Test-Case Generation for Maude Functional Modules*

Adrián Riesco

Facultad de Informática, Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es

Abstract. Testing takes much of the time of the software development process, so several efforts have been devoted to automate it. We present here a tool that is able to generate test cases for Maude functional modules, and check their correctness with respect to a given specification or select a subset of these test cases to be checked by the user by using different strategies. Since these processes are very expensive we also present different trusting techniques to ease them.

Keywords: Test cases, Maude, black-box testing, white-box testing, code coverage.

1 Introduction

Testing takes much of the time of the software development process, so several efforts have been devoted to automate it. Although initially much progress was done in testing for imperative languages [17,14,12], during the last years several efforts have been devoted to develop test-case generators for declarative languages [9,10,4,2,5], being specially notable the development of Quickcheck [4], a very powerful test-case generator developed for Haskell (and coded in Haskell itself) that has been adapted to imperative languages as Java[1] or C++,[2] thus filling the gap between testing strategies for imperative and declarative languages. To perform testing we use *test cases*, whose definition depends on the programming language being tested, that the programmer uses to examine his program by checking the correctness of these test cases against an oracle, which usually is a specification of the system or the programmer himself.

These test cases are generated following two different strategies: black-box and white-box testing. The former uses a specification language, usually with a formal semantics, to generate the test cases that are later translated to test cases in the implementation language; a semantical relation must be established between the test cases in both languages to determine the correctness of the implementation. Examples of black-box testing are the translation to Java and C++ of the test cases generated by Quickcheck presented above and the language Congu,[3] a framework to create algebraic specifications to test Java programs. On the other hand, white-box testing (also known as glass-box testing) uses the

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

[1] https://quickcheck.dev.java.net/

[2] http://software.legiasoft.com/quickcheck/

[3] http://gloss.di.fc.ul.pt/congu/

T. Mossakowski and H.-J. Kreowski (Eds.): WADT 2010, LNCS 7137, pp. 287–301, 2012.

current implementation of the system to select the most appropriate test cases. Both approaches have been followed in imperative and declarative contexts; black-box testing has been studied in imperative languages [12,13], in declarative languages [4,15,2], and in general contexts [1,11], while white-box testing has been investigated in [17,14] for imperative programming and in [10,9] for declarative programming.

Maude [6] 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 [16], 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 [3], which, in addition to equations, allows the statement of membership axioms characterizing the elements of a sort. 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.

Although the initial aim of the Maude system was to be used as a specification language, the last releases of the system introduce new features such as TCP/IP sockets [6, Chapter 11] and unification [7] that encourage to use Maude as a programming language. Thus, Maude specifications grow in size and complexity, growing consequently the difficulty to debug and analyze them. As part of an ongoing project to debug Maude specifications, we have already implemented a declarative debugger for Maude [19] that allows to debug both wrong and missing answers (incorrect and incomplete results, respectively). Following this line, this paper presents a methodology to test Maude functional modules by using both black-box testing, where the specification language is Maude itself, and white-box testing, where we adapt some strategies already developed for declarative languages and, in addition, present a new strategy to test sort inferences. These techniques have been implemented in Maude and integrated with the the declarative debugger, which allows the user to debug the erroneous test cases at once.

Exploiting the fact that rewriting logic is *reflective* [8], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [6, Chap. 14], that allows access to metalevel entities such as specifications or computations as usual data. Therefore, we are able to generate and check the test cases in Maude itself. The system provides another module, Full Maude [6, Chap. 18], that includes features for parsing, evaluating, and pretty-printing terms, improving the input/output interaction. By extending Full Maude our test-case generator, including its user interactions, is implemented in Maude itself.

