Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 1743 – 1752

Procedia
Computer Science

International Conference on Computational Science, ICCS 2012

An Innovative Teaching Tool for the Verification of Abstract Data Type Implementations from Formal Algebraic Specifications

Rafael del Vado Vírseda, Fernando Pérez Morente [*]

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain, Madrid, C.P. 28040

Abstract

This paper presents an educational tool for testing abstract data types implemented in *C++* against formal algebraic specifications written in *Maude*, a formal specification language based on rewriting logic that allows the specification of abstract data types in a clear and concise manner. *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 generated from the *Maude* specification based on the data type constructors and the corresponding membership axioms. The observation of the implementation under test is done for each abstract data type through explicit methods defined by the user. The teaching 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 translates them into a sequence of *C++* instructions. The *C++* instructions are compiled and executed, and the results are compared with the results obtained from the formal algebraic specification. This educational tool is being used during this academic year by the Computer Science students in a data types course. They have tested typical abstract data type implementations, like complex numbers, stacks, lists, and binary search trees, as well as other data types based on them.

Keywords: Tools to aid in teaching, Abstract data types, Formal algebraic specifications, Software testing tools

1. Introduction

The study of *abstract data types* and their formal *algebraic specifications* [5] constitutes one of the essential aspects of the academic formation of every student in Computational Science. Nevertheless, the high level of abstraction necessary to teach these topics occasionally difficults its understanding to students. There is little incentive and motivation for writing formal algebraic specifications unless they can be used to prove the correctness of their data type implementations in a simple and friendly way. For this reason, the development of an educational tool in a integrated programming environment, familiar to the students, in which they can define formal algebraic specifications, write data types implementations in *C++* or *Java* and prove them to correct their errors, can be useful in Computer Science Education to show students the usefulness of formal methods and encouraging their use in future software developments.

[*]Corresponding authors
Email address: `rdelvado@sip.ucm.es,fperezmo@fdi.ucm.es` (Fernando Pérez Morente)

doi:10.1016/j.procs.2012.04.192

This paper presents an educational tool for testing abstract data type implementations against formal algebraic specifications [6, 7]. Formal 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 helps in the analysis of the correctness of specifications, like the *Maude termination tool*, the *Church-Rosser checker*, and the *Maude debugger and testing tool* [12, 13].

Our testing tool is designed for helping Computer Science students with the implementation of data structures. In fact, it has been used during the academic year 2010-11 in a data type structures course, which motivated the use of the *C++* language as implementation language. The teaching tool is fully integrated in the *Eclipse* environment [1], which is platform-independent and provides environments, defined by plugins, for *Maude* specifications and *C++* implementations. The students can write, compile, and execute their formal algebraic specifications in the same environment in which they implement the abstract data types, generate the test cases, and prove them. Algebraic specifications define the abstract data type behavior using constructor functions, that create or initialize the data elements, and other functions, that operate on the data types. Currently our 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://gpd.sip.ucm.es/Testing/ICCS2012/`.

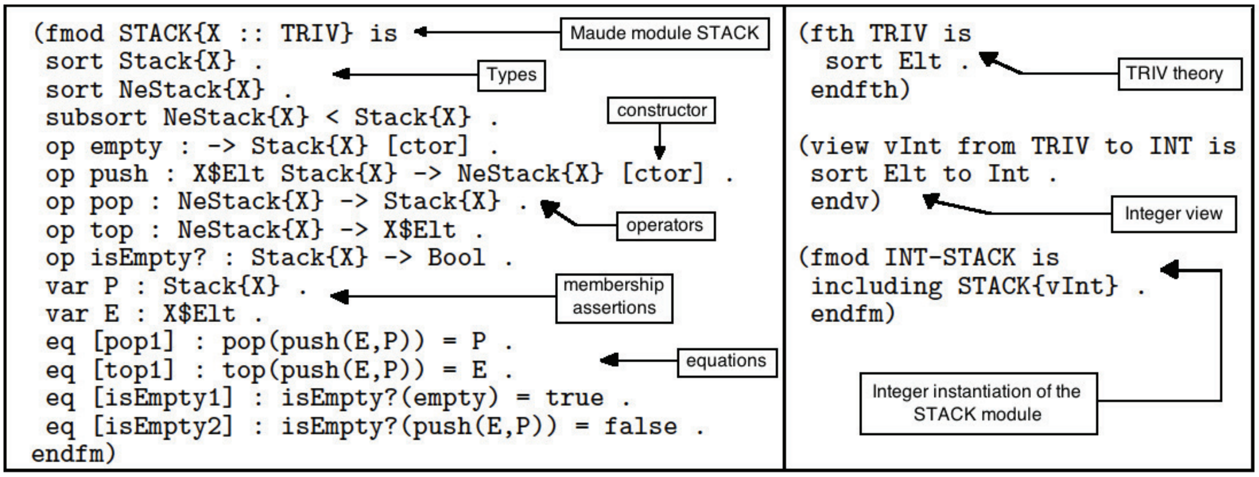
There has been much related work in the area of software testing tools based on algebraic specifications from the 80s and 90s. These approaches mainly use algebraic specifications to help on the generation of the test sets. They focus on finding the conditions for constructing an ideal exhaustive test 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-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* [2]. *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 abstract data type 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 defined an algebraic 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 abstract data type implementations in *C++*. **Section 4** explains the design of the educational testing 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. Abstract Data Type Specification in *Maude*

