Using Semantics Specified in Maude to Generate Test Cases*

Adrián Riesco

Facultad de Informática, Universidad Complutense de Madrid, Spain

Abstract. Testing is one of the most important and most time-consuming tasks in the software developing process and thus techniques and systems to generate and check test cases have become crucial. For these reasons, when specifying a prototype of a programming language it may be very useful to have a tool that, beyond testing the semantics of the program, generates test cases for the programs written in the specified language. In this way, we could use the test cases generated in the prototyping stage to check the implementation. To build these prototypes we can use rewriting logic, which has been proposed as a logical framework where other logics can be represented, and as a semantic framework for the specification of languages and systems.

In this paper we propose a technique to generate test cases for programs written in programming languages specified in Maude, although it can be generalized to similar languages. In this way Maude becomes an even more powerful prototyping language providing a test-case generator (in addition to an interpreter of the language). The test cases can be generated in two ways: computing a set of test cases using all the instructions required by a given coverage criterion or trying to disprove a property over the program. This methodology has been implemented in a Maude prototype and its use is described by means of an example.

Keywords: testing, semantics, Maude, coverage, property-based, narrowing.

1 Introduction

Testing is a technique for checking the correctness of programs by means of executing several inputs and studying the obtained results. Testing is one of the most important stages of the software-development process, but it also is a very time-consuming and tedious task, and for this reason several efforts have been devoted to automate it [12,3]. Basically, we can distinguish two different approaches to testing: white-box testing [9,17], that uses the specific statements of the system to generate the most appropriate test cases, and black-box testing [26,10,3], that considers the system as a black box with an unknown structure and where a specification of the system is used to generate the test cases and check their correctness. When using white-box testing we can also distinguish between generating ground test cases that are later executed and using test cases with variables that must be refined by using the program. From the programming-languages prototyping point of view, it would be interesting to generate test cases for

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

A. Roychoudhury and M. D'Souza (Eds.): ICTAC 2012, LNCS 7521, pp. 90–104, 2012.

programs written in the programming language being prototyped, instead of the standard approach which generates test cases for the semantics of the language. That is, if we define, for example, the semantics of Java in a programming language providing a test-case generator, we could use it for testing that the semantics are defined in the appropriate way (e.g., the while loop works properly in general), but we could not use it for testing a program written in Java and executed used this semantics (e.g., does my sorting method work?). The advantages of having this kind of tool are that (i) test cases can be obtained in the prototyping stage, executed, and used again once the real system is implemented, and (ii) since several well-known languages have been already represented in Maude [15], we can generate test cases for them. The technique presented here—which can be adapted to any programming language providing features similar to reflection and narrowing—is, to the best of our knowledge, the first one applying this "meta-level" approach.

We are especially interested in prototypes of programming languages specified in Maude [5], a high-level language and high-performance system supporting both equational and rewriting logic computation. 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 [2], 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. The current version of Maude supports a limited version of narrowing [25], a generalization of term rewriting that allows to execute terms with variables by replacing pattern matching by unification, for some unconditional rewriting logic theories without memberships. As a semantical framework, Maude has been used to specify the semantics of several languages, such as LOTOS [27], CCS [27], or C [7]. These researches, as well as several other efforts to describe a methodology to represent the semantics of programming languages in Maude, led to the *rewriting logic semantics project* [15], which presents a comprehensive compilation of these works, and to the development of K [24], a programming language built upon a continuation-based technique that provides mechanisms (i) to ease language definitions and (ii) to translate these definitions into Maude, allowing the programmer to uses its analysis tools.

In previous works we have presented a tool to generate test cases for Maude modules [19,20]. This kind of tool generates test cases in different ways: they can try to use all the equations, membership axioms, and rules required by a given *coverage criterion*; they can try to falsify a given property; or they can check whether a given Maude system module, the implementation, performs the same actions as another Maude system module, the specification, which has been previously tested and debugged. However, when specifying other programming languages the user could test the Maude specification but not the programs written in that programming language. In this paper we present a methodology for generating test cases for the programming languages specified in Maude. In this way, one of the main features of Maude, providing an interpreter of the language being described for free (obtained because Maude specifications are

executable) is now extended with a test-case generator for the language being specified. This tool computes test cases (i) trying to cover all the statements of the sort indicated by the user (as we will explain later, this is known as *global branch coverage*), what means that he can e.g. try to cover all the assignments in imperative languages or all the equations in functional languages; and (ii) trying to disprove a property over the program. As an extra feature obtained from the coverage approach, we can perform static analysis in the programs written in these languages and check whether all the statements are reachable, that is, if the program contains dead code.