Although the Maude metalevel allows an efficient implementation of black-box testing by providing mechanisms to test the correctness of a Maude module against another one and, consequently, its performance should be comparable to Quickcheck's [4], the main drawback of our approach is the term generation: while Quickcheck uses narrowing to obtain the test cases, the Maude's machinery for narrowing is still under development[4] and cannot be used with general Maude theories, so we incrementally

[4] Currently, the narrowing command only supports some theories, does not allows the user to introduce a condition the returned terms are assumed to fulfill, and does not allow incremental searches, that is, we cannot obtain new results without computing again the previous ones.

generate terms and then check whether they are appropriate for testing. Although our black-box testing is less efficient that the one in Quickcheck, we overcome it by providing white-box testing. This testing is based on [10], which adopted some techniques from imperative languages to select a set of terms fulfilling a giving coverage, that is, a number of statements that must be executed to consider the specification tested. We improve these coverage techniques by providing coverage for membership inferences, which takes into account both positive (statements used) and negative (statements that could not be used) information.

The rest of the paper is organized as follows. After briefly introducing Maude functional modules in Section 2, we describe how the terms are generated in Section 3. Our methodology to test Maude functional modules is described in Section 4, while Section 5 outlines the implementation of the tool. Section 6 concludes and outlines some future work.

More information about the test-case generator, related papers, examples, and its source code can be found at `http://maude.sip.ucm.es/testing/`.

2 Maude

Maude [6] 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 [3] that support multiple sorts, subsort relations, equations, and assertions of membership in 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. It is important to note that in membership equational logic sorts are grouped into equivalence classes called *kinds*. For this purpose, two sorts are grouped together in the same equivalence class if and only if they belong to the same connected component.

For our purposes in this work we take advantage of the fact that membership equational logic theories are assumed to be terminating, confluent, and sort decreasing [6]. In this way, we can use a calculus that modifies the usual one shown in [3] by considering that equations are only applied from left to right, which allows us to infer judgments of the form $t \rightarrow_n t'$ and $t :_{ls} s$, introduced in [18] and which indicate, respectively, that the normal form of t is t' and that the least sort of t is s. Models of these judgments, given a signature Σ and a set of equations and membership axioms E, are Σ-term models $\mathcal{T}_{\Sigma/E}$ [16]; see [18] for details in the relation between models and judgments.

Below we present the basics of Maude functional modules and present an example that will be used throughout the rest of the paper.

2.1 Maude Functional Modules

Maude functional modules [6, Chapter 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 (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;[5] memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`). In Maude the user can specify each operator with its own syntax, which can be prefix, postfix, infix, or any "mixfix" combination. This is done by indicating with underscores the places where the arguments appear in the mixfix syntax. Another interesting feature for our tool is that Maude allows both equations and membership axioms to be identified with a label, which is introduced after either the keyword `eq` or `ceq` (`mb` or `cmb` for memberships).[6]

Maude does automatic kind inference from the sorts declared by the user and their subsort relations. Kinds are *not* declared explicitly, and correspond to the connected components of the subsort relation. The kind corresponding to a sort `s` is denoted `[s]`.

For example, we show how to specify lists of natural numbers in the module `LIST` below. We declare the sort `List` for these lists, while the subsort declaration indicates that a single natural number is also a list:

```
fmod LIST is
  pr NAT .

  sort List .
  subsort Nat < List .
```

Lists are built with the operator `nil` for empty lists and the juxtaposition operator `_ _`, which is associative and has `nil` as identity, for bigger lists:

```
  op nil : -> List [ctor] .
  op _ _ : List List -> List [ctor assoc id: nil] .
```

Finally, we define a function `reverse` to reverse a list. Note that this function is buggy: the equation labeled with `rev1` should return `nil` instead of `0`:

```
  var N : Nat .
  var L : List .

  op reverse : List -> List .
  eq [rev1] : reverse(nil) = 0 .
  eq [rev2] : reverse(N L) = reverse(L) N .
endfm
```

3 Term Generation

The tool is able to generate terms by using the constructor information provided by the user. As a first approach, we computed them in a recursive fashion: starting with constants, in each step the new terms were computed from the ones previously obtained.

[5] It is important to note that the equational theory works modulo these axioms.

[6] It is also possible to write this label at the end of the statement as an attribute, although we will always use labels in the way described above.

