MUnit: A Unit Framework for Maude

Adrián Riesco(✉)

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

Abstract. Unit testing is a widely-used methodology for checking whether the units of a given program work as expected. Maude is a high performance rewriting engine based on rewriting logic. Although Maude has been used to implement complex specifications and tools, it lacks the testing tools usually supported by other languages. In this work we present MUnit, a unit testing framework for Maude that takes into account its main features to define meaningful unit tests. MUnit extends Full Maude and supports functional and system modules, as well as specifications using the Loop Mode and, in particular, Full Maude.

Keywords: Unit testing · Maude · Loop Mode · Full Maude

1 Introduction

Debugging and testing conform one of the most important stages of the software developing cycle, requiring up to the 50% of the time [5]. Among the best known testing methodologies we find property-based testing [8], conformance testing [23], and unit testing [22]. In particular, unit testing is a well-known testing method for checking whether a *unit* behaves as expected. In general, unit tests consist of the unit applied to ground arguments and an expected value; the test passes if both values are equal, although equality can be substituted by a more general notion in particular cases. In imperative languages units usually refer to methods/calls, although other units can be considered depending of the target language. Finally, note also that in unit testing fulfilling the target coverage [3] is in charge of the user.

Maude [10] is a logical framework and high-performance rewriting engine. Maude modules correspond to specifications in *rewriting logic* [15], a logic that allows specifiers to represent many models of concurrent and distributed systems. This logic is an extension of *membership equational logic* [4], an equational logic that, in addition to equations, allows the statement of *membership axioms* characterizing the elements of a sort.

Research partially supported by MINECO Spanish project *TRACES* (TIN2015-67522-C3-3-R) and Comunidad de Madrid project N-Greens Software-CM (S2013/ICE-2731).

Maude modules are executable rewriting logic specifications. Maude functional modules [10, Chap. 4] are executable membership equational specifications that allow the definition of sorts, subsort relations between sorts, operators for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative or commutative, for example; memberships asserting that a term has a sort; and equations asserting that terms are equal. Both memberships and equations can be conditional. Maude system modules [10, Chap. 6] are executable rewrite theories. A system module can contain all the declarations of a functional module and, in addition, declarations for rules and conditional rules. Finally, Full Maude [10, Part II] is an extension of Maude written in Maude itself. Full Maude is built on top of the `LOOP-MODE` module [10, Chapter 17]. This module allows input/output interaction by means of the `[_,_,_]` operator, which builds terms of sort `System` and where the first argument corresponds to the input introduced by the user, which must be enclosed in parentheses to be recognized; the second one is a term of sort `State` that can be defined by the user for each application; and the third one the output shown to the user.

Maude has been used to implement tools such as termination and confluence checkers, theorem provers, real-time extensions, etc. and to specify a wide range of systems, including bio informatics, network protocols, and mobile languages, among many others.[1] However, Maude only provides a limited property-based testing tool [19], which supports functional and system modules. This tool is implemented using narrowing [9], which does not support some theories, in particular those using conditional equations/rules. The transformation to overcome this problem makes the process slower, so the tool lacks the efficiency required to work with large specifications. Since the implementation of MUnit directly uses Maude (meta)commands, the time required to test any Maude specification will be similar to the time required to execute it. More generally, a unit framework is also useful (i) to test functions that do not have associated properties, (ii) to test particular inputs that the user knows might lead to errors, and (iii) to quickly check whether changes in the implementation are correct with respect to a test suite.

In this paper we present MUnit, a unit test framework for Maude that supports functional and system modules, as well as applications on top of the `LOOP-MODE` module (using both input/output facilities and in particular Full Maude features, such as object-oriented modules). Supporting Full Maude applications is particularly interesting for a number of reasons: (i) they are difficult to test, especially those commands producing "side effects" in the internal state of the loop; (ii) in contrast to other Maude applications that are used to analyze particular systems and produce a results, Full Maude applications are designed for being used by other users so, in addition to be thoroughly tested, it is interesting to add the errors reported by users in an easy way; (iii) most of them are large, complex software systems, like Full Maude itself, Real-Time Maude [17], the Maude Formal Environment [11], and the CafeOBJ environment [20], so

[1] See http://maude.cs.illinois.edu/ for a comprehensive list of Maude projects.

a well established testing methodology would help to maintain them; and (iv) there are many applications of this kind; actually, any application requiring I/O interaction or manipulating the database usually extends Full Maude.