The rest of the paper is organized as follows: Section 2 presents the related work and the differences with our approach. Section 3 introduces Maude and narrowing, while Section 4 describes the methodology followed to test programs whose semantics has been previously specified in Maude. Section 5 shows how this approach has been implemented in a Maude prototype by means of an example. Finally, Section 6 concludes and outlines the future work. The source code of the tool, examples, related papers, and much more information is available at `http://maude.sip.ucm.es/testing/`.

2 Related Work

Different approaches to testing have been proposed in the literature. We present in this section the most similar approaches to ours: testing of imperative languages following a declarative approach (in the sense that they use a methodology initially designed for declarative languages) and testing using symbolic execution approaches (like narrowing). We thus rule out from the picture other interesting approaches like conformance testing [26], which checks that an implementation performs the same actions as a given specification, because it is very different from the ideas presented here, although we consider it an interesting subject of future work. Correspondingly, we do not include the verification of security protocols [13], that symbolically explore the state space trying to find a flaw in the protocol, because they focus on a very specific problem.

An important example of test-case generator initially developed for a functional language that has been extended to imperative languages is Quickcheck [4], a tool developed for Haskell specifications where the programmer writes assertions about logical properties that a function should fulfill; test cases are randomly generated by using the constructors of the data type to test and attempt to falsify these assertions. Note that these test cases are just ground terms used to check whether the properties stated by the user hold. The project, started in 2000, has been extended to generate test cases for several languages such as Java, C++, Erlang, and several others following the same approach. Quickcheck has also inspired other tools like PropEr [18], a test case generator for Erlang, although we will focus our comparison on Quickcheck.

An interesting symbolic approach is applied by Lazy Smallcheck [23] (an improvement of a previous system called SparseCheck), a library for Haskell to test partially-defined values that uses a mechanism similar to narrowing to test whether the system fulfills some requirements. Another way of achieving symbolic execution in a generic way is by considering that the statements in the program under test introduce constraints on the variables, an approach followed by PET (Partial Evaluation-based Test Case Generation) [11], that uses Constraint Logic Programming to generate test cases satisfying some coverages on object-oriented languages (focusing on Java bytecode).

Table 1. Comparison of different test-case generators

	Quickcheck	Lazy Smallcheck	PET	MSTCG
Tested language	Haskell, C++, Java, Erlang, and others	Haskell	Java bytecode	Languages specified in Maude
Type of testing	Property-based	Property-based	Code coverage	Property-based and code coverage
Technique	Random testing	Narrowing	Constraints	Narrowing
Other remarks	Shrinking Best performance (industrial tool)	Shrinking Research tool	Breakpoints GUI More coverages	Shrinking Generic approach Research tool

The comparison between these tools and ours (called MSTCG, from Maude Semantical Test-Case Generator) is outlined in Table 1. Note that we do not include our own test-case generator for Maude specifications [19], because it would share part of the features of Quickcheck and Lazy Smallcheck, and thus it would not add any new information. The table presents the tested language, the focus of the test cases generated (property-based or code coverage), the mechanism used to generate them, and some interesting remarks about them; we discuss in the following the main points of this comparison. Quickcheck is applied to several different languages; however, the tool for testing each language is implemented specifically and thus it is not generic. From the features point of view, it provides property-based testing and, since it is an industrial tool with several heuristics, it presents a better performance than our tool. On the other hand, an advantage of our tool is the computation of test cases fulfilling a coverage criterion, allowing the user to test the specification by checking test cases "by hand" (that is, against his intended interpretation) even when no properties over the specification are stated. Finally, Quickcheck implements a mechanism called shrinking that computes, for a test-case disproving the property, the smaller one (in terms of constructors) that also disproves it. Test cases obtained by using symbolic execution provide a similar result (for both code coverage and property-based testing in our case), due to the fact that they perform the smallest amount of modifications to the initial terms in order to execute the given program; in this way, we are sure that the test cases are the smallest ones. Lazy Smallcheck is very similar to our tool in the sense that both are narrowing-based experimental tools that focus on research rather than in efficiency, and thus they present a similar performance; however, this tool is not generic and only applies property-based testing. Finally, PET is not generic and does not allow the user to state properties over the program. However, it is more mature than MSTCG and presents many advantages: its provides a graphical user interface, which allows the user to see which commands are covered with each test case, put breakpoints in the code that are later used for the tool, and many other options; and offers more coverage strategies than our approach.