Abstract data types are formally specified in *Maude* as *functional modules*, which correspond to equational theories in membership equational logic [3]. Specifically we use a *Core Maude* extension called *Full Maude*, since its syntax is almost equal to that used in *Core Maude* and we found very convenient to keep the abstract data type modules in the *Full Maude* database for the test cases generation. **Figure 1** presents a specification of the abstract data type *stack* in the module `STACK`. The module starts with the keywork `fmod`, followed by the module name, an optional parameter

[1]`http://www.eclipse.org/`
[2]`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)
```

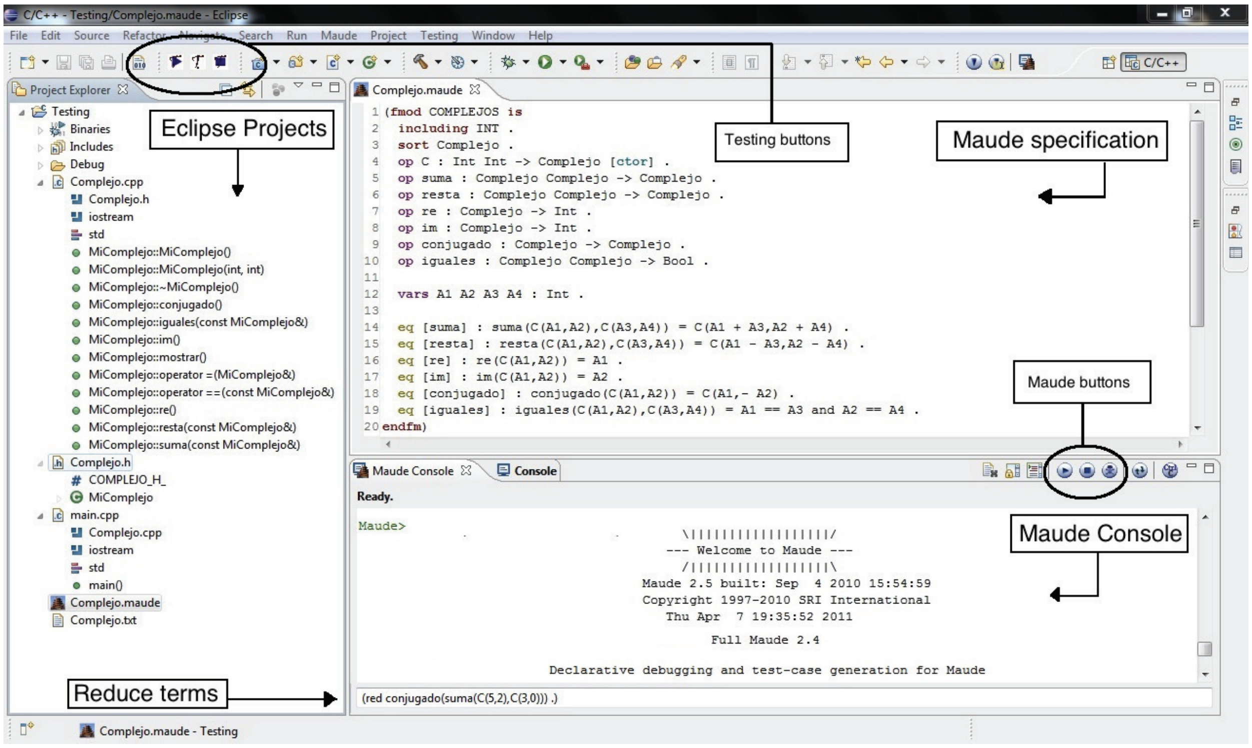
```
(fth TRIV is
  sort Elt .
 endfth)

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

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

Figure 1: Abstract data type specification of a generic stack in *Maude*.

declaration (in our case by the theory `TRIV`, that we will explain below, with the parameter `X`), and the keywork `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 keywork `op`, is declared together with the sorts of its arguments and the sort of its results; 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 abstract data type; it is used to generate the test cases.