The rest of the paper is organized as follows: Sect. 2 presents the main features of MUnit, while Sect. 3 illustrates these features by means of an example. Section 4 outlines the implementation of the tool. Section 5 discusses the related work. Finally, Sect. 6 concludes and proposes some lines of future work to improve the tool. The source code of MUnit, examples, and more information is available at https://github.com/ariesco/MUnit.

2 MUnit

In this section we present the tests available in MUnit. We distinguish the different tests depending on the module under test in order to describe their particularities.

Given that functional modules are confluent and terminating, we can consider that function calls are units that must be reduced to a particular value, and hence equality is enough for checking correctness. Moreover, functional modules support the definition of membership axioms stating the elements of a sort, so we need units to test whether a term has a given sort:

– The test `assertEqual(f(t1, ..., tn), t)` passes if the function `f`, when applied to the ground terms `t1, ...,tn`, is reduced to the same normal form as `t` (modulo axioms). Similarly, the test `assertDifferent(f(t1, ..., tn), t)` passes if the normal form of `f(t1, ..., tn)` is different from the normal form of `t` (modulo axioms). Note that `t` might not be a normal form; for example, it might be a constant defined to ease the testing process.[2] Note also that these tests are commutative, so users can define them according to their preferences.
– MUnit provides shortcuts for Boolean tests. The test `assertTrue(f(t1, ..., tn))` (respectively, `assertFalse(f(t1, ..., tn))`) passes if the term is reduced to `true` (respectively, `false`).
– We can also test whether membership axioms are properly defined by checking the sort of a given term by using `assertSort(t,s)`, which passes if the normal form of the term `t` has exactly sort `s`. MUnit also provides a test `assertLeqSort(t, s)`, which passes if the sort of the normal form of `t` is less or equal to `s`.

When dealing with system modules we face specifications potentially non-terminating and non-confluent. For this reason the units for these modules assert *reachability* rather than equality. Note that object-oriented modules are considered as standard system modules by MUnit.

[2] Likewise, if `f` is not a function but a constructor and we are interested in testing how the terms `t1, ..., tn` behave the test would compute the normal form and compare it with the normal form of `t`. However, this test would not follow completely the philosophy of Maude unit tests as we have defined it.

- The test `assertReachable(t, t')` passes if the term `t'` is reachable from `t` within an unbounded number of steps. Similarly, `assertReachableBnd(t, t', bnd)` adds information about the bound (`bnd`) in the number of steps.
- Usually we are not interested on reaching a specific term but on a solution matching a pattern or just in the lack of solutions. Taking this consideration into account, the test `hasSolution(t, pat, mode, bound, cond)` passes when there exists at least one reachable term that, starting from `t`, matches the pattern `pat` and fulfills the condition `cond` in at most `bound` (either `unbounded` or a natural number) steps. The `mode` can be either `*`, for 0 or more steps; `+`, for 1 or more steps; and `!`, for final states. Similarly, the test `noSolution(t, pat, mode, bound, cond)` passes when no solution was expected.

Finally, we present how to test applications extending the Loop Mode. The basic idea for testing this kind of applications is essentially the same that we have discussed above: testing those units that will be used during the input/output process. However, the internal state of the loop makes some tests difficult to define. For this reason, MUnit provides instructions to start an inner loop and execute commands on it; the user can then perform tests on the intermediate states with the tests above. The instructions for manipulating the inner loop are:

- `loop(initial-state)`. This instruction starts the MUnit inner loop by rewriting `initial-state`, which must have sort `System`. For example, we would use `loop(init)` to start a Full Maude session.
- `command(comm)`. This instruction introduces the command `comm` into the first element of the loop and rewrites the thus obtained term to evolve the system. For example, once started Full Maude with the command above we would use `command(select NAT .)` to change the default module to `NAT`.

Although it is possible to define tests while executing commands, right now we have no means to access the attributes defined in the state (the second argument of the loop). MUnit makes attributes available by using `@` before the attribute name. For example, given that the Full Maude explicit database is identified by `db` we would use `@db` to access it. We can use it to ensure that the database is well formed after introducing new modules, to analyze the modules it contains, and to test functions that require it. In particular, we would check that it is well formed by using `assertSort(@db, Database)`, that indicates that the database has sort `Database` (and hence it is not defined at the kind level, which indicates an error occurred). We show a detailed example in the following section.