We can also understand this approach as a grammar, where sorts are non-terminals, constants are terminals, and operators are production rules. After each step, membership axioms were applied to ensure the terms were assigned the appropriate sort. For example, the terms in the `LIST` example above (assuming that the predefined natural numbers have constructors `0` and `s_` for zero and successor) were generated as follows:

1. The constants `0` and `nil` for natural numbers and lists respectively are built.
2. Sort inference is applied. Thus, the term `0` is also considered a term of sort `List`.
3. Nontrivial constructors are applied. The term `s(0)` (pretty printed as `1` by Maude) is built for natural numbers, while the term `0 0` is generated for lists.
4. Steps 2 and 3 are applied until enough terms have been generated.

However, although this method builds up to several thousand terms very quickly, it presents a major drawback: most of the terms are very similar and thus they find the same bugs, while some other problems, that would be found with more complex terms, cannot be found due to the quick growth in the number of terms, that prevents the system from computing more terms once a few steps have been performed (although the user can select the number of steps that are applied in function of the complexity of the constructors, the amount of time required for big bounds greatly limits this option).

To palliate this problem we tried to use the narrowing features available in Maude, using the constructors to distinguish between the different kinds of terms and then trying to fulfill the conditions imposed by the equations and membership axioms. However, these narrowing features do not support general theories and some combinations of attributes cannot be used. Another major problem is that the narrowing command returns the first n solutions but, since it does not receive the condition to be fulfilled but only the lefthand side of the statement to be matched, it is possible to obtain terms that finally cannot be used as term cases, and thus more terms are needed but, with the current format, the system has to recalculate the n previous solutions. We expect this command to be improved soon and thus incorporate this feature to our test generator.

Since narrowing did not improve the tool as expected, for the time being we decided to randomly remove some terms in each iteration of the previous algorithm in order to reduce the number of combinations in the next levels and thus be able to generate bigger terms.[7] Once these terms are computed, we can start the testing process.

4 Testing Maude Functional Modules

We define a test case in Maude as a judgment $t \rightarrow_n t'$ or $t :_{ls} s$, where t and t' are terms and s is a sort. We describe in this section how, starting with the terms generated in the previous section, test cases of this form are generated in Maude and used for testing. First, we show in Section 4.1 how they can be checked against a correct specification; then Section 4.2 describes how to select a set of terms to be inspected by the user depending on different strategies. Finally, Section 4.3 explains how to improve the testing process by allowing the user to select some statements as trusted, preventing the tool from taking them into account when creating this set of terms.

[7] Although this technique does not guarantee that the terms are more suitable for testing, we have checked that it works better in practice.

4.1 Black-Box Testing

Usually, a good approach to testing consists in checking the correctness of several test cases against a specification of the system [4,12]. In our case this relation can be easily established because both the correct specification and the program under test are Maude specifications: assuming that $\mathcal{T}$ is the model of the correct specification and $\mathcal{T}'$ the model of the specification under test, then a test case j fulfills the specification when $\mathcal{T} \models j \iff \mathcal{T}' \models j$. This technique can be efficiently adopted in our prototype thanks to the reflective capabilities of Maude, that allow us to use modules as data. Thus, the tool compares the results obtained from the current specification with respect to the correct one and extracts several pieces of information: the results are different (either they have different constructors or the terms are equal but the inferred sorts are different), the term is not in normal form, or the results are incomparable. Note that it is not necessary to have a correct module with the same functions used in the tested module: if a property over the function to be tested can be defined, it is enough to define this property in a correct module as a constant function that always returns `true`:

```
fmod MY-SPEC is
 ...
endfm
```

```
fmod PROP is
 pr MY-SPEC .
 op prop : ... -> Bool .
 eq prop(...) = ... .
 ...
endfm
```

```
fmod CORRECT is
 pr MY-SPEC .
 op prop : ... -> Bool .
 eq prop(...) = true .
endfm
```

More specifically, we can define the property `revProp` for our lists specification, stating that the reverse of a composition of lists is equal to the composition of the reverses of the lists in inverse order, as follows:

```
fmod REV_LIST is
 pr LIST .
 vars L1 L2 : List .
 var N : Nat .

 op revProp : List List -> Bool .
 eq [prop] : revProp(L1, L2) = reverse(L2) reverse(L1) == reverse(L1 L2) .
endfm
```