Summarizing, an important point of our approach is that it is *generic*, in the sense that any user can define the semantics of a programming language and then generating test cases for its programs; our tool is to the best of our knowledge the first one of this kind. It is also important to note that we provide two different techniques: code coverage and property-based testing, while most tools only provide one of them. The

strong point of our tool reveals its main weaknesses: (i) it is necessary to specify in Maude the semantics of the language and (ii) the performance of the tool is low due to the intensive use of metalevel computation, which takes a great amount of time.

3 Preliminaries

We present in this section the Maude system and its narrowing mechanisms [6].

3.1 Maude

Maude modules are executable rewriting logic specifications. Maude functional modules [5, Chap. 4], introduced with syntax `fmod ... endfm`, are executable membership equational specifications that allow the definition of 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; 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 [5, Chap. 6], introduced with syntax `mod ... endm`, are executable rewrite theories. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`).

We introduce Maude modules with an example showing how to define the evaluation semantics of a simple imperative language; the complete specification of this language is presented in [27]. In the following, we will call this programming language the *object language* to distinguish it from Maude; we will also use this name in general to refer to any programming language specified in Maude. Assume we have defined in a module called `EVALUATION-EXP-EVAL` the syntax of a language with the empty instruction `skip`, assignment, `If` statement, `While` loop, and composition of instructions, all of them of sort `Com`; some simple operations over expressions and Boolean expressions such as addition and equality; and a state, of sort `ST`, mapping variables to values. Using this module we specify the evaluation semantics of this language in `EVALUATION-SEMANTICS`, that first defines a `Program` as a pair of a term of sort `Com` and a state:

```
(mod EVALUATION-SEMANTICS is
  pr EVALUATION-EXP-EVAL .
  op <_,_> : Com ST -> Program .
```

The semantics for the assignment are described by the rule `AsR`, that computes the value of the expression in the condition and then updates the state of the variables by substituting the value in the variable with the one computed in the condition:

```
  crl [AsR] : < X := e, st > => < skip, st[v / X] >
   if < e, st > => v .
```

The rules `IfR1` and `IfR2` describe the behavior of the `If` statement. If the boolean condition `be` reaches the value `T` (which stands for `true`) then we execute the `Then` branch and return the reached state `st'`; otherwise, the `Else` part is executed:

```
  crl [IfR1] : < If be Then C Else C', st > => < skip, st' >
   if < be, st > => T /\
      < C, st > => < skip, st' > .
  crl [IfR2] : < If be Then C Else C', st > => < skip, st' >
   if < be, st > => F /\
      < C', st > => < skip, st' > .
```

Analogously, `WhileR1` and `WhileR2` describe the behavior of the while loop:

```
  crl [WhileR1] : < While be Do C, st > => < skip, st >
   if < be, st > => F .
  crl [WhileR2] : < While be Do C, st > => < skip, st' >
   if < be, st > => T /\
      < C ; (While be Do C), st > => < skip, st' > .
```

Finally, the rule `ComR` combines two different computations, using the state reached in the first one to continue with the second one:

```
  crl [ComR] : < C ; C', st > => < skip, st'' >
   if < C, st > => < skip, st' > /\
      < C', st' > => < skip, st'' > .
endm)
```

Given this semantics, we can now write and execute programs of the form:

```
< If Equal(x, 0) Then y := 0
                 Else y := 1 ;
  If Equal(w, 0) Then z := 0
                 Else z := 1, st:ST >
```

where `st` is a free variable of sort `ST` (the state) that will be instantiated by the testing process. We can see now the differences between testing the Maude specification and testing the imperative program executed by Maude.[1] Assuming that in both Maude and our imperative language the coverage criteria is *global branch coverage* [9], which requires that all the reachable statements (membership, equations, and rules in the Maude case; assignments, `If`, and `While` statements otherwise) are executed, we would obtain the following results: the state `x = 0, w = 1` would cover all the possible Maude rules (`AsR`, `IfR1`, `IfR2`, and `ComR` rules, the rest of the rules are not reachable) but only covers two assignments `y := 0` and `z := 1` (and, obviously, the two `If` statements), that is, we would need another state (e.g. `x = 1, w = 0`) to meet the criterion. It is easy to see that, although the techniques for covering Maude specifications and for programs written in another language previously specified in Maude are similar, the rules that must be applied are different, and thus it is necessary to specify the kind of statements we want to cover, manipulate the object program to "mark" these statements,[2] and indicate for each Maude rule the statements that are executed. We will see the details in Section 4.

[1] In fact, test case generation is even more different between Maude and any other program written in the given object language, because in Maude we would start with a term containing only variables, that is, the initial term would be `< prog:Com, st:ST >`. However, we present the differences over this partially instantiated term to show that even setting the program the required coverage is different.