3 Running Example

In this section we present how to use the tool by using a simple inventory specification. Note that the tests shown here are used for illustrating the tool; discussions about coverages are beyond the scope of this section. The complete source code of the example is available at https://github.com/ariesco/MUnit.

We first define binary search trees in module BSTREE to store the information about the stock. We will use the product name (a String) as key and a Pair of the form < QTY, PR > as value, where QTY stands for the quantity of the product and PR for its price. Then, we define the sort BSTree for binary search trees and BST? for trees that do not fulfill the appropriate property. Hence, the empty tree (mt) has sort BSTree, while the constructor for bigger trees builds terms of sort BSTree?:

```
fmod BSTREE is
 pr STRING .

 sort BSTree BSTree? Pair .
 subsort BSTree < BSTree? .

 op <_,_> : Nat Nat -> Pair [ctor] .

 op mt : -> BSTree [ctor] .
 op _[_,_]_ : BSTree? String Pair BSTree? -> BSTree? [ctor] .
```

We define a membership axiom to assign the sort BSTree. Given that the left and the right trees have the appropriate sort (i.e., the variables L and R have sort BSTree), we check with the auxiliary function correctOrder that the key in the root is appropriately sorted with respect to the children:

```
 cmb L [S, P] R : BSTree
  if correctOrder(L, S) /\ correctOrder(S, R) .
```

We also specify functions to check whether an item is in the tree (inStock); to insert a new item (insert); to delete an item from the tree (delete); and to update the tree by subtracting one unit of the given item (oneSold):

```
 op inStock : BSTree String -> Bool .
 op insert  : BSTree String Pair -> BSTree .
 op delete  : BSTree String -> BSTree .
 op oneSold : BSTree String -> BSTree .
```

However, we introduced an error in the definition of insert: we did not define the case when the element being introduced is already in the tree:

```
 eq insert(mt, S, P) = mt [S, P] mt .
 ceq insert(L [S, P] R, S', P') = insert(L, S', P') [S, P] R
  if S' < S .
 ceq insert(L [S, P] R, S', P') = L [S, P] insert(R, S', P')
  if S < S' .
 *** eq insert(L [S, P] R, S, P') = L [S, combine(P, P')] R .
 ...
endfm
```

Finally, we define a module FTEST with constants for testing. Tree treeOK1 is a binary search tree with three elements, a, c, and e; treeOK2 consists of treeOK2 after introducing the item b; treeOK3 is treeOK1 after removing the item in the root, c; treeOK4 has one unit of c less than treeOK1; and treeError is a tree with unordered keys:

```
fmod FTEST is
 pr BSTREE .

 ops treeOK1 treeOK2 treeOK3 treeOK4 treeError : ~> BSTree .
 eq treeOK1 = (mt [ "a", < 1, 3 > ] mt) ["c", < 2, 7 > ]
              (mt [ "e", < 3, 5 > ] mt) .
 eq treeOK2 = (mt [ "a", < 1, 3 > ] (mt ["b", < 1, 1 >] mt))
              ["c", < 2, 7 > ] (mt [ "e", < 3, 5 > ] mt) .
 eq treeOK3 = (mt ["a", < 1, 3 >] mt)["e", < 3, 5 >] mt .
 eq treeOK4 = (mt [ "a", < 1, 3 > ] mt) ["c", < 1, 7 > ]
              (mt [ "e", < 3, 5 > ] mt) .
 eq treeError = (mt [ "a", < 1, 3 > ] mt) ["d", < 1, 1 > ]
                (mt [ "c", < 2, 7 > ] mt) .
endfm
```

We test the specification as follows:

