighters are currently located:

```
op [_]_ _ : Neighborhood Nat Nat -> FFDistrict [ctor] .
```

The firefighters travel with the rule:

```
crl [go] : [NH] Av1 St1
 => [NH''] Av2 St2
 if extinguished?(NH, Av1, St1) /\
    B NH' := NH /\
    fire?(B) /\
    Av2 := getAvenue(B) /\
    St2 := getStreet(B) /\
    N := distance(Av1, St1, Av2, St2) /\
    NH'' := update(NH, N) .
```

where the conditions check that the building in the current location is not in fire with `extinguished?`, search for a building in fire `B` with the matching condition and the condition `fire?`, extract the avenue `Av2` and the street `St2` of this building with `getAvenue` and `getStreet`, compute the distance between the current and the new location with `distance`, and finally update the neighborhood (the time to collapse of the buildings is reduced an amount equal to the distance previously computed) with `update`. We are interested in the equational condition `fire?(B)`, because the equation `f2` specifying `fire?`, a function that checks whether the status of a building is `fire`, is buggy and returns `false` when it should return `true`:

```
op fire? : Building -> Bool .
eq [f1] : fire?(< av : Av, st : St, tc : TC, sts : ok >) = false .
eq [f2] : fire?(< av : Av, st : St, tc : TC, sts : fire >) = false .
```

If we use the command `search` to find the final states where the fire in both buildings has been extinguished, in an example with two buildings on fire with time to collapse `4` located in $(1,2)$ and $(2,1)$, and the firefighters initially in $(0,0)$, we realize that no states fulfilling the condition are found:

```
Maude> (search nh =>! F:FFDistrict s.t. allOk?(F:FFDistrict) .)
search in TEST : nh =>! F:FFDistrict .
No solution.
```

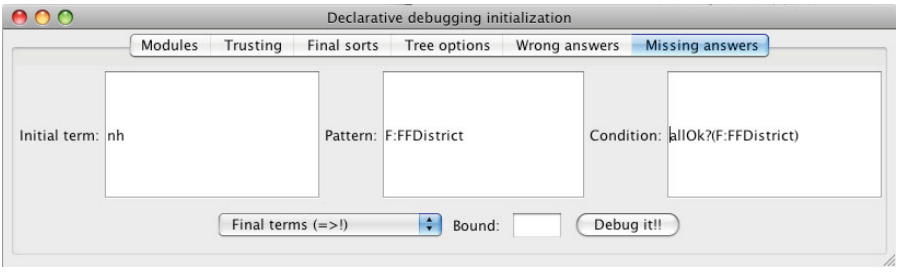


**Fig. 1.** Initial command for the firefighters example

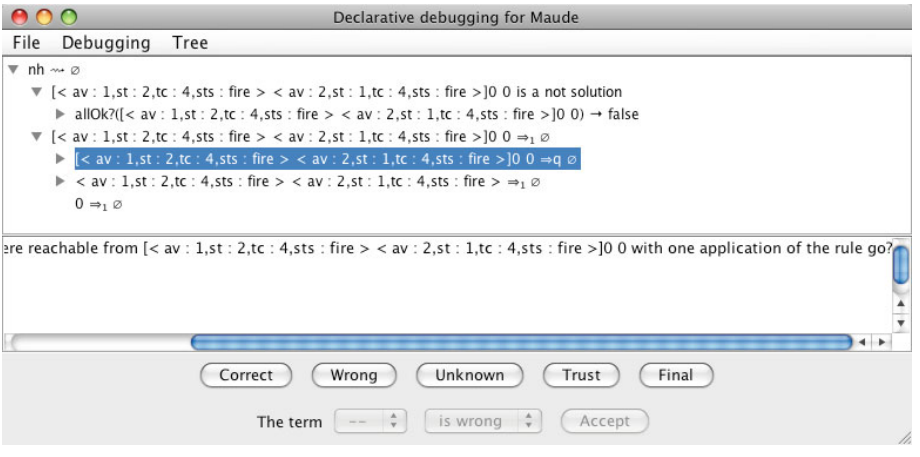


**Fig. 2.** Debugging tree for the firefighters example

where `nh` is a constant with value `[< av : 1, st : 2, tc : 4, sts : fire > < av : 2, st : 1, tc : 4, sts : fire >] 0 0`. We can debug this behavior with the command shown in Figure 1, that builds the default one-step tree. We show in Figure 2 how the corresponding tree, with only two levels expanded, is depicted by the graphical user interface.

Although this graphical interface has both the divide and query and the top-down navigation strategies implemented, advanced users can find more useful the free navigation strategy, that can greatly reduce the number of questions asked to the user. To achieve this, the user must find either correct nodes (which are removed from the tree) rooting big subtrees or wrong nodes (which are selected as current root) rooting small trees.