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* $t : s$, stating that the term t has sort s, and *equations* $t = t'$ (introduced with keyword `eq`), stating that t and t' are equal. Parameterized abstract data types use theories to specify the requirements that the parameter must satisfy. The ones are defined by the user. *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, 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 also 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 an *Eclipse* window for a simple *Maude* specification of *complex numbers*: 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. 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 on the console.



```
(fmod COMPLEJOS is
  including INT .
  sort Complejo .
  op C : Int Int -> Complejo [ctor] .
  op suma : Complejo Complejo -> Complejo .
  op resta : Complejo Complejo -> Complejo .
  op re : Complejo -> Int .
  op im : Complejo -> Int .
  op conjugado : Complejo -> Complejo .
  op iguales : Complejo Complejo -> Bool .

  vars A1 A2 A3 A4 : Int .

  eq [suma] : suma(C(A1,A2),C(A3,A4)) = C(A1 + A3,A2 + A4) .
  eq [resta] : resta(C(A1,A2),C(A3,A4)) = C(A1 - A3,A2 - A4) .
  eq [re] : re(C(A1,A2)) = A1 .
  eq [im] : im(C(A1,A2)) = A2 .
  eq [conjugado] : conjugado(C(A1,A2)) = C(A1,- A2) .
  eq [iguales] : iguales(C(A1,A2),C(A3,A4)) = A1 == A3 and A2 == A4 .
endfm)
```

```
Maude>
                        \||||||||||||||||||/
                      --- Welcome to Maude ---
                        /||||||||||||||||||\
                 Maude 2.5 built: Sep  4 2010 15:54:59
                 Copyright 1997-2010 SRI International
                    Thu Apr  7 19:35:52 2011

                          Full Maude 2.4

           Declarative debugging and test-case generation for Maude
```

```
(red conjugado(suma(C(5,2),C(3,0))) .)
```

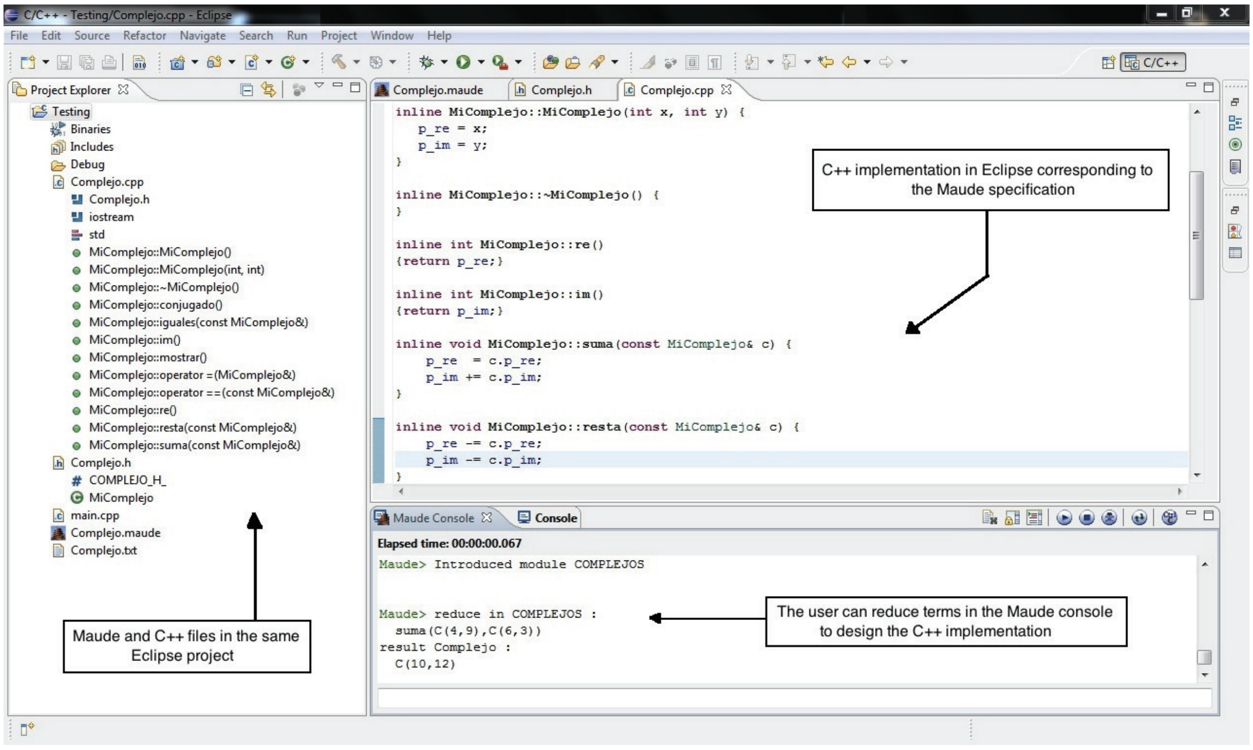
Figure 2: *Maude* specifications can be executed under the *Eclipse* environment.