- We first check types by using assertLeqSort and assertSort. We can check both the type of particular terms and the sort of the term obtained after applying a function. In our example we check that treeOK1 has sort BSTree? and least sort BSTree. On the other hand, the least sort of treeError is BSTree?, while inserting in a BSTree should return another BSTree. Note that we test how insert behaves with a tree that does not contain the element being inserted (treeOK1) and a tree that contains it (treeOK2).
- Then, we check the obtained results. We use assertTrue and assertFalse to test the Boolean function inStock, while we use assertEqual to indicate how insert, delete, and oneSold should work when applied to treeOK1.

```
(munit FTEST is
  assertLeqSort(treeOK1, BSTree?)
  assertSort(treeOK1, BSTree)
  assertSort(treeError, BSTree?)
  assertSort(insert(treeOK1, "b", < 1, 1 >), BSTree)
  assertSort(insert(treeOK2, "b", < 1, 1 >), BSTree)

  assertTrue(inStock(treeOK1, "a"))
  assertFalse(inStock(treeOK1, "f"))

  assertEqual(insert(treeOK1, "b", < 1, 1 >), treeOK2)
  assertEqual(delete(treeOK1, "c"), treeOK3)
  assertEqual(oneSold(treeOK1, "c"), treeOK4)
endu)
```

When executed, the tool indicates one of the test cases failed. In this case the insertion is not typed as expected because the `insert` function was not completely reduced, since we did not define one of the equations. The output shows how the reduction stopped when facing the missing case:

```
12 test cases were executed.
1 failures.

assertSort(treeOK1,BSTree) passed.

assertLeqSort(treeOK1,BSTree?) passed.

assertSort(treeError,BSTree?) passed.

assertSort(insert(treeOK1,"b",< 1,1 >),BSTree) passed.

assertSort(insert(treeOK2,"b",< 1,1 >),BSTree) failed.
The normal form is (mt["a",< 1,3 >] insert(mt["b",< 1,1 >]mt,"b",< 1,1 >))
                    ["c",< 2,7 >](mt["e",< 3,5 >]mt)
Its sort is BSTree?
...
```

We assume we fix the trees and continue with the example. We use the module above to simulate a shop where buyers try to purchase items and sellers provide new products to the shop. The module `SHOP` defines the sort `People` as a set of `Person`:

```
mod SHOP is
 pr BSTREE .

 sorts Person People Shop .
 subsort Person < People .

 op nobody : -> People [ctor] .
 op __ : People People -> People [ctor assoc comm id: nobody] .
```

Then, we define sellers and buyers. Buyers only take as argument the identifier of the product they want to buy (we assume they buy one unit), while sellers have the identifier of the item, the number of units, and the price they ask for the product. The shop consists of a set of people, the binary search tree storing the available products, and the current money:

```
 *** Product Quantity Price
 op seller : String Nat Nat -> Person [ctor] .
 op buyer : String -> Person [ctor] .

 op [_|_,_] : People BSTree Nat -> Shop [ctor] .
```

Rule `buyer` removes a buyer from the people once he/she buys the item indicated by its argument, given it is available; the shop uses the function `oneSold` to decrease the quantity of the product and the money of the shop is updated. In turn, the shop obtains more products from sellers with the rule `seller`. This rule indicates that, if the shop has enough money, it adds the products to the inventory, asking buyers to get it for twice the price it paid to the seller:

```
 crl [buyer] : [buyer(S) P | T, M]
  => [P | oneSold(T, S), M + PR]
  if inStock(T, S) /\
     PR := getPrice(T, S) .

 crl [seller] : [seller(S, Q, PR) P | T, M]
  => [P | insert(T, S, < Q, 2 * PR >), sd(M, PR * Q) ]
  if M >= PR * Q .
endm
```

The module `STEST` defines an initial shop with two sellers, offering items `a` and `b`, a `buyer` who wants an item `a`, and `10` as initial money. Note that two different final states are possible: if the shop takes `a` first it will be able to sell it and then it will have money to obtain `b`, while buying `b` first would prevent further rewrites:

```
mod STEST is
 pr FTEST .
 pr SHOP .

 op init-shop : -> Shop .
 eq init-shop = [seller("a", 1, 5) seller("b", 1, 5) buyer("a") | mt, 10] .
endm
```

The tests for this module check that it is possible to reach a final state where `nobody` remains in the shop and it has `5` as money and `"b"` in stock, but it is not possible to reach a state where the money is `20`:

```
(munit STEST is
  hasSolution(init-shop, ([nobody | T:BSTree, 5]), !, unbounded,
              inStock(T:BSTree, "b"))
  noSolution(init-shop, ([P:People | T:BSTree, 20]), +, unbounded, nil)
endu)
```