Now, we create a new module `CORRECT_LIST` where a function with the same name and profile is defined as the constant `true`, that is, our specification indicates that this property is true:

```
fmod CORRECT_LIST is
  pr LIST .
  vars L1 L2 : List .

  op revProp : List List -> Bool .
  eq revProp(L1, L2) = true .
endfm
```

Now, we can use our tool to check the property. First, we identify which is the correct module, and then we start the testing process with the `test` command:

```
Maude> (correct test module CORRECT_LIST .)
CORRECT_LIST selected as correct module for testing.

Maude> (test in REV_LIST : revProp .)
8464 test cases were generated.
8464 test cases are incorrect with respect to the correct module.
```

Notice that the property never holds. We can ask the tool to show some of the incorrect test cases found, and use the debugger to fix the specification:

```
Maude> (show 1 incorrect .)
The following test cases are incorrect with respect to the correct module:
1. The term test(0,0) has been reduced to false

Maude> (invoke debugger with incorrect test case 1 .)
Declarative debugging of wrong answers started.
...
The buggy node is:
reverse(nil) -> 0
with the associated equation: rev1
```

Complete explanations of this example and the ones in the following sections, including the debugging sessions, are available at `http://maude.sip.ucm.es/testing/`.

4.2 White-Box Testing

Since Maude is a specification language itself, the user does not always have another specification (or is able to define a property) to check the results with. In this case the correctness of the test cases depends on the intended semantics given by the user, and hence a strategy that selects a subset of the generated terms, called *code coverage*, is needed in order to be easily checked by humans. We assume that this intended interpretation is a Σ-term model I corresponding to the model that the user had in mind while writing the specification, and thus we require that, given a test case j and the initial model $\mathcal{T}$ of the specification, $I \models j \iff \mathcal{T} \models j$.

Covering Equations. In [10] some strategies for selecting a coverage in functional languages are described: *global branch coverage* and *function coverage*. The former selects a set of terms such that they cover all branches (both direct and indirect) of the function being tested; the latter tries that, in addition to all branches of the original call to the function, also all branches of all recursive calls to that function have to be considered. Although function coverage is more difficult to apply, it detects more bugs in general than global branch coverage.

In the Maude case, these strategies select a subset of the equations and membership axioms in the specification and then looks for a set of test cases whose inference requires the application of the statements previously selected:

– Global branch coverage tries to find terms that use all the statements potentially used by the function under test (which, of course, also includes the functions in the conditions). That is, the coverage of a function symbol f using this strategy includes all the equations whose lefthand side matches the term $f(x_1,\ldots,x_n)$, where $x_1,\ldots,x_n$ are variables on the kinds specified by the program, and, for each equation $l = r$ if $\bigwedge_{i=1}^{n} t_i = t'_i \wedge \bigwedge_{j=1}^{m} t''_j : s_j$ added to the coverage we must also add all the membership axioms for each sort s_j and the coverage for all the function symbols in the equation, $\mathit{funs}(r) \cup \bigcup_{i=1}^{n} \mathit{funs}(t_i) \cup \mathit{funs}(t'_i) \cup \bigcup_{j=1}^{m} \mathit{funs}(t''_j)$, where

$$\begin{array}{ll} \mathit{funs}(f(t_1,\ldots,t_n)) & = \{f\} \cup \mathit{funs}(t_1) \cup \ldots \cup \mathit{funs}(t_n) \\ \mathit{funs}(a) & = \{a\} \\ \mathit{funs}(X) & = \emptyset \end{array}$$

For example, if we want to test the function `revProp` from our lists specifications, we should cover the equations `prop`, `rev1`, and `rev2`. We can use the tool to test it with the commands:

```
Maude> (global coverage .)
Global Branch Coverage selected

Maude> (test in REV_LIST : revProp .)
1 test cases have to be checked by the user:
    1. The term revProp(nil,0) has been reduced to false

All the statements were covered.

Maude> (invoke debugger with user test case 1 .)
...
```

Actually, reducing this term we cover `prop` (it is the only equation that can be initially used), and `rev1` and `rev2` (by reducing `reverse(nil)` and `reverse(0)` once the first equation has been applied). Once again, we can invoke the debugger to fix the specification by using this term.