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 *Maude* 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 *Maude* system. Only instantiated terms can be reduced. The results 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 *complex numbers* specification in **Figure 2**, 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 Abstract Data Type Implementation

When the student is convinced of the algebraic specification correctness, she selects an appropriate representation for the abstract data type and implements it in *C++*. He may use the *Eclipse* environment and defines the *C++* files in the same project that the *Maude* specification files (see **Figure 3**). 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].


C/C++ - Testing/Complejo.cpp - Eclipse
File Edit Source Refactor Navigate Search Run Project Window Help
Project Explorer
Testing
Binaries
Includes
Debug
Complejo.cpp
Complejo.h
iostream
std
MiComplejo::MiComplejo()
MiComplejo::MiComplejo(int, int)
MiComplejo::~MiComplejo()
MiComplejo::conjugado()
MiComplejo::iguales(const MiComplejo&)
MiComplejo::im()
MiComplejo::mostrar()
MiComplejo::operator =(MiComplejo&)
MiComplejo::operator ==(const MiComplejo&)
MiComplejo::re()
MiComplejo::resta(const MiComplejo&)
MiComplejo::suma(const MiComplejo&)
Complejo.h
COMPLEJO_H_
MiComplejo
main.cpp
Complejo.maude
Complejo.txt
Complejo.maude
Complejo.h
Complejo.cpp
C++ implementation in Eclipse corresponding to the Maude specification
Maude and C++ files in the same Eclipse project
Maude Console
Console
The user can reduce terms in the Maude console to design the C++ implementation

```
inline MiComplejo::MiComplejo(int x, int y) {
   p_re = x;
   p_im = y;
}

inline MiComplejo::~MiComplejo() {
}

inline int MiComplejo::re()
{return p_re;}

inline int MiComplejo::im()
{return p_im;}

inline void MiComplejo::suma(const MiComplejo& c) {
    p_re  = c.p_re;
    p_im += c.p_im;
}

inline void MiComplejo::resta(const MiComplejo& c) {
    p_re -= c.p_re;
    p_im -= c.p_im;
}
```

```
Elapsed time: 00:00:00.067
Maude> Introduced module COMPLEJOS


Maude> reduce in COMPLEJOS :
  suma(C(4,9),C(6,3))
result Complejo :
  C(10,12)
```

Figure 3: Students can use the integrated environment of *Eclipse* to write implementations in *C++* and prove them in the *Maude* console.

The student 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:

(1) (`init`): It is used only once to initialize the testing tool.

(2) (`exec`): It generates a new test case.

(3) (`stop`): It ends the session.

The student 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 4**). 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. Then, the testing 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 student can now compile and run this program in the usual way.

The testing tool shows a dot in the console for each test that passes (see **Figure 6**). 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 5**). No reduction is done on