In the tree depicted in Figure 2 we notice that Maude cannot apply the rule `go` (the rule is shown once the node is selected) to a configuration with some buildings with status `fire`, which is indicated by the interface by showing that the term is rewritten to the empty set, `∅`. Since this is incorrect, we can use the button `Wrong` to indicate it and the tree in Figure 3 is shown. Note that this node is associated to a concrete rule

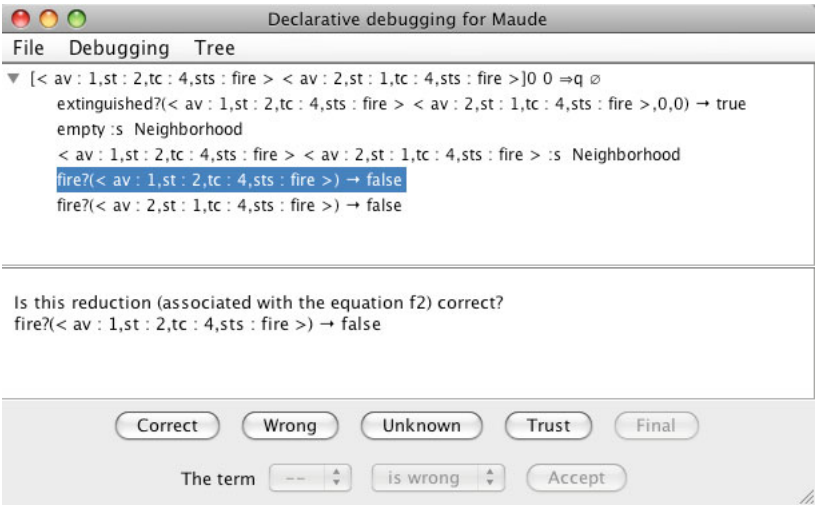


**Fig. 3.** Debugging tree for the firefighters example after one answer

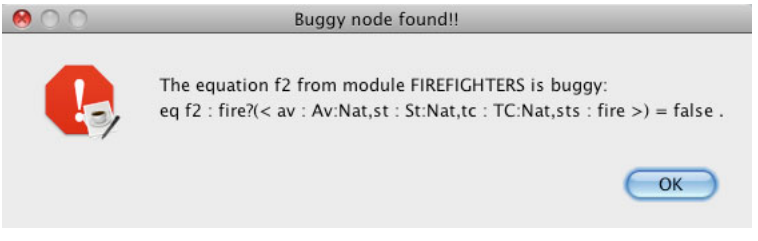


**Fig. 4.** Bug found in the firefighters example

and then the debugger allows the user to use the command `Trust`, that would remove all the nodes associated to this same rule from the debugging tree; trusting the most frequent statements is another strategy that can easily be followed by the user when using the graphical interface. However, since the question is incorrect, we cannot use this command to answer it.

In this case, the first three nodes are correct (the notation `_:s_` appearing in the second and third nodes indicates that the term has this sort as least sort), but we can select any of the other nodes as erroneous; the interested reader will find that both of them lead to the same error. In our case, we select the fourth node as erroneous and the debugger shows the error, as shown in Figure 4.

## 4 Conclusions

We have implemented a declarative debugger of wrong and missing answers for Maude, that is able to detect wrong and missing statements and wrong search conditions. Since

one of the main drawbacks of declarative debugging is the size of the debugging trees and the complexity of the questions, we provide several mechanisms to shorten and ease the tree, including a graphical user interface. As future work, we plan to implement a test case generator, that integrated with the debugger will allow to test Maude specifications and debug the erroneous cases.

## References


1. Caballero, R., Martí-Oliet, N., Riesco, A., Verdejo, A.: A declarative debugger for Maude functional modules. In: Roşu, G. (ed.) Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008. ENTCS, vol. 238(3), pp. 63–81. Elsevier, Amsterdam (2009)
2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
3. Riesco, A., Verdejo, A., Caballero, R., Martí-Oliet, N.: A declarative debugger for Maude specifications - User guide. Technical Report SIC-7-09, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid (2009), `http://maude.sip.ucm.es/debugging`
4. Riesco, A., Verdejo, A., Caballero, R., Martí-Oliet, N.: Declarative debugging of rewriting logic specifications. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 308–325. Springer, Heidelberg (2009)
5. Riesco, A., Verdejo, A., Martí-Oliet, N.: Declarative debugging of missing answers for Maude specifications. In: Lynch, C. (ed.) Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010). Leibniz International Proceedings in Informatics, vol. 6, pp. 277–294. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2010)
6. Riesco, A., Verdejo, A., Martí-Oliet, N.: Enhancing the debugging of Maude specifications. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 226–242. Springer, Heidelberg (2010)
7. Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R.: A declarative debugger for Maude. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 116–121. Springer, Heidelberg (2008)
8. Shapiro, E.Y.: Algorithmic Program Debugging. In: ACM Distinguished Dissertation. MIT Press, Cambridge (1983)
9. Silva, J.: A comparative study of algorithmic debugging strategies. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 143–159. Springer, Heidelberg (2007)