

A tool for testing data type implementations from Maude algebraic specifications

Isabel Pita^{1,2}

*Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Madrid, España*

Adrián Riesco^{1,3}

*Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Madrid, España*

Abstract

This paper presents a tool for testing data types implemented in C++ against formal specifications written in Maude. Maude is a formal specification language based on rewriting logic that allows the specification of abstract data types in a clear and concise manner. Moreover, Maude specifications are executable, which provides two advantages: firstly, we can test our specifications and, secondly, we can obtain the results of the test cases automatically.

We focus our test cases on the correctness of the obtained data values, therefore they are generated from the Maude specification based on the data type constructors and the corresponding membership axioms. On the other hand, the observation of the implementation under test is done for each data type through explicit methods defined by the user. The tool is fully integrated in the Eclipse environment and is platform-independent. We have developed an Eclipse plug-in that calls the Maude system to generate the test cases and then translates them into a sequence of C++ instructions. The C++ instructions are compiled and executed, and the results are compared with the results from the specification.

The tool is being used during this academic year by the Computer Science students at the Universidad Complutense de Madrid in the data types course. They have tested typical data type implementations, like stacks, lists, and binary search trees, as well as other data types based on them. The experience is being very useful as it allows the students to test their implementations and correct their errors.

¹ Research supported by MICINN Spanish project *DESAFIOS10* (TIN2009-14599-C03-01) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC-1465).

² Email: ipandreu@sip.ucm.es

³ Email: ariesco@fdi.ucm.es

1 Introduction

This paper presents a tool for testing abstract data type (ADT) implementations against formal algebraic specifications. Specifications are written in Maude [3], which is a formal specification language based on rewriting logic that allows the specification of abstract data types in a clear and concise manner. Rewriting logic can be parameterized by different equational logics; in the case of Maude the logic is membership equational logic. This logic allows, in addition to equations, the statement of membership axioms characterizing the elements of a sort, which is very useful to define data types such as sorted lists, search trees, etc., that require a complex characterization of their properties beyond the definition of their constructors. In [8], Martí Oliet, Verdejo, and Palomino present equational specifications of a series of typical data structures in Maude including advanced ones such as AVL and 2-3-4 trees. Maude also provides several tools that help in the analysis of the correctness of specifications, like the *Maude termination tool* [3, Chap. 21.1.2], the *Church-Rosser checker* [3, Chap. 21.1.3], and the *Maude debugger and testing tool* [13,12].

The C++ testing tool is designed for helping Computer Science students with implementation of data structures. In fact, it has been used during this academic year, 2010-11, at the Universidad Complutense de Madrid (UCM) in a data type structures course, which motivated the use of the C++ language as implementation language. The tool is fully integrated in the Eclipse environment,⁴ which is platform-independent and provides environments, defined by plugins, for Maude specifications and C++ implementations. The students write, compile, and execute their specifications in the same environment in which they implement the ADTs, generate the test cases, and prove them.

Algebraic specifications define the ADT behavior using constructor functions, that create or initialize the data elements, and other functions, that operate on the data types. Currently the testing tool requires at least one specification constructor to be implemented by a C++ object constructor, while the other constructors may be implemented by public methods. Methods are tested one by one, since we do not generate test cases that use more than one public method. The tool, documentation, and examples are available at <http://maude.sip.ucm.es/maudeADTesting/>.

There has been much work in the area of software testing based on algebraic specifications from the 80s and 90s. These approaches mainly use algebraic specifications to help on the generation of the test sets. They focus on finding the conditions for constructing an ideal exhaustive test suite and on how to select practicable test cases from it. A pioneering work by Gannon, McMullin, and Hamlet is reported in [5]. More recent studies have focused on the so

⁴ <http://www.eclipse.org/>

called oracle problem, that is, whether a decision procedure can be defined for interpreting the results of tests according to a formal specification [7]. Gaudel and Le Gall present a good compilation of the work done so far in [6].

Our work has been inspired by the QuickCheck tool [2] designed by Claessen and Hughes, and its re-implementation for Erlang.⁵ QuickCheck was first designed for testing Haskell programs, although its extension to Erlang allows testing C implementations from Erlang specifications. The QuickCheck test case generator is random, while we build our testing cases incrementally from the ADT specification constructors. Another testing tool for algebraic specifications is HOL-TestGen, which is based on the Isabelle/HOL theorem prover [1].

The rest of the paper is organized as follows: Section 2 presents how to define a specification in Maude and how to use the Eclipse environment to compile and test it; Section 3 shows how to generate the test cases and how to use them to test the ADT implementations. Section 4 explains the design of the tool, Section 5 summarizes the experience of the students in using the tool, and finally Section 6 concludes and explains the improvements to be made to the tool based on the students experience.