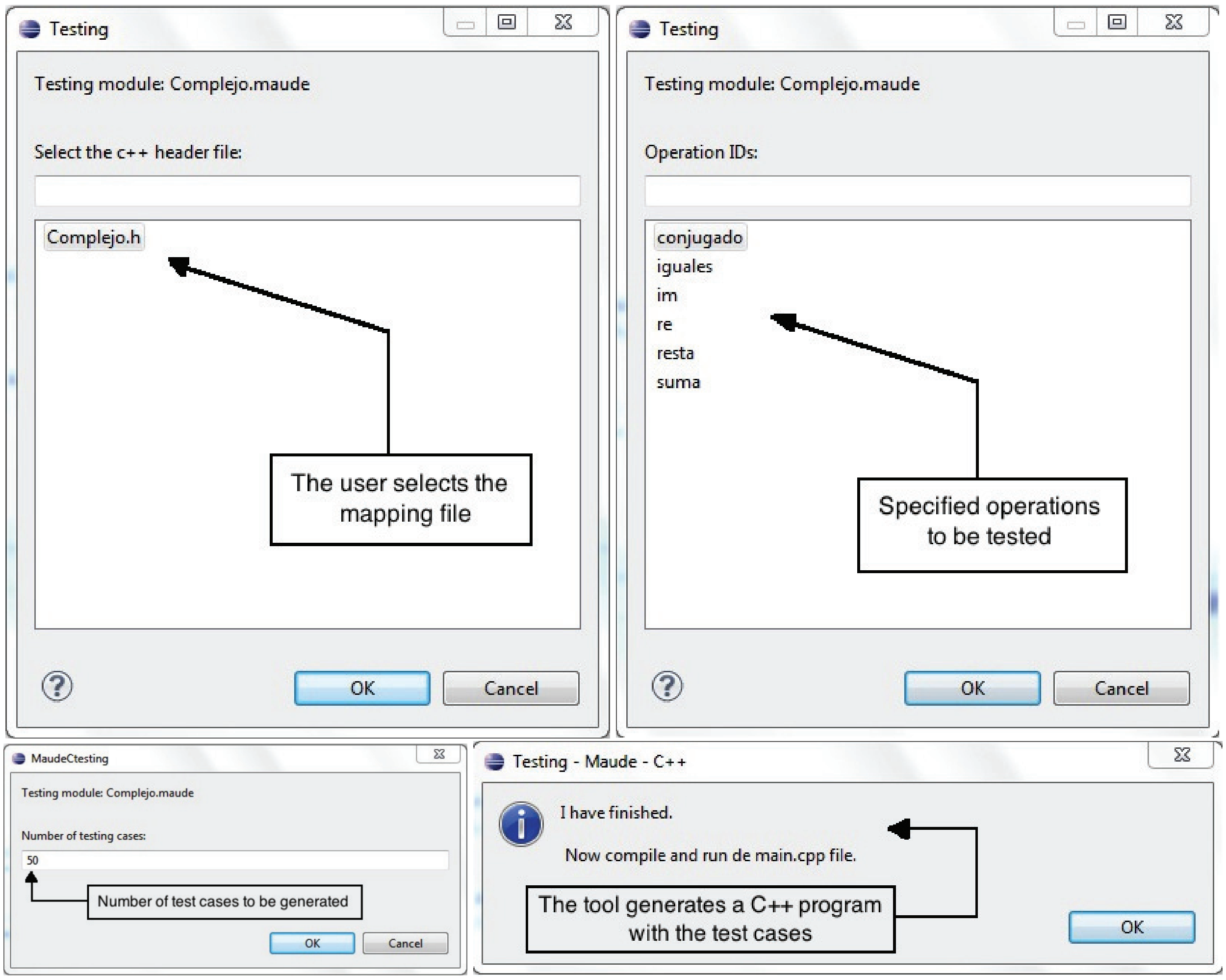
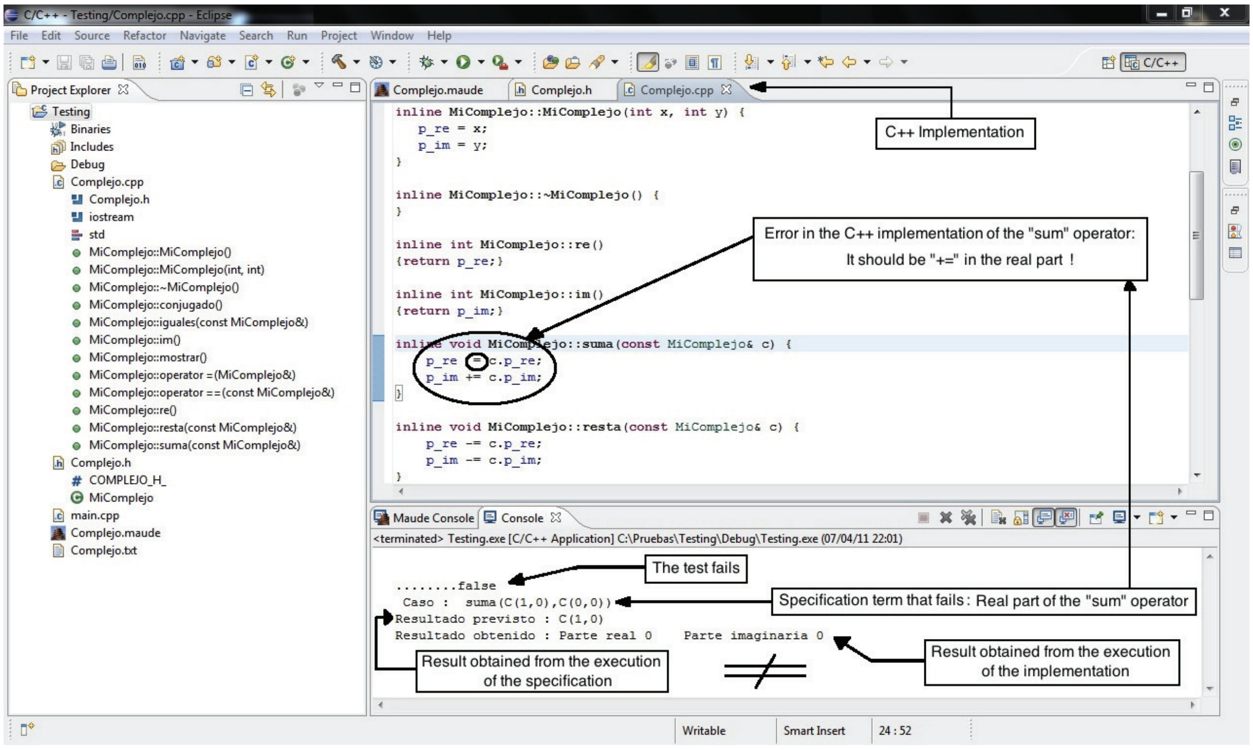