– Function coverage checks that all the statements that can be applied for a given function are applied by all the recursive calls (including all those calls in the conditions) in the program. That is, if we try to compute the coverage of a function symbol f with respect to the recursive calls to a function r, then we must find all the appearances of r traversing the specification in the same way we explained for global branch coverage. Once all the reachable calls to r from f have been found, the coverage requires each of them to execute all the equations whose lefthand side matches $r(x_1,\ldots,x_n)$, with $x_1,\ldots,x_n$ variables of the appropriate kind.

  In our lists example, if we want to test `revProp` taking into account the calls to `reverse` we have to distinguish between the four different calls to this function: the first one in `rev2` and three more in `prop`. Each one of these calls must execute both `rev1` and `rev2`. We can use our test-case generator to look for a coverage with the commands:

```
Maude> (function coverage .)
Function Coverage selected

Maude> (test in REV_LIST : revProp wrt reverse .)

2 test cases have to be checked by the user:
    1. The term revProp(0,0) has been reduced to false
    2. The term revProp(nil,nil) has been reduced to false

All calls were covered.

Maude> (invoke debugger with user test case 2 .)
...
```

In that case it is impossible to complete the coverage with only one term, because the calls in `prop` can only execute one of the equations for `reverse` with each test case: with the first test case all these calls execute `rev2`, while with the second one they execute `rev1`. Regarding the recursive call in `rev2`, it executes both equations when reducing `reverse(0 0)` from the first test case. Finally, note that both test cases detect the error and can be used to debug the specification.

Testing memberships. Maude functional modules contain not only equations; as said in the introduction, they also allow the user to define membership axioms and, although initially one could think that the strategies described above can be straightforwardly adapted to work in this case, we soon notice that to apply the axioms (and thus computing an erroneous sort) is as important as *not to apply* them (and thus obtaining a least sort bigger than expected). This problem does not arise with equations, because when a term is not reduced the test generator indicates it is not in normal form by using the constructors, while in this case the system cannot state whether the inferred sort is the least one or just one possible sort of the term.

For this reason, a new coverage strategy that takes into account this information (that we call negative) has been developed: some of the terms in the coverage have to apply all reachable statements *but also* some other terms have to fail, in a special way we will explain below, when trying to apply them. However, some constraints have to be applied to this negative information in order to obtain a realistic coverage strategy:

– It should not consider as negative information trivial failures, which in fact usually occurs when matching the current term with the lefthand side of a membership axiom. For example, assume we are defining the sort `OList` for ordered lists[8] and we state the following axiom:

```
cmb E E' L : OList if E <= E' /\ E' L : OList .
```

Of course, this membership cannot be applied to the test cases `nil` or `0`, but this information is probably unimportant to the user, since even the number of subterms are different.

[8] We prefer “ordered lists” over ”sorted lists” because “sort” is already used to refer to types in this context.

- However, asking the term to match the lefthand side of the axiom can also be too restrictive, since the lefthand side can contain information about the sorts of the terms. For example, we could replace the previous membership axiom for our ordered lists specification with the following one:

  ```
  cmb E OL : OList if E' OL' := OL /\ E <= E' .
  ```

  where the variables `OL` and `OL'` have sort `OList`. In that case, if we only consider terms matching the lefthand side as negative information we are discarding important terms: those that cannot be applied because the membership for `OList` is wrong and thus prevents the term from matching.
- To solve these problems we have decided to consider as valid test cases those that match the lefthand side of the membership axiom *at the kind level*. That is, we consider the variables in the lefthand side as declared in their corresponding kind and then we add the matching at the sort level as the new first condition of the membership axiom.