2 Data type specification in Maude

Data types are specified in Maude *functional modules*, which correspond to equational theories in membership equational logic [3]. Specifically we use a *Core Maude* extension called *Full Maude* [3, Chap. 18], since its syntax is almost equal to that used in *Core Maude* and we found very convenient to keep the ADT modules in the *Full Maude* database for the test cases generation. Figure 1 presents a specification of the ADT *stack* in the module **STACK**. The module starts with the keyword `fmod`, followed by the module name, an optional parameter declaration (in our case by the theory **TRIV**, that we will explain below, with the parameter **X**), and the keyword `is`. Then, other modules can be included. Types are declared by means of the keywords `sort` or `sorts`, as the declaration for **Stack{X}** in the example. There is an inclusion relation between types, which is described by means of `subsort` declarations, as shown in the example to specify that a nonempty stack, of type **NeStack{X}**, is a specific case of **stack**, of type **Stack{X}**. Then, each operator, introduced by means of the keyword `op`, is declared together with the sorts of its arguments and the sort of its result; for example, the operation `pop` in the example has an element of type **NeStack{X}** and returns an element of type **Stack{X}**. Also note that we use the attribute `ctor` to designate the constructors of the data type; it is used to generate the test cases.

⁵ <http://www.quviq.com/>

```

(fmod STACK{X :: TRIV} is
  sort Stack{X} .
  sort NeStack{X} .
  subsort NeStack{X} < Stack{X} .
  op empty : -> Stack{X} [ctor] .
  op push : X$Elt Stack{X} -> NeStack{X} [ctor] .
  op pop : NeStack{X} -> Stack{X} .
  op top : NeStack{X} -> X$Elt .
  op isEmpty? : Stack{X} -> Bool .
  var P : Stack{X} .
  var E : X$Elt .
  eq [pop1] : pop(push(E,P)) = P .
  eq [top1] : top(push(E,P)) = E .
  eq [isEmpty1] : isEmpty?(empty) = true .
  eq [isEmpty2] : isEmpty?(push(E,P)) = false .
endfm)

(fth TRIV is
  sort Elt .
endfth)

(view vInt from TRIV to INT is
  sort Elt to Int .
endv)

(fmod INT-STACK is
  including STACK{vInt} .
endfm)

```

Fig. 1. Algebraic specification of generic stacks and integer stacks in *Maude*

With typed variables and operators, we can build terms in the usual way. A given term can have many different sorts, because of subsorting and overloading but, under some easy-to-satisfy requirements, a term has a least sort. Terms are used to form:

- membership assertions $t : s$ (introduced with keyword `mb`), stating that the term t has sort s , and
- equations $t = t'$ (introduced with keyword `eq`), stating that t and t' are equal.

Both memberships and equations can be conditional, with respective keywords `cmb` and `ceq`. Conditions are formed by a conjunction of equations and memberships or using the Boolean connectives `and`, `or`, `not`. They can also have a label, which is written enclosed in brackets after the `eq` keyword. The use of memberships is illustrated in the *search tree* specification of Figure 2

Parameterized data types use theories to specify the requirements that the parameter must satisfy. Maude provides some predefined theories that define typical requirements, like the existence of a total order over elements of a given sort, in the `STRICT-WEAK-ORDER` theory (see Figure 2), or just the existence of a sort, in the `TRIV` theory (see Figure 1). This theory requires the existence of a sort `Elt`, that is qualified with the name X of the parameter as `X$Elt`, and that is used to implement generic stacks. The way to express the instantiation of a parameterized module, and thus state the specific sort mapped to `Elt` in our case, is by means of views. An integer instantiation of the `STACK` module is shown in module `INT-STACK` of Figure 1. We refer the reader to [3] for the concrete syntax of Maude theories and views.

The Maude system is available for *Linux* and *Mac OS X* at <http://maude.cs.uiuc.edu>. It is available for *Windows* at <http://moment.dsic.upv.es>.