Instead of simulating the shop we can implement an input/output application extending the Loop Mode for managing it. We would define two commands for buying and adding products:

```
  op sold_ : @Token@ -> @ShopCommand@ [ctor] .
  op add_,_,_ : @Token@ @Token@ @Token@ -> @ShopCommand@ [ctor] .
```

as well as two attributes for storing the current `tree` and `money`:

```
 op tree :_ : BSTree -> Attribute [ctor] .
 op money :_ : Nat -> Attribute [ctor] .
```

The rules dealing with these commands check whether the conditions hold (e.g. there is enough money or the item required is available, respectively) and update the tree and the money accordingly, showing the corresponding message. We show the rule for `add`, which contains a bug and updates the money erroneously (it is decreased when no units are available and unchanged otherwise); we refer to the webpage above for details about the rest of rules and the correct implementation.

```
crl [add] :
    < O : SDC | input : ('add_`,_`,_['token[T], 'token[T'],
                'token[T'']]),  output : nil, tree : BST, money : M, AtS >
 => < O : SDC | input : nilTermList, output : QIL, tree : BST',
                money : M', AtS >
 if Q := downQid(T) /\
    S := string(Q) /\
    QTY := rat(string(downQid(T')), 10) /\
    PR := rat(string(downQid(T'')), 10) /\
    B := M >= QTY * PR /\
    BST' := if B
            then insert(BST, S, < QTY, 2 * PR >)
            else BST
            fi /\
    M' := if B
          then M               *** Should be sd(M, QTY * PR)
          else sd(M, QTY * PR) *** Should be M
          fi /\
    QIL := if B
           then '\n '\! 'Database 'updated. qid(QTY) 'units 'of qid(S)
                'added. '\n
                'It 'was 'bought 'at qid(PR) 'euros. '\n
                'It 'will 'be 'sold 'at qid(2 * PR) 'euros. '\o '\n
           else '\n '\! '\r 'Error: '\o 'Not 'enough 'money. '\n
           fi .
```

In this case we test the specification by starting the loop and first checking that we do not have a `cake` in the inventory and the value of `money` is `10`. Then, we add one cake and pay `5` for it (it will be sold at `10` later). Once the cake has been introduced, we check that in fact the inventory has a `cake` and only `5` as `money`. Then, we indicate that we have sold the cake, hence checking that it disappears from the inventory and we have a final amount of `15` in `money`:

```
(munit SHOP-INIT is
  loop(init-shop)

  assertFalse(inStock(@tree, "cake"))
  assertEqual(@money, 10)

  command(add cake, 1, 5)

  assertTrue(inStock(@tree, "cake"))
  assertEqual(@money, 5)

  command(sold cake)

  assertFalse(inStock(@tree, "cake"))
  assertEqual(@money, 15)
endu)
```

When executed, the tool indicates that the tests for `@money` fail. The first one states that, after adding a cake, the money is `10` although the user expects it to be `5`. In the second case the money is `20`, which was unexpected but makes sense if we consider the money after introducing the cake was `10`, so the user would check the rule for `add`:

```
6 test cases were executed.
2 failures.
...
command (add cake,1,5) executed.

assertEqual(@money,5) failed.
First term reduced to 10
Second term reduced to 5
...
command (sold cake) executed.

assertEqual(@money,15) failed.
First term reduced to 20
Second term reduced to 15
...
```

4 Implementation

In this section, we present some details of the implementation of MUnit. The source code of the tool is available at https://github.com/ariesco/MUnit.

MUnit extends Full Maude by defining a new type module and its corresponding commands. Following the ideas in [13], the module `MUNIT-SIGNATURE` defines the syntax of MUnit, which will be used later to parse the commands introduced by the user (via its meta-representation in module `META-MUNIT-SIGN`). The module `MUNIT` defines the rule for parsing the only available input in MUnit, MUnit modules:

```
 crl [munit] :
     < O : MUDC | db : DB, input : ('munit_is_endu[T, T']),
                  output : nil, AtS >
  => < O : MUDC | db : DB, input : nilTermList, output : printUR(UR),
                  AtS >
  if UR := procMUnit(T, T', DB) .
```

This rule uses the auxiliary function `procMUnit` to parse the tests and returns a `UnitResult`, which consists of a tuple indicating the number of tests that passed and failed, a `QidList` with the information for the user, and the inner loop:

```
op [_,_,_,_] : Nat Nat QidList Term -> UnitResult [ctor] .
```