Besides this problem, another question arises when taking into account the negative information: is it necessary to check that each condition fails? Although in general this approach would detect more errors, with medium examples the computation of the coverage takes too much time to be useful. For this reason we have decided to consider that a membership axiom provides enough negative information when any of its conditions (including the ad hoc condition indicating that the sorts of the terms are correct) fails.

Following the ideas presented previously, assume that we specify ordered lists of natural numbers with:

```
(fmod OLIST is
 pr NAT .

 sorts List OList .
 subsort OList < List .

 op nil : -> OList [ctor] .
 op _:_ : Nat List -> List [ctor] .
 cmb [ol1] : (N : N' : L) : OList if N <= N' /\ N' : L : OList .
endfm)
```

That is, the membership axiom stating that singleton lists are ordered lists is missing. We can look for test cases for this specification with the command:

```
Maude> (test sort in OLIST : OList .)

1 test cases have to be checked by the user:
    1. The term 0 : 0 : nil has least sort List

The following statements were not checked with the given test cases:
ol1
```

```
All the negative information was covered.

Maude> (invoke debugger with user test case 1 .)
...
The buggy node is:
The least sort of 0 : nil is List
Either the operator _:_ needs more membership axioms or the conditions
of the current axioms are not written in the intended way.
```

The tool is not able to apply `ol1` (actually, it cannot be applied without the membership axiom for singleton lists) but it informs the user that it has found a term that, although it matches the lefthand side of one of the memberships, it cannot be finally applied. In fact, the term should have as least sort `OList` instead of `List`, and thus it reveals the failure in our specification.

4.3 Enhancing the Performance

While developing the Maude declarative debugger several buggy specifications, describing all possible errors, were developed. The tool has successfully generated test cases for all the functional examples. However, the main drawback of the tool is its poor performance when facing large specifications, specially when computing the code coverage.

More specifically, although the term generator is able to build up to ten thousand test cases, only the testing with respect to a correct module can use all these test cases, while when computing the coverage it is recommended to select a lower bound for the number of test cases to be checked. The coverage is computed quite slowly (it works with less than one thousand cases), due both to the fact that it performs several operations at the metalevel (see Section 5 for details) and that it computes the minimum coverage, which has exponential complexity.

To improve the performance, a trusting mechanism that hastens the computation of the coverage has been developed: some statements can be pointed out as correct, and thus the tool will omit them when computing the required coverage. The tool offers several options to trust the statements: only labeled statements are taken into account when generating the coverage, specific statements can be trusted, and even complete modules can be selected as correct.

The previous examples are very simple and thus the trusting mechanisms cannot be applied with all their power. We could trust the equation `rev2` with the commands:

```
Maude> (set test select on .)
Debug select is on for test generation.

Maude> (test include REV_LIST .)
Labels prop rev1 rev2 have been added to the coverage.

Maude> (test deselect rev2 .)
Labels rev2 have been excluded from the coverage.
```

The first command initializes the trusting mode, the second one introduces all the labels in the (flattened) module `REV_LIST` as suspicious, and the third one trusts the equation `rev2`. We can use now function coverage with our initial example:

```
Maude> (test in REV_LIST : revProp wrt reverse .)

1 test cases have to be checked by the user:
    1. The term revProp(nil,nil) has been reduced to false

All calls were covered.
```

Note that now one test case is enough to cover all the (non-trusted) equations for `reverse`.

Finally, the tool also allows to trust a specific kind of statement of different modules with the command:

```
(test include/exclude eqs/mbs MODULES .)
```

where `MODULES` is a list of module names separated by spaces.

5 Implementation

We present in this section how the ideas shown in the previous sections have been implemented. This implementation makes extensive use of Maude metalevel [6, Chapter 3], which allows metalevel entities such as terms and modules be used as usual data. Moreover, the test-case generator, as well as the declarative debugger, is implemented on top of Full Maude [6, Chap. 18], which improves the input/output loop provided by the LOOP-MODE [6, Chapter 17] with several parsing features. In this way, we are able to generate the term cases, compute the coverage, check the correctness of the test cases against a correct module, and implement the user interface in Maude itself.