Maude specifications can be executed under *Eclipse* [9] by means of special *plugins* [11]. This environment facilitates the usage of Maude by integrating the text editor with the execution commands of the system. Figure 2 shows

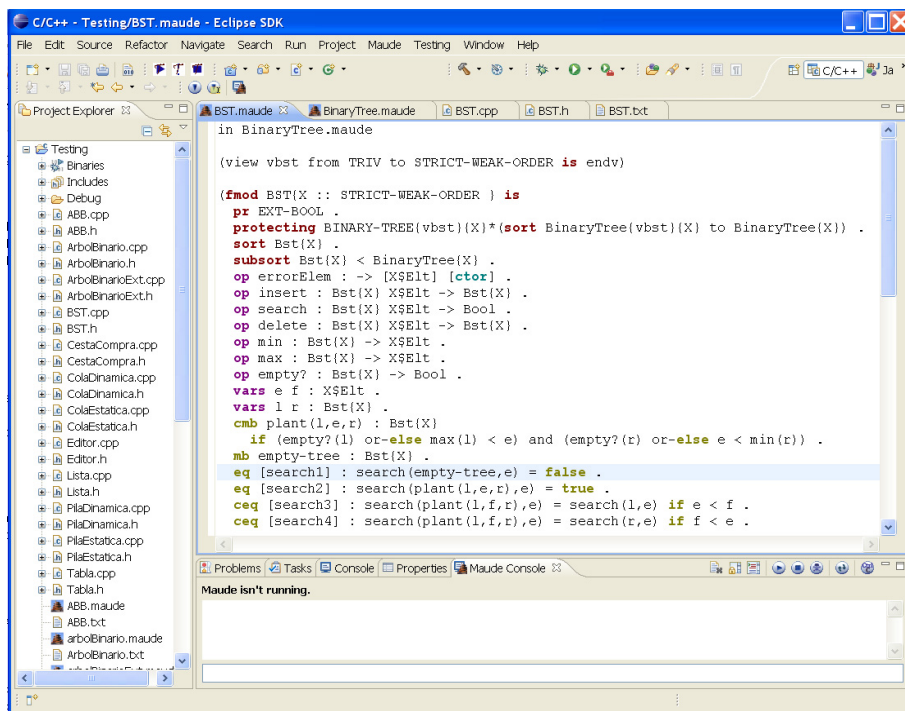



Fig. 2. Eclipse window

an Eclipse window: on the left the defined projects are displayed; the central part shows the editor; below there is the control panel that shows the result of the action and below it the command line is located. Other windows that allow the definition of different system options and debugging can be opened.

The user writes the specification in a Maude file using the Eclipse editor and saves it in an existing Eclipse project. Due to testing restrictions, the file must contain all the user modules used in the specifications (the `in` Maude command is allowed), and the instantiated module to be tested should be the last one. The specification is then executed by opening the Maude console and initializing Maude by clicking on the right button of the console.

When a Maude file is opened in the editor window, two buttons are displayed in the menu bar . The first one is used to send the file to the system. Once sent, the file is compiled and the system reports the syntax errors in the Maude console. Then, the user can reduce terms by using the command line or by writing them on the file and sending them with the second button (*send selection*) to the system. Only instantiated terms can be reduced. The result is shown in the Maude console.




As part of an ongoing work to test and debug Maude specifications, a declarative debugger that allows the user to debug both wrong and missing answers has been implemented [13]. This debugger has been extended with a test-case generator for Maude functional modules in [12], which allows the user to generate, following different strategies, a set of test cases fulfilling

a given coverage and whose correctness will be checked by the user, or to check the correctness of a Maude specification against another specification, which is known to be correct. In this way, we could test the functions of the **STACK** specification in Figure 1, and debug the errors found by the test-case generator with the associated debugger by using it as a standard Maude file in the Eclipse environment. The source code of these tools, documentation, and examples are available at <http://maude.sip.ucm.es/debugging/>.

3 Testing the ADT implementation

When the user is convinced of the specification correctness, she selects an appropriate representation for the data type and implements it in C++. She may use the Eclipse environment and define the C++ files in the same project that the specification files. The generation of test cases requires a mapping between the sorts and function names of the Maude specification and the C++ implementation. This mapping is defined in a text file and it should contain all the operations that may be tested, including those with the same identifier in Maude and C++; see the testing tool manual for the concrete syntax of this file [10].

The user can now generate the test cases. First, he opens the specification file in the Eclipse editor window to obtain the buttons that manage the test-case generation in the menu bar (see Figure 2). These buttons are:

- (i)  (**init**). It is used only once to initialize the tool.
- (ii)  (**exec**). It generates a new test case.
- (iii)  (**stop**). It ends the session.

The user starts the test-case generator with the **init** button. Then she clicks the **exec** button and automatically the testing tool loads the specification file from the editor window. First, the tool will ask for the name of the mapping file. There can be several mapping files for one specification since there can be different implementations; for example, for the **STACK** specification there can be a static implementation with arrays and a dynamic implementation with linked lists. When the user selects the mapping file a new dialog box appears with the specified operations and asks for the one to be tested (see Figure 3). The operations that can be tested are obtained from the mapping file. The operations of the specification that do not appear in the mapping file, and therefore do not have an associated method implemented in C++, are considered private operations of the specification. Finally, the tool requires for the number of test cases to be generated.