The function `procMUnit` computes the module from the module expression, initializes the loop, and uses the function `procProps` to traverse the tests in the module and execute them:

```
op procProps : Module Term OpDeclSet Bool Database Term -> UnitResult .
ceq procProps(M, '__[T, T'], VDS, B, DB, SYS) =
                                   [N + N', N1 + N1', QIL QIL', SYS'']
 if [N, N1, QIL, SYS'] := procProps(M, T, VDS, B, DB, SYS) /\
    [N', N1', QIL', SYS''] := procProps(M, T', VDS, B, DB, SYS') .
```

Among all possible tests, we briefly present below how the inner loop is updated. This inner loop consists of a meta-represented loop that is initialized by the `loop` instruction and is later manipulated when a `command` instruction is found. The equation below shows (a simplification of) how `procProps` evaluates the `command` instruction given the current module, the instruction, and the current loop. It first transforms, in the first two conditions, the command the user wants to introduce into the loop from a list of quoted identifiers into a term. Then, it uses the auxiliary function `sysInput` to introduce that term as the first argument of the loop. Then, we use `metaRewrite` to execute the thus obtained loop and generate the information that will be shown to the user. Finally, the result is a tuple containing the tests that passed and failed (`0` in both cases, since this instruction is not a test), the message, and the updated loop. Note that other rules are in charge of handling errors in this process.

```
ceq procProps(M, 'command['bubble[T]], SYS) = [0, 0, QIL', SYS'']
 if QIL := downQidList(T) /\
    T' := upTerm(QIL) /\
    SYS' := sysInput(SYS, T') /\
    SYS'' := getTerm(metaRewrite(M, SYS')) /\
    QIL' := '\! 'command '\b ' '`( QIL '`) ' '\o '\! 'executed. '\o '\n .
```

5 Related Work

The maturity of a programming language is, to an extent, related to its tool support. In this way, mainstream languages (mostly imperative) have integrated development environments where debuggers, test tools, integration tools, etc. are integrated, while other languages have limited support of these tools. In fact, as argued in [25], the application of declarative languages out of the academic world is inhibited by the lack of convenient auxiliary tools. However, during the last decades the distance in this subject between declarative and imperative languages has been reduced thanks to the implementation, among other tools, of debuggers and testing tools adapted to the particular features of these languages. We focus in this section in property-based testing and unit testing, widely used in many declarative languages.

An important step in the field of testing was the development of QuickCheck [14], a property-based testing tool for Haskell. Property-based testing is a two-step technique: first, the user states some properties that the functions under test must fulfill and where some of the inputs have been replaced by so called *generators* for the corresponding data structures. These generators are used during the second step to build several inputs (usually several hundreds) to check whether the property holds. Generators for un-structured datatypes such as natural numbers usually just require to build random elements in a given range, while generators for structured datatypes such as lists are built on top of the generators for their elements, including options for the size of the data structure and the relation between elements.[3] This simple philosophy allows property-based testing tools to check properties on hundreds of inputs in a short time, and experiments have shown that property-based testing is in general as good as more complex testing techniques, so the QuickCheck approach has been implemented in other declaratives languages such as Erlang (PropEr [18] and QuviQ [2]), Scala (ScalaCheck [16]), Curry (EasyCheck [7]), and Prolog (PrologCheck [1]), among many others. In contrast to these tools, the Maude property-based generator tries to falsify the property using narrowing, which allows the tool to find corner cases that can be missed by the tools above but results in a less efficient implementation for big specifications.

Besides property-based testing, several languages support some kind of unit testing. In particular, languages like Erlang (EUnit [6]), Scala (ScalaTest [24]), Prolog (Plunit [26]), and Curry (CurryTest [12]) support this kind of tests. In general, unit tests are used in these languages to state equality between terms. However, the case of Prolog and Curry is interesting in our case because they support non-determinism, just like Maude system modules. In their case they require the user to indicate the set of possible solutions, so each of the solutions are contained in the set.

It is also important to note that property-based testing and unit tests are complementary, as illustrated by the number of languages supporting both approaches. In general, while property-based testing is used to test the main functions, which have associated properties, unit tests are (ideally) defined for checking all functions, so changes in the implementation can be easily checked when integrated.

6 Conclusions and Ongoing Work

We have presented a unit testing framework for Maude. It supports functional and system modules, as well as extensions of the Loop Mode, in particular Full

[3] Most of the tools also include specific strategies for corner cases, such as empty lists.