The first phase in the implementation of the tool is the term generator. To build the terms the tool traverses all the operators in the specification looking for those with the `ctor` attribute indicating that they are constructors of the given sort. As explained in Section 3, it first selects the constant constructors (those whose arity is `nil`) and then the rest of operators are used, using as arguments the terms obtained in the previous steps. However, when creating these new terms we must be careful with the operator attributes, that can identify terms that at first sight are different. To take into account these attributes we use the predefined function `metaNormalize`, that computes the normal form of the term with respect to the equational theory consisting of these equational attributes. Finally, after each step we use the predefined function `leastSort` to obtain the least sort of the term and then add it to the set of all its supersorts.

Black-box testing is implemented in a straightforward way; we use the function `metaReduce` in both the correct module and the module under test, and then we check that both the term and the sort correspond. White-box testing is more complicated: starting from the function to be tested, we check all the possible paths in order to keep the reachable statements, in the case of global branch coverage, or the reachable recursive calls, in the case of function coverage. Once the needed coverage has been computed

we execute the test cases obtained in the previous step; however, the usual way of executing a term in a functional module is just obtaining the result, while in our case we need to examine each term to keep the coverage thus far. To do this we use the function `metaMatch` to check whether the current term matches the lefthand side of an equation and fulfills the conditions and, in case the matching succeeds, we apply the obtained substitution to the righthand side, which generates the next term to be examined.

Regarding the interaction with the user, we have extended the internal state of the loop shown in [19] with attributes to keep the type of coverage selected, the trusting information, the test cases, and the type of error detected by each test case (in case we are using black-box testing). With these attributes and the new commands described in this paper we are able to combine the declarative debugger with the test-case generator, which shows the scalability of the system.

6 Concluding Remarks and Ongoing Work

This work is the first step toward developing a test-case generator for Maude specifications. Currently, the tool allows the user to debug functional modules following two different strategies: black box and white box. While the former compares the results obtained in the module under test with those obtained in a correct specification, the latter selects a set of terms in such a way that they fulfill a so called code coverage. In addition to known coverage strategies like global branch and function coverage, that have been adapted to the Maude case, we have designed a membership coverage that takes into account not only the statements applied, but also the memberships that were not applied.

Regarding scalability, we distinguish between the scalability with respect to the complexity of the constructors and with respect to the number of statements. In the first case the tool only scales well for medium-sized specifications, because the number of terms generated for a given sort in each step of the term-generation process depends on the number of terms built for the sorts used as arguments and thus, if several levels are needed to build the sort (i.e., if the sort is complex) then each step is very expensive and only a few can be taken before the system collapses. In the second case, the tool works even for large specifications, since the complexity does not depend on the size of the specification but on the complexity of the function being tested (number of statements/recursive calls); moreover, the trusting mechanisms work better for large (and structured) specifications, since we expect the user to test the imported modules before using them, and thus they can be trusted.

For the reasons sketched above, most of the ongoing work is devoted to improve the performance of the tool. We are now working on the term generator. The narrowing command working on the Maude metalevel is being enhanced to allow consecutive searches in an efficient way (currently, it recomputes the previous results). Using this command we can generate terms, check whether these terms fulfill the conditions of any of the statements under test, and then continue generating terms until the required number of terms have been generated. It will also be required an extension of narrowing to more theories than the currently supported, especially taking membership axioms into account.

The prototype can be improved, first, with new strategies (both new coverage strategies and black-box testing) and, second, by enhancing its performance by providing new trusting mechanisms. We also intend to improve the current coverage strategies: currently, the smallest set of terms fulfilling the selected strategy are presented; however, it could be easier for the user to check a big set of simple terms than a small set of very complex terms. Thus, we are developing different strategies to allow the user to select the most appropriate set of test cases depending on his expertise. We also plan to allow the user to fix some complex values (e.g. tables and arrays which do not change the behavior of the function) in the functions to be tested, so the test-case generator can focus on the rest of parameters. We intend to improve the performance of all these tasks by using a distributed architecture, where each processor is in charge of a specific task while another processor gathers and handles all the information.