Once all these steps are completed, the tool generates a C++ main program with the test cases. The user compiles and runs this program in the usual way.

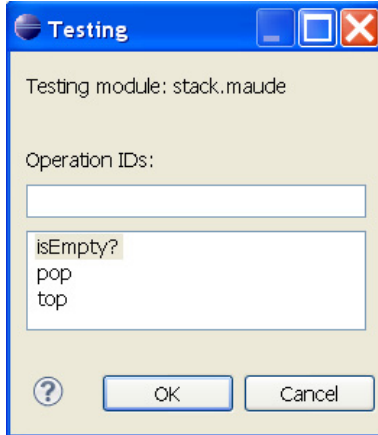


Fig. 3. Window for choosing the operation to be tested

```
.....false
case : search(plant(empty-tree,0,empty-tree),0)
specification result : true
implementation result : 0
```

Fig. 4. Failure of the search operation of BST

The tool shows a dot in the console for each test that passes. When a test fails, the tool finishes and writes in the console the specification term that fails, the result obtained from the execution of the specification, and the result obtained from the execution of the implementation (see Figure 4). No reduction is done on the failing case since the testing cases are obtained from the data type constructors in an incremental manner starting with the simplest ones.

The tool checks that the data computed by the implementation is *similar* to the one obtained from the specification. The notion of *similarity* is given for each ADT by the user by means of the comparison operator defined for the ADT, which shall be appropriately overloaded. The user may also overload the output operator that will show the implementation results in case of a failure. The more detailed the implementation of these operators will get the user more information about program bugs.

The number of test cases that can be generated in a reasonable time depends mainly on the number of constructor operations of the specification. For example, for the stack specification it takes about 7 seconds to generate 500 test cases and 25 seconds to generate 1000 cases on a PC. Concerning the execution of the test it takes few seconds to compile and execute the main program for 300 test cases. However, for more test cases the main program cannot be compiled due to lack of memory.

4 Tool design

The tool has a modular design to facilitate its evolution and the incorporation of new functionality (see Figure 5). It has four main modules:

- The *front-end* module is used to communicate with the user and enter the data. It has been implemented in Java under the Eclipse environment.

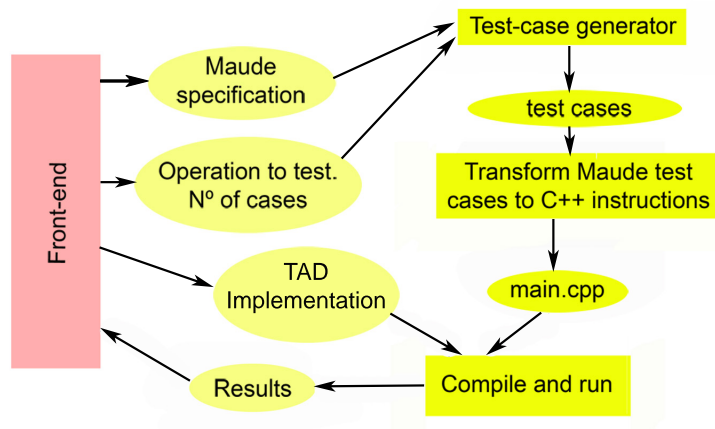


Fig. 5. Testing tool design

```

Maude> top (push (0, empty) )
top (push (0, push (0, empty) ) )
top (push (0, push (0, push (0, empty) ) ) )
top (push (0, push (1, empty) ) )
top (push (0, push (1, push (0, empty) ) ) )
top (push (-1, empty) )
top (push (-1, push (0, empty) ) )
top (push (-1, push (0, push (0, empty) ) ) )
top (push (-1, push (1, empty) ) )
top (push (-1, push (1, push (0, empty) ) ) )

```

Fig. 6. First generated test cases for the stack specification

- The *test-case generator* is implemented in Maude taking advantage of its reflective capabilities [4]. It uses Full Maude to facilitate the setting of: the module to be tested, the number of test cases, and the operation to be tested. The test-case generator looks in the module for the constructors and then uses them to generate terms for the specified function [12]. Some test cases generated for the stack specification are shown in Figure 6. These terms are later reduced in the metarepresented module to obtain the result.
- The module, implemented in Java, that transforms the Maude test cases into C++ instructions. Each test case is a shortlist of Maude terms, the first one represents the test case, the second one the result of reducing the test case using the equations of the specification, and the third one the sort of the result term. The module generates the sequence of C++ instructions that give rise to the objects that represents the test case and the result terms using the object constructors and the public methods. Then, uses the comparison operator, implemented by the user, to compare the two generated objects. This is done for all test cases.
- The Eclipse C++ compiler.