[2] In this specific example we want to cover assignments, but note that this methodology also works for other languages and even other paradigms. For example, in Haskell-like programs we would want to cover all the cases for all the reachable functions.

3.2 Narrowing

Narrowing [25,8,16] is a generalization of term rewriting that allows free variables in terms and replaces pattern matching by unification in order to reduce these terms. It was first used for solving equational unification problems [22] and then generalized to deal with problems of symbolic reachability. Similarly to rewriting, where at each rewriting step one must choose which subterm of the subject term and which rule of the specification are going to be considered, at each narrowing step one must choose which subterm of the subject term, which rule of the specification, and which instantiation of the variables of the subject term and the rule's lefthand side are going to be considered. The difference between a rewriting step and a narrowing step is that in both cases we use a rewrite rule $l \Rightarrow r$ to rewrite t at a position p, but narrowing unifies the lefthand side l and the chosen subject term t before actually performing the rewriting step, while in rewriting this term must be an instance of l (i.e., only matching is required). Using this narrowing approach, we can obtain a substitution that, applied to an initial term that contains variables, generates the most general term that can apply the traversed rules.

We denote by $t \leadsto^{\sigma}_{\theta} t'$, with $\sigma = q_1; \ldots; q_n$ a sequence of labels, the succession of narrowing steps applying (in the given order) the statements $q_1; \ldots; q_n$ that leads from the initial term t (possibly with variables) to the term t', and where θ is the substitution used by this sequence, which results from the composition of the substitutions obtained in each narrowing step. The latest version of Maude includes an implementation of narrowing for unconditional free, C, AC, or ACU theories in Full Maude [6]. We have improved this implementation by using the techniques described in [20] to use membership axioms and conditional statements.

4 Using Semantics to Generate Test Cases

We describe in this section the methodology to test programs whose semantics has been previously specified in Maude. It consists of three steps: identifying the appropriate sort of statements that must be covered, manipulating the program to mark these statements, and modifying the Maude rules to indicate the statements applied by each of them. It is important to state first that the coverage criterion applied to all the programs is global branch coverage, that tries to apply all the reachable statements, as illustrated above. Also note that some of the steps explained in this section will be later performed automatically, as shown in Section 5. Note that this approach is only required for coverage strategies, and hence property checking will be explained later.

The first step relies on the user to indicate the sort of statements that he wants to cover. In our example, the user should mark `Com`, the sort for all the possible instructions (`skip`, assignment, `If` conditional, and `While` loop) as this sort. The second step is automatically performed by the tool, and consists of "marking" the given program and each Maude rule with a unique identifier distinguishing the different statements. For example, the program presented at the end of Section 3.1 would be marked as follows:[3]

[3] In fact, an extra label for the whole program, which will be executed by an application of the `extraInfo-ComR` rule, would also be computed. We do not show it here for the sake of readability.

```
< 1If Equal(x, 0)
  Then 2y := 0
  Else 3y := 1 ;
  4If Equal(w, 0)
  Then 5z := 0
  Else 6z := 1, st:ST >
```

where the two conditional statements and the four assignments have been labeled with natural numbers indicating that they are statements of the sort given by the user. The test-case generator will look for states executing all of these commands. In the same way, rules have to be modified to deal with this kind of terms by adding variables of sort `Nat` (the predefined sort for natural numbers) to each statement of the given sort. In this way, the rule `IfR1` is extended as follows:

```
crl [extraInfo-IfR1] : <n If be Then n' C Else n'' C', st> => <skip, st'>
 if < be, st > => T /\
    < n' C, st > => < skip, st' > .
```

Note that extended rules are renamed by using the prefix `extraInfo`. We can see now in the `extraInfo-IfR1` rule above that it contains several labeled statements, but not all of them are executed by the rule: the `Else` branch is never executed, the `Then` branch is executed in the conditions (and thus the execution of this statement is in charge of the the assignment rule, which will indicate that this statement has been used), and finally the whole `If` statement *is* executed. As we will show in the next section, we provide a semi-automatic approach to relate rules and executed statements: the tool computes a mapping following a fixed strategy, that the user can check and edit it if required. Finally, note that the initial term must also be provided by the user. It must contain at least one variable, indicating the values of the variables through the execution of the program, which is instantiated to generate the test cases.