Testing
Testing module: Complejo.maude
Select the c++ header file:
Complejo.h
The user selects the mapping file
OK
Cancel
Testing
Testing module: Complejo.maude
Operation IDs:
conjugado
iguales
im
re
resta
suma
Specified operations to be tested
OK
Cancel
MaudeCtesting
Testing module: Complejo.maude
Number of testing cases:
50
Number of test cases to be generated
OK
Cancel
Testing - Maude - C++
I have finished.
Now compile and run de main.cpp file.
The tool generates a C++ program with the test cases
OK

Figure 4: A dialog box appears with the specified operations and asks for the one to be tested and the number of test cases to be generated.

the failing case since the testing cases are obtained from the data type constructors in an incremental manner starting with the most simple ones.

The testing tool checks that the data computed by the *C++* implementation is *similar* to the one obtained from the *Maude* specification. The notion of *similarity* is given for each abstract data type by the student by means of the comparison operator defined for the abstract data type, 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 constructors 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*. Concercing 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.


C/C++ - Testing/Complejo.cpp - Eclipse
File Edit Source Refactor Navigate Search Run Project Window Help
Project Explorer
Testing
Binaries
Includes
Debug
Complejo.cpp
Complejo.h
iostream
std
MiComplejo::MiComplejo()
MiComplejo::MiComplejo(int, int)
MiComplejo::~MiComplejo()
MiComplejo::conjugado()
MiComplejo::iguales(const MiComplejo&)
MiComplejo::im()
MiComplejo::mostrar()
MiComplejo::operator =(MiComplejo&)
MiComplejo::operator ==(const MiComplejo&)
MiComplejo::re()
MiComplejo::resta(const MiComplejo&)
MiComplejo::suma(const MiComplejo&)
Complejo.h
COMPLEJO_H_
MiComplejo
main.cpp
Complejo.maude
Complejo.txt
Complejo.maude
Complejo.h
Complejo.cpp
C++ Implementation
Error in the C++ implementation of the "sum" operator:
It should be "+=" in the real part !
Maude Console
Console
The test fails
Specification term that fails : Real part of the "sum" operator
Result obtained from the execution of the specification
Result obtained from the execution of the implementation
Writable
Smart Insert
24 : 52

```
inline MiComplejo::MiComplejo(int x, int y) {
   p_re = x;
   p_im = y;
}

inline MiComplejo::~MiComplejo() {
}

inline int MiComplejo::re()
{return p_re;}

inline int MiComplejo::im()
{return p_im;}

inline void MiComplejo::suma(const MiComplejo& c) {
    p_re = c.p_re;
    p_im += c.p_im;
}

inline void MiComplejo::resta(const MiComplejo& c) {
    p_re -= c.p_re;
    p_im -= c.p_im;
}
```

```
<terminated> Testing.exe [C/C++ Application] C:\Pruebas\Testing\Debug\Testing.exe (07/04/11 22:01)

........false
 Caso :  suma(C(1,0),C(0,0))
Resultado previsto : C(1,0)
Resultado obtenido : Parte real 0    Parte imaginaria 0
```

Figure 5: When a test fails, the tool finishes and writes in the *Maude console* the specification term that fails.