The integration with Eclipse is implemented as a plug-in. It uses the Maude APIs developed under the MOMENT project [11] that allow the exe-

cution of Maude as a batch process which is called by the test case generator. It is platform-independent and has been used in Mac and Windows systems.

5 Students experience

The tool has been used during this academic year, 2010-11, by the Computer Science students at the UCM in a data type structures course. They have tested typical data type implementations, like stacks, lists, and binary search trees, as well as other data types based on them. Since the implementation of these data types is well-known, the students are required to specify and implement new operations over them in order to practice implementations with linked lists or the usage of other data types. Over 70 students have completed a total of six programming assignments whose specification and implementation can be found at [10].

The experience has been very useful as it allows students to test their implementations and correct their errors. They have found not only implementation errors, but also specification ones, since the tool detects that the results of the specification and the implementation are different. More importantly, the tool has helped students to find the usefulness of a formal specification.

In general, the tool helps to find implementation errors and to obtain correct results, but not to perform an efficient design of the algorithm.

6 Conclusions and future work

There is little incentive for writing formal specifications unless they can be used to prove the correctness of the implementation in a simple and friendly way. Our tool can help Computer Science students to test their ADT implementations increasing their motivation to define formal specifications, resulting in improved software quality. The use of an integrated environment like Eclipse, familiar to the students, in which they can define specifications, write implementations, and prove them, shows students the usefulness of formal methods, not demotivating them with long formal proofs, but encouraging their use in future developments.

We plan to improve the Maude test-case generator incorporating new strategies to generate the test cases, such as narrowing, which would enhance the performance of the tool. Moreover, we are also interested in generating test cases that, in addition to constructors, are built by using some other defined functions; it will allow us to detect other errors, like dangling pointers. We need also to improve the algorithm that builds the C++ objects to consider some ADTs that cannot be currently treated, like those that do not have a relation between the specification constructors and the implementation constructors and introduce cppunits to cope with more test cases.

Acknowledgments

We are very grateful to Francisco Durán, Alberto Verdejo, and Narciso Martí for their ideas and comments about new ways to generate test cases and tool interface improvements.

References

- [1] A. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing, LNCS 3395*, pages 16–32. Springer, 2005.
- [2] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM Press, 2000.
- [3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework, LNCS 4350*. Springer, 2007.
- [4] M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2):70–91, 2007.
- [5] J. Gannon, P. McMullin, and R. Hamlet. Data abstraction, implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3:211–223, July 1981.
- [6] M.-C. Gaudel and P. Le Gall. Testing data types implementations from algebraic specifications. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal methods and testing*, pages 209–239. Springer, Berlin, Heidelberg, 2008.
- [7] P. D. L. Machado. Testing from structured algebraic specifications. In T. Rus, editor, *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology, AMAST 2000, LNCS 1816*, pages 529–544. Springer, 2000.
- [8] N. Martí-Oliet, M. Palomino, and A. Verdejo. A tutorial on specifying data structures in Maude. In S. Lucas, editor, *Proceedings of the Fourth Spanish Conference on Programming and Computer Languages, PROLE 2004, ENCS 137(1)*, pages 105–132. Elsevier, 2005.
- [9] I. Pita. Guía rápida sobre ejecución de especificaciones algebraicas en Maude bajo el entorno Eclipse para estudiantes de estructuras de datos. Technical Report 5/11, Departamento de Sistemas Informáticos y Computación, 2011. <http://maude.sip.ucm.es/maudeADTesting/>.
- [10] I. Pita. Manual para realizar testing de TADs especificados en Maude e implementados en C++. Technical Report 6/11, Departamento de Sistemas Informáticos y Computación, 2011. <http://maude.sip.ucm.es/maudeADTesting/>.
- [11] I. Ramos, J. A. C. Cubel, A. Boronat, and A. Gómez. MOMENT: A framework for MOdel manageMENT. <http://moment.dsic.upv.es/>.
- [12] A. Riesco. Test-case generation for Maude functional modules. In *Proceedings of the 20th International Workshop on Algebraic Development Techniques, WADT 2010*, Lecture Notes in Computer Science. Springer, 2011. To appear.
- [13] A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. Declarative debugging of rewriting logic specifications. *Journal of Logic and Algebraic Programming*, 2011. To appear.