We can use now narrowing to our labeled initial term. Assuming that `initial` is the labeled term shown above, we can apply the following narrowing step:

$$\texttt{initial} \leadsto^{\texttt{extraInfo-ComR}} \texttt{< skip, x = 0 y = 0 w = 0 z = 0 >}$$

after applying the statements 1, 2, 4, and 5, where `extraInfo-ComR` is a rule that defines the composition of statements by execution them in the conditions and then putting the result in the righthand side of the rule. However, this step does not clarify the process due to the fact that all the important steps are applied in the rewrite conditions. For this reason, let's see how the first `If` statement is executed (note here the importance of transmitting the label information to the conditions, which requires a careful rule transformation) by using a narrowing step with the rule `extraInfo-IfR1` shown above:

$$\texttt{< 1If Equal(x, 0) Then 2y := 0 Else 3y := 1, st >} \leadsto^{\texttt{IfR1}} \texttt{< skip, x = 0 st' >}$$

It is important to see that the mapping between rules and applied statements computed by the tool indicates that `extraInfo-IfR1` applies (covers) the statement labeled with *n* in the transformed rule above, which refers to the complete `If` statement. In this case *n* is bound to 1, and thus this rule is executing the first `If` statement. Again, most of the information is developed in the rewrite conditions. Although we have not shown

the rule for the equality, it is easy to see that it requires both values in `Equal` to be equal to return `T`. Hence, in this step we really see how narrowing works, since thus far all the rules have been applied by using matching instead of unification. In this case, unification requires the state to contain the variable `x` mapped to the value `0`, while another state `st'` remains:

$$\texttt{< Equal(x, 0), st >} \leadsto^{\texttt{EqR1}}_{\texttt{st} \mapsto \texttt{x = 0 st'}} \texttt{< T, x = 0 st' >}$$

where `EqR1` is the rule in charge of the positive case of `Equal`; note that this rule has not been modified with respect to the original module because it does not contain statements of sort `Com`, and thus its label does not use the `extraInfo` prefix. Once this first rewrite condition of `extraInfo-IfR1` holds, we try to fulfill to second one, that contains the statement labeled with 2 in the initial term. Before using narrowing, the substitution obtained in the previous step is applied:

$$\texttt{< }^{2}\texttt{y := 0, x = 0 st' >} \leadsto^{\texttt{AsR}} \texttt{< skip, x = 0 y = 0 st'' >}$$

where the new state requires a simple application of narrowing to update the state that we can omit. The main points in this step, which concludes the execution of the first `If` statement, is that the statement labeled with 2 is executed and `y = 0` added to the state. The second `If` statement can be executed in a similar way to obtain the final state `x = 0 y = 0 w = 0 z = 0`. Note that this is the *final* state reached after executing the statements 1, 2, 4, and 5, and thus we need now to compute the *initial* one. For this reason, our tool also returns the initial states, which consists of the term introduced by the user with the variables instantiated with the substitution computed during the narrowing process, in this case it would be $\texttt{st} \mapsto \texttt{x = 0 y = 0 w = 0 z = 0}$.

Given the explanation above for the variables, it may seem strange to have the variables `y` and `z` with value `0`, since these variables may contain any value and it does not affect the executed statements, and thus it deserves a careful explanation. In fact, the rule in charge of updating the state checks whether the variable is already there. If it is not in the state, what would force the state to be `empty`, then it is added and it would not appear in the initial state, but in this case the next statements cannot be applied because the state does not contain `y` and the process cannot continue. For this reason, we (as well as the tool in the next section) have used the rule that forces the state to contain the variable. Thus, at the end of the process, the term representing the initial state will contain a value of the form `y = N:Nat`, with `N:Nat` a fresh variable. This variable is instantiated, to ease the readability, with the smaller term of the appropriate sort, which is the constant `0`. The same happens with the remaining variable of sort `ST`, that would be named `st''''`, which is substituted with the identity element for states, `empty`, and thus disappears.

5 Maude Prototype

We briefly present in this section the Maude prototype and its main commands by using the example in the previous sections. Much more information about the prototype can be found at `http://maude.sip.ucm.es/testing/`.