Since Maude is a specification language, it would be interesting to use Maude to specify a system and another language to implement it. Currently, this approach is been followed to teach data structures at the Universidad Complutense: the data structures are first specified in Maude and then implemented in C++. To test them a translation from Maude to C++, written by hand for each data structure, is required. The results obtained from this experience will be used to develop translations to other languages.

An extension to system modules is also outlined; since these modules are not required to be either terminating or confluent, the test cases must take into account different information. Probably, a coverage strategy that checks which terms cannot be further rewritten (i.e., provides negative information) will be useful. Finally, the graphical user interface is being updated to connect the test-case generator with the Maude declarative debugger.

Acknowledgments. I thank Sebastian Fischer for his kind explanations of his coverage strategies, Fernando Orejas for his help in preliminary versions of the paper, Ricardo Peña for his useful comments on previous versions of the tool, and Markus Roggenbach for his help with the final version of the paper.

References

1. Bernot, G.: Testing Against Formal Specifications: A Theoretical View. In: Abramsky, S., Maibaum, T.S.E. (eds.) TAPSOFT 1991, CCPSD 1991, and ADC-Talks 1991. LNCS, vol. 494, pp. 99–119. Springer, Heidelberg (1991)
2. Borba, P., Cavalcanti, A., Sampaio, A., Woodcook, J. (eds.): PSSE 2007. LNCS, vol. 6153. Springer, Heidelberg (2010)
3. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science 236, 35–132 (2000)
4. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of Haskell programs. In: ACM SIGPLAN Notices, pp. 268–279. ACM Press (2000)
5. Claessen, K., Smallbone, N., Hughes, J.: QUICKSPEC: Guessing Formal Specifications Using Testing. In: Fraser, G., Gargantini, A. (eds.) TAP 2010. LNCS, vol. 6143, pp. 6–21. Springer, Heidelberg (2010)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Bevilacqua, V., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)

7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual (Version 2.5) (June 2010), http://maude.cs.uiuc.edu/maude2-manual
8. Clavel, M., Meseguer, J., Palomino, M.: 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)
9. Degrave, F., Schrijvers, T., Vanhoof, W.: Automatic generation of test inputs for Mercury. In: 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2008), Valencia, Spain, July 17-18, 2008, Revised Selected Papers, pp. 71–86. Springer, Heidelberg (2009)
10. Fischer, S., Kuchen, H.: Systematic generation of glass-box test cases for functional logic programs. In: Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP 2007, pp. 63–74. ACM Press, New York (2007)
11. Gaudel, M.-C., Le Gall, P.: Testing Data Types Implementations from Algebraic Specifications. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 209–239. Springer, Heidelberg (2008)
12. Hierons, R.M., Bogdanov, K., Bowen, J.P., Rance Cleaveland, J.D., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., Zedan, H.: Using formal specifications to support testing. ACM Computing Surveys 41(2), 1–76 (2009)
13. Koopman, P., Alimarine, A., Tretmans, J., Plasmeijer, R.: GAST: Generic Automated Software Testing. In: Peña, R., Arts, T. (eds.) IFL 2002. LNCS, vol. 2670, pp. 84–100. Springer, Heidelberg (2003)
14. Lembeck, C., Caballero, R., Müller, R.A., Kuchen, H.: Constraint solving for generating glass-box test cases. In: Kuchen, H. (ed.) Proceedings of International Workshop on Functional and (Constraint) Logic Programming (WFLP 2004), pp. 19–32 (2004)
15. Machado, P.D.L.: On Oracles for Interpreting Test Results against Algebraic Specifications. In: Haeberer, A.M. (ed.) AMAST 1998. LNCS, vol. 1548, pp. 502–518. Springer, Heidelberg (1998)
16. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
17. Müller, R.A., Lembeck, C., Kuchen, H.: A symbolic Java virtual machine for test case generation. In: IASTED Conf. on Software Engineering, pp. 365–371 (2004)
18. 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)
19. Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R.: Declarative debugging of rewriting logic specifications. Technical Report SIC-02-10, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid (2010), http://maude.sip.ucm.es/debugging