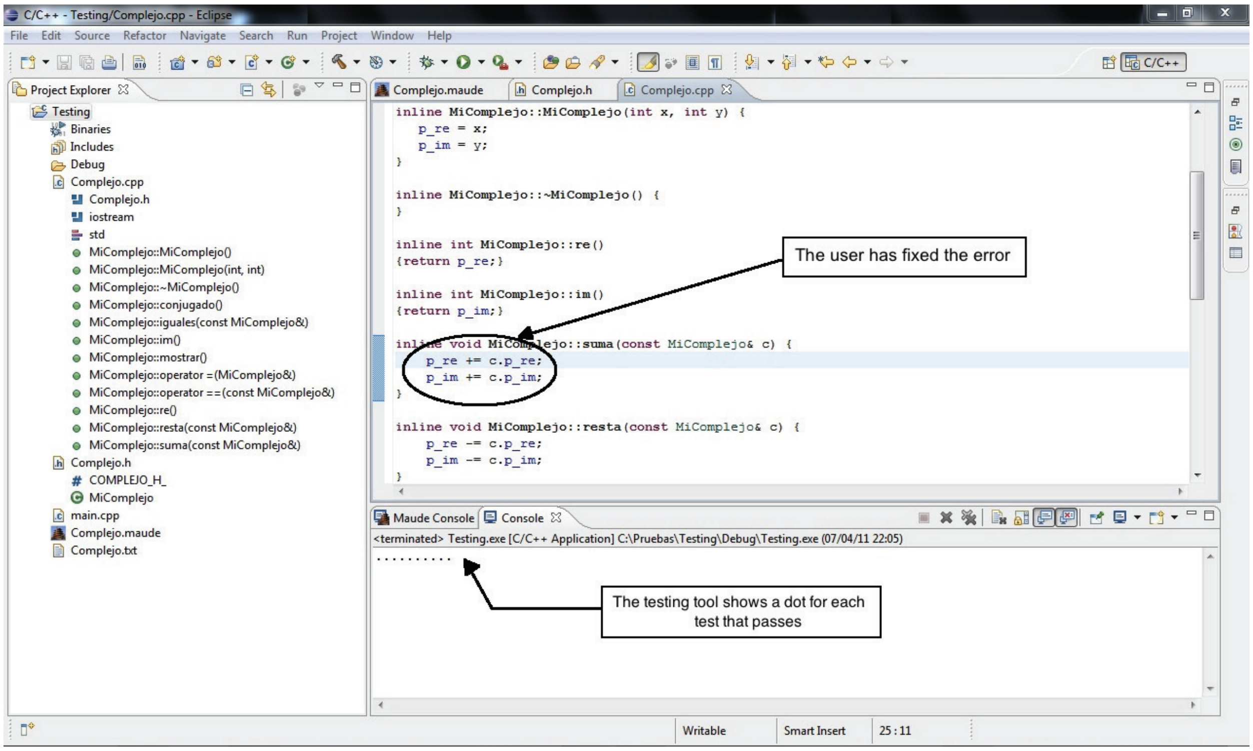
4. Testing Tool Design

The testing tool has a modular design to facilitate its evolution and the incorporation of new functionality (see **Figure 7**). It has four main modules:

- The *front-end* module is used to communicate with the user and enter the data. It obtains the specification code, the mapping file, the operation to be tested, and the number of testing cases. It has been implemented in *Java* under the *Eclipse* environment.
- 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 below. These terms are later reduced in the metarepresented module to obtain the result.

  ```
  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))))
  ```

- The module, implemented in *Java*, that transforms the *Maude* test cases into *C++* instructions. Each test is a shortlist of *Maude* terms, the first one represents the test case, the second one the result of reducing the test


C/C++ - Testing/Complejo.cpp - Eclipse
File Edit Source Refactor Navigate Search Run Project Window Help
Project Explorer
Testing
Binaries
Includes
Debug
Complejo.cpp
Complejo.h
iostream
std
MiComplejo::MiComplejo()
MiComplejo::MiComplejo(int, int)
MiComplejo::~MiComplejo()
MiComplejo::conjugado()
MiComplejo::iguales(const MiComplejo&)
MiComplejo::im()
MiComplejo::mostrar()
MiComplejo::operator =(MiComplejo&)
MiComplejo::operator ==(const MiComplejo&)
MiComplejo::re()
MiComplejo::resta(const MiComplejo&)
MiComplejo::suma(const MiComplejo&)
Complejo.h
COMPLEJO_H_
MiComplejo
main.cpp
Complejo.maude
Complejo.txt
Complejo.maude
Complejo.h
Complejo.cpp
The user has fixed the error
Maude Console
Console
The testing tool shows a dot for each test that passes
Writable
Smart Insert
25 : 11

```
inline MiComplejo::MiComplejo(int x, int y) {
   p_re = x;
   p_im = y;
}

inline MiComplejo::~MiComplejo() {
}

inline int MiComplejo::re()
{return p_re;}

inline int MiComplejo::im()
{return p_im;}

inline void MiComplejo::suma(const MiComplejo& c) {
    p_re += c.p_re;
    p_im += c.p_im;
}

inline void MiComplejo::resta(const MiComplejo& c) {
    p_re -= c.p_re;
    p_im -= c.p_im;
}
```

```
<terminated> Testing.exe [C/C++ Application] C:\Pruebas\Testing\Debug\Testing.exe (07/04/11 22:05)
..........
```

Figure 6: The testing tool shows a dot in the *Maude console* for each test that passes.

case using the equations of the specification, and the third one the sort of the result term. The module generates the *C++* instructions that give rise to an object that represents the *Maude* term. It adds the execution of the method on the created object, generates the sequence of instructions for the result term, and uses the comparison operator implemented by the student on the abstract data type to compare the two generated objects.