5.1 Code Coverage

Once all the modules have been introduced in Full Maude and the tool has been started, we can indicate the module where testing must take place and the sort of the statements that we want to cover with the following commands:

```
Maude> (semantics module EVALUATION-SEMANTICS .)
Module EVALUATION-SEMANTICS selected for semantics testing.
Maude> (set sort statements Com .)
Sort Com selected as sort of statements.
```

Once these commands are introduced the tool manipulates the rules in the (flattened) `EVALUATION-SEMANTICS` module as described in the previous section. We can display these modified rules with the command `(show semantics rules .)`, but it is worth seeing the map between the rules and the executed statements. The tool follows a simple strategy to generate this map that consists of selecting as executed statement the first term of the given sort found (i) in the lefthand side of the rule, traversed following a breadth-first search in the tree representing the term; or (ii) in the conditions if the lefthand side does not contain a term of this sort. Note that, although this strategy is quite simple, it works very well in practice because, when several statements appear in a term, the outermost is the one usually applied (e.g. to direct the execution of the inner ones). The command in charge of showing this information is:

```
Maude> (show applied statements .)
The rule extraInfo-AsR :
 crl < stmntIndx(X:Var := e:Exp,V$#0:Nat),st:ST >
  => < skip,st:ST[v:Num / X:Var]>
  if < e:Exp,st:ST > => v:Num .
  applies the statement
  X:Var := e:Exp identified by the variable V$#0
...
```

which shows for each rule the associated statement, as in the `extraInfo-AsR` shown above, where the new variables have the form `V$#` to avoid clashes with other variables in the rule and the operator `stmntIndx` generates pairs of the given sort and variables. When the tool fails to associate the appropriate information to a rule, the following commands can be used:

```
Maude> (rule Q is not associated to any statement .)
The mapping for rule Q has been updated.
Maude> (rule Q is associated to VL .)
The mapping for rule Q has been updated.
```

where `Q` is a rule label and `VL` is the list of variables identifying the applied statements. We can now introduce the program we want to test with the following command:

```
Maude> (object program init .)
Object program introduced.
```

where `init` has the form indicated in the previous section (a term with variables instead of state). It will be reduced by using equations before labeling to allow the user to use constants instead of big terms in the command line. Assuming that `init` is the program described at the end of Section 3.1 and it is labeled as shown in Section 4 (which

can be checked with the command (`show object program .`)), the tool can start the test generation process. However, we may be interested in only covering some of the statements of the initial coverage (e.g., because we want to know whether a specific one, such as an exception, is reached). The user can modify the statements in the coverage with:

```
Maude> (statements in coverage NL .)
The required coverage has been updated.
```

where `NL` is a list of natural numbers indicating the statements to be covered. When all the options have been set, we can start the testing process. The current version of the tool only supports, as explained in the previous section, the global branch coverage strategy, and thus the following result is obtained:

```
Maude> (start semantics testing .)
2 test cases must be checked by the user:
  The program reaches the state < skip, x = 0 y = 0 w = 0 z = 0 >
starting with the substitution st:ST |-> x = 0 y = 0 w = 0 z = 0
and covers the statements 1, 2, 4, 5
  The program reaches the state < skip, x = 1 y = 1 w = 1 z = 1 >
starting with the substitution st:ST |-> x = 1 y = 0 w = 1 z = 0
and covers the statements 1, 3, 4, 6
All the statements were covered
```

Where the variables `y` and `z` appear in the initial state for the reasons given in the previous section. The user would be now in charge of checking whether the reached states are the expected ones. Remember that an important idea behind this tool is to use the generated test cases during the prototyping stage to test programs written in the programming language prototyped in Maude once it is implemented in other language. In this case, we can introduce the values for the variables in the "real" language and then check that the results obtained in both cases are equivalent.

5.2 Property-Driven Test-Case Generation

Another very useful way of testing a program is by defining a property and then trying to find a state where the *negation* of the property holds, that is, where the property does not hold. This generic scheme has been studied in [5, Chapter 12] when using the Maude `search` command to check invariants; we apply the same idea here with *symbolic* search. Note that, since we are just trying to check if the property holds in all the reachable states we are not concerned about coverages, and thus the module transformations presented above are not required, although the underlying narrowing mechanism remains unchanged. It is also important to note that, as explained below, we take conditions into account when performing this search, thus making this approach much more powerful than the current narrowing search available in Maude.

The current command is an improvement of a previous command used in [20], where we provided a command to check whether a property holds in a Maude specification providing the name of the function we wanted to test, when dealing with functional modules, or the sort under test, when working with system modules. We have modified this command to accept terms partially instantiated that, as we have seen above, stand in our case for programs written in some programming language specified in Maude. For

the sake of example, we can state a very simple property over the program described in the previous sections. We can define a function `allEq` that checks whether all the variables in a state contain the same values. Since we use the negation of the property, what we are really checking is that all the reachable states contain at least two different values, which is incorrect, as we have seen in Section 4. The command for this kind of testing is:

```
Maude> (semantics property reaches < skip, st':ST > s.t. allEq(st':ST) .)
  The property does not hold.
  The program reaches the state < skip, x = 0 y = 0 w = 0 z = 0 >
starting with the substitution st:ST |-> x = 0 y = 0 w = 0 z = 0
```

where the command mimics the standard `search` command available in Maude and thus requires a pattern to wrap the reached state and a condition over that pattern;[4] note that the initial state is introduced with the command described above for the object program and thus it is not necessary. An interesting approach using this pattern would consist of trying to match part of the program, for example related to loops, and then check that some properties hold before and after the loop. In our case, the pattern is `< skip, st':ST >`, indicating that the program has finished, while the condition is `allEq(st':ST)`. As expected, the tool has found a counterexample proving that the property does not hold, showing the initial and final states.