Maude object-oriented modules and interactive tools. Since Full Maude extensions are in general difficult to test and maintain, one of the main features of MUnit is the support for this kind of applications.

We are currently working to extend MUnit modules with standard Maude declarations. In this way users will be able to define constants and functions to ease the testing process. It would be also interesting to identify tests and support messages in them, so tests would carry information about its intended coverage. It is also interesting to allow users to name tests, so complex tests can be easily identified.

Finally, we are also interested on integrating unit testing and the Maude declarative debugging [21]. Since declarative debugging asks questions about the computation to the user to find the bug, it would generate unit tests that can be later used. Likewise, some answers would be answered by unit tests, saving time and effort to the user.

References

1. Amaral, C., Florido, M., Santos Costa, V.: PrologCheck – property-based testing in prolog. In: Codish, M., Sumii, E. (eds.) FLOPS 2014. LNCS, vol. 8475, pp. 1–17. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07151-0_1
2. Arts, T., Castro, L.M., Hughes, J.: Testing Erlang data types with QuviQ QuickCheck. In: Teoh, S.T., Horváth, Z. (eds.) Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG, pp. 1–8. ACM (2008)
3. Beizer, B.: Software Testing Techniques. Dreamtech, India (2002)
4. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theor. Comput. Sci. **236**, 35–132 (2000)
5. Britton, T., Jeng, L., Carver, G., Cheak, P., Katzenellenbogen, T.: Reversible debugging software. University of Cambridge-Judge Business School, Technical report (2013)
6. Carlsson, R., Rémond, M.: EUnit: a lightweight unit testing framework for Erlang. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, p. 1. ACM (2006)
7. Christiansen, J., Fischer, S.: EasyCheck — test data for free. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 322–336. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78969-7_23
8. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of Haskell programs. In: ACM SIGPLAN Notices, pp. 268–279. ACM Press (2000)
9. Clavel, M., et al.: Maude Manual (Version 2.7), March 2015. http://maude.cs.illinois.edu/w/images/1/1a/Maude-manual.pdf
10. Clavel, M., et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
11. Clavel, M., Durán, F., Hendrix, J., Lucas, S., Meseguer, J., Ölveczky, P.: The maude formal tool environment. In: Mossakowski, T., Montanari, U., Haveraaen, M. (eds.) CALCO 2007. LNCS, vol. 4624, pp. 173–178. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73859-6_12
12. CurryTest. https://www-ps.informatik.uni-kiel.de/currywiki/tools/currytest

13. Durán, F., Ölveczky, P.C.: A guide to extending Full Maude illustrated with the implementation of Real-Time Maude. In: Roşu, G. (ed.) Proceedings of the 7th International Workshop on Rewriting Logic and its Applications, WRLA 2008, Electronic Notes in Theoretical Computer Science, vol. 238(3), pp. 83–102. Elsevier (2009)
14. Hughes, J.: Software testing with QuickCheck. In: Horváth, Z., Plasmeijer, R., Zsók, V. (eds.) CEFP 2009. LNCS, vol. 6299, pp. 183–223. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17685-2_6
15. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992)
16. Nilsson, R.: Scalacheck: The Definitive Guide. IT Pro, Artima Incorporated (2014)
17. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order Symb. Comput. **20**, 161–196 (2007)
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.: Using narrowing to test maude specifications. In: Durán, F. (ed.) WRLA 2012. LNCS, vol. 7571, pp. 201–220. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34005-5_11
20. Riesco, A., Ogata, K., Futatsugi, K.: A Maude environment for CafeOBJ. Formal Aspects Comput. **29**(2), 309–334 (2016)
21. Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R.: Declarative debugging of rewriting logic specifications. J. Logic Algebraic Program. **81**(7–8), 851–897 (2012)
22. Runeson, P.: A survey of unit testing practices. IEEE Softw. **23**(4), 22–29 (2006)
23. Tretmans, J.: Conformance testing with labelled transition systems: implementation relations and test generation. Computer Netw. ISDN Syst. **29**(1), 49–79 (1996)
24. Venners, B.: Scalatest 3.0.5 (2018). http://www.scalatest.org/
25. Wadler, P.: Why no one uses functional languages. SIGPLAN Not. **33**(8), 23–27 (1998)
26. Wielemaker, J.: Prolog unit tests. http://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/plunit.html%27)