- 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 execution of *Maude* as a batch process to call to the test case generator. It is platform-independent and has been used in *Mac* and *Windows* systems.

5. Student Experience

The software testing tool has been used during the academic year 2010-11, by the Computer Science students at the Complutense University of Madrid in the "Data Structures" academic subject at the second year. They have tested typical abstract data type implementations, like *complex numbers*, *stacks*, *lists*, *binary search trees*, and *tables* as well as other data types based on them. Since the implementation of these abstract 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 abstract data types. Over 75 of 89 students have completed a total of five programming assignments whose specification and implementation can be found at `http://gpd.sip.ucm.es/Testing/ICCS2012/`.

The experience has proved very useful as it allows students to test their implementations and correct their errors. They have found not only implementation errors, but also specifications ones, since the testing tool detects that the results of the specification and the implementation are different. More importantly, the testing tool has helped students to find the usefulness of a formal algebraic specification. The following table shows the high number of students that have passed each of the five programming assignments: *complex numbers*, *stacks*, *lists*, *binary search trees*, *tables*.

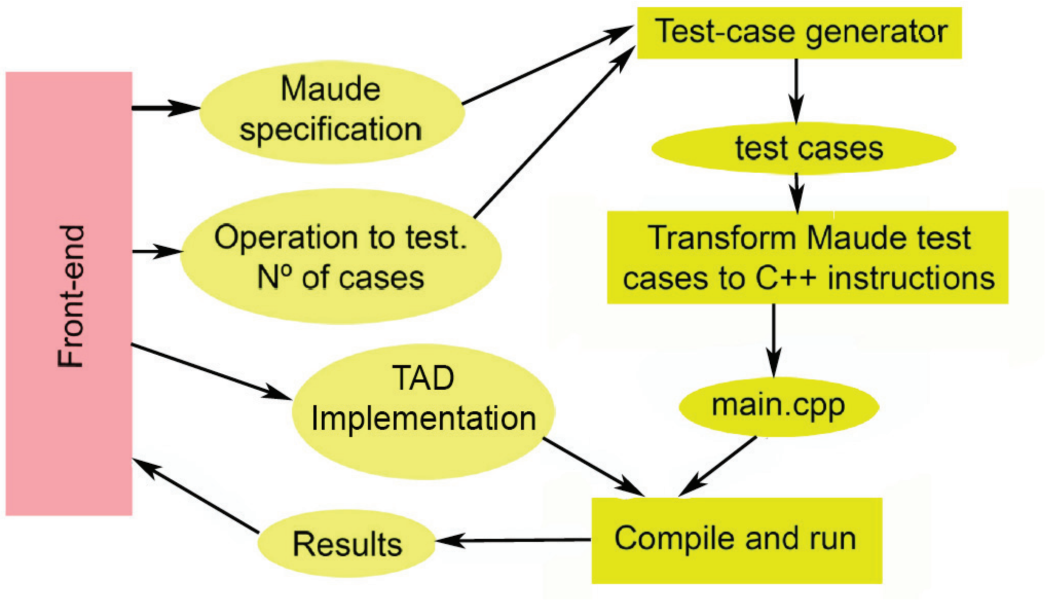


Figure 7: Design of the software testing tool: From the *Maude* specification to the *C++* implementation.

Pract. 1	Pract. 2	Pract. 3	Pract. 4	Pract. 5
83	82	74	72	75

In general, in this educational experience we have noted that the testing tool helps to find implementation errors and to obtain correct results, but not to perform an efficient design of the algoritm.

6. Conclusions and Future Work

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

As future work, we plan to improve the *Maude* test-case generator incorporating new strategies to generate the test cases, such as *narrowing* [3], which would enhance the performance of the testing tool. Moreover, we are also interested in generating test cases that, in addition to constructors, are built by using some other defined functions; helping in finding other errors, like dangling pointers. We need also to improve the algorithm that builds the *C++* objects from the *Maude* terms in order to consider some abstract data types 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

The work of this author has been partially supported by the Spanish projects FAST-STAMP (TIN2008-06622-C03-01), PROMETIDOS-CM (S2009TIC-1465), GPD (UCM-BSCH-GR58/08-910502), and PIMCD n^{o}150 2011/2012 (Project for the Innovation and Improvement of the Educational Quality).

I am grateful to Isabel Pita, Adrián Riesco, 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. Palominio, 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, ENTCS 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, 2001. `http://gpd.sip.ucm.es/Testing/ICCS2012/`.

[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://gpd.sip.ucm.es/Testing/ICCS2012/`.

[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*.