Observe that, in this kind of analysis it is not necessary to extend the test case (if it exists) to the implementation. We have proved the program is erroneous and must be modified; once it is fixed and the property holds, we could use global branch coverage to test all its possible branches. It is also important to note that, as the rest of testing approaches, the ones used by our tool are not complete, in the sense that the program may contain a bug and it cannot be found. To palliate this problem the tool uses a bound in the number of steps that can be modified by the user with `(set narrowing depth N .)`, with `N` a natural number, in order to traverse a bigger search space.

5.3 Implementation Notes

Exploiting the fact that rewriting logic is reflective, a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [5, Chap. 14], a feature that makes Maude remarkably extensible 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. In this way, we can manipulate the modules introduced by the user, direct the narrowing process, and implement the input/output interactions in Maude itself. More specifically, we are interested in the `metaNarrowSearchPath` function that, given a term and a bound on the number of narrowing steps, returns all the possible paths starting from this term, the used substitutions, and the applied rules. We use this command to perform a breadth-first search of the state space. Moreover, the test-case generator is implemented on top of Full Maude [5, Chap. 18], a tool completely written in Maude which includes features for parsing, storing modules, and pretty-printing

[4] Obviously, if the pattern contains enough information about the reached state the condition can be `nil`.

terms, improving the input/output interaction. Although conceptually our tool uses two levels of reflection (the meta-represented Maude module standing for the semantics of a programming language and the program written in this language, which is used to generate the test cases; this second level is the novelty of our approach), at the implementation level one reflection level is enough.

It is worth mentioning some important implementation issues. First, and following the module transformation described in [20], we consider that equations are oriented from left to right and thus can be transformed into rules (analogously transforming equational conditions into rewrite conditions with fresh variables), allowing the narrowing process to use equations to refine the variables. Moreover, the current narrowing commands available in Maude only work with unconditional rules, which prevents us from using this kind of rules. Since this constraint would exclude most of the language specifications developed in Maude thus far, we use a mechanism [20] that checks by using narrowing, for each narrowing step, the rewrite conditions (remember that, as said above, equational conditions become rewrite conditions), and adds the obtained substitution to the one obtained with the body of the rule. In this way we can extend the substitution obtained for the rule with the extra variables that appear in rewrite conditions and may appear, as shown in the example at the end of Section 4, in the righthand side of the rule. This extension makes the search command used in the previous section more powerful than the current narrowing search available in Maude. Finally, we provide a rule-based definition for some predefined functions, such as _<_ for natural numbers. They follow the standard definitions distinguishing the different constructors (0 and successor) of these functions and allow us to apply our technique to a much wider range of programming languages, that use most of these functions in conditions.

6 Concluding Remarks and Ongoing Work

We have presented a methodology to test programs written in any programming language whose semantics has been previously defined. In this way we propose a novel way to generate test-cases using a meta-level approach, instead of just testing the given semantics. We use this approach to improve the Maude features as prototyping language because it provides now, in addition to an interpreter of the language being specified, a test-case generator for programs written in that language. These test cases can be also used after prototyping to check that the implementation follows the specification or to detect dead code. The process to accomplish this generic coverage is semi-automatic: the user is in charge of indicating the sort of statements he wants to cover and of checking that the rules execute the statements inferred by the tool, modifying them if needed; the rest of module manipulations is automatically performed by the tool. Finally, we also allow the user to check whether a property holds in the program.

The work presented in this paper offers a good basis for potential extensions and enrichment that can improve its usability and generality. We are currently working on a generic way to modify the output generated by the tool; our goal is to generate JUnit-like [1] output for each programming language, i.e., an executable program with assertions written in the object language that allows the user to really test its program in the original language. This idea would be an important step to perform testing

against the specification [10] in a natural and automatic way. Similarly, it is interesting to study how to combine conformance testing [26] with our approach, that is, how to check that transitions used in the specification are replicated in the implementation. Moreover, expanding our approach to deal with program definitions specified in the K framework [24] would also be very useful. This option is not available in the current version of the tool due to the internal transformations of K, which modifies the form of the rewrite rules. We also want to provide more coverage criteria in addition to the global branch coverage criterion presented in this work. However, we require generic criteria, that is, criteria that can be applied independently of the paradigm of the object language, which makes the implementation of these criteria far more complicated than for specific programming languages. It would also be interesting to see how the random testing approach, successfully followed in other tools like Quickcheck [4], works here. Furthermore, we plan to extend our declarative debugger [21], that currently presents the same problem as the previous version of the test-case generator: it can only debug Maude specifications, but not the object language. Our aim is to develop a universal declarative debugger that takes as input a program in any object language specified in Maude, applies the test-case generator presented in this work and, if any of the test cases reveal an error, debug it with this new debugger. Finally, we want to study how the test-case generator works for languages with parallelism and synchronization. We expect the narrowing mechanism to traverse all the possible paths and check, following the ideas of property-based testing, whether the program fulfills some requirements.

References

1. Beck, K., Gamma, E.: Test-infected: programmers love writing tests, pp. 357–376. Cambridge University Press (2000)
2. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science 236, 35–132 (2000)
3. Cartaxo, E.G., Neto, F.G.O., Machado, P.D.L.: Test case generation by means of UML sequence diagrams and labeled transition systems. In: Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, SMC 2007, pp. 1292–1297. IEEE (2007)
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. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual (Version 2.6) (January 2011), `http://maude.cs.uiuc.edu/maude2-manual`
7. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: Proceedings of the 39th Symposium on Principles of Programming Languages, POPL 2012, pp. 533–544. ACM (2012)
8. Fay, M.J.: First-order unification in an equational theory. In: Joyner, W.H. (ed.) Proceedings of the 4th Workshop on Automated Deduction, pp. 161–167. Academic Press (1979)
9. 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 (2007)

10. Gaudel, M.-C.: Software Testing Based on Formal Specification. In: Borba, P., Cavalcanti, A., Sampaio, A., Woodcook, J. (eds.) PSSE 2007. LNCS, vol. 6153, pp. 215–242. Springer, Heidelberg (2010)
11. Gomez-Zamalloa, M., Albert, E., Puebla, G.: Test case generation for object-oriented imperative languages in CLP. Theory and Practice of Logic Programming 10, 659–674 (2010)
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. Meadows, C.: Applying formal methods to the analysis of a key management protocol. Journal of Computer Security 1 (1992)
14. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
15. Meseguer, J., Roşu, G.: The rewriting logic semantics project. Theoretical Computer Science 373(3), 213–237 (2007)
16. Middeldorp, A., Hamoen, E.: Counterexamples to Completeness Results for Basic Narrowing. In: Kirchner, H., Levi, G. (eds.) ALP 1992. LNCS, vol. 632, pp. 244–258. Springer, Heidelberg (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. Papadakis, M., Sagonas, K.: A PropEr integration of types and function specifications with property-based testing. In: Proceedings of the 2011 ACM SIGPLAN Erlang Workshop, pp. 39–50. ACM Press (2011)
19. Riesco, A.: Test-Case Generation for Maude Functional Modules. In: Mossakowski, T., Kreowski, H.-J. (eds.) WADT 2010. LNCS, vol. 7137, pp. 287–301. Springer, Heidelberg (2012)
20. Riesco, A.: Using Narrowing to Test Maude Specifications. In: Durán, F. (ed.) Proceedings of the 9th International Workshop on Rewriting Logic and its Applications, WRLA 2012. LNCS. Springer (to appear, 2012)
21. Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R.: Declarative debugging of rewriting logic specifications. Journal of Logic and Algebraic Programming (to appear, 2012)
22. Robinson, J.A.: A machine-oriented logic based on the resolution principle. Journal of the ACM 12(1), 23–41 (1965)
23. Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In: Gill, A. (ed.) Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, pp. 37–48. ACM (2008)
24. Şerbănuţă, T., Ştefănescu, G., Roşu, G.: Defining and Executing P Systems with Structured Data in K. In: Corne, D.W., Frisco, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2008. LNCS, vol. 5391, pp. 374–393. Springer, Heidelberg (2009)
25. Slagle, J.R.: Automated theorem-proving for theories with simplifiers, commutativity and associativity. Journal of the ACM 21(4), 622–642 (1974)
26. Tretmans, J.: Conformance testing with labelled transition systems: Implementation relations and test generation. Computer Networks and ISDN Systems 29(1), 49–79 (1996)
27. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. Journal of Logic and Algebraic Programming 67, 226–293 (2006)