Maude manual

Adrián Riesco

Theory of Programming Languages

Academic year 2020/21

Contents

1 Introduction

Formal methods in computer science rely in four principles: (1) the existence of a clear and concise semantic framework of the behavior of software systems; (2) the existence of a clear, detailed, and non-ambiguous representation that allows developers to specify software systems and associate them a particular semantics; (3) the existence of a clear and detailed formalism where the properties of systems can be defined and where the validity of these properties with respect to a particular semantics can be established; and (4) the existence of techniques to verify these properties. Thus far, formal methods have been applied, on the one hand, as part of the software development process, producing secure and reliable software by construction, making use of its above-mentioned semantics. On the other hand, they have been used once the software development process has finished, verifying whether the system fulfills some given properties.

In the first case, the so-called declarative programming (or specification) languages (such as the algebraic languages Maude and OBJ, the functional languages Haskell and ML, the logic languages Prolog and Gödel, or the functional-logic languages Curry and TOY) are very appropriate for developing software, in contrast to more difficult to read languages such as UML, which are presented in natural language and might only include semi-formal specifications that cannot be automated. In the second case, specification and verification techniques based on different formalisms, like reachability and model checking, have been successfully used not only in academy, but also in industry.

This second approach is the followed by Maude, which is not an alternative to standard programming languages but a specification language to specify and verify systems. For example, Maude has been used to reason about communication protocols like FireWire (IEEE 1394), the CORBA and SOAP platforms, the UML metamodel, programming languages like Java, and it has even used at NASA in the development of software for recognising objects in space.

2 Maude

The programming language Maude [**?**, **?**] uses rewrite rules in a similar way to functional languages such as Haskell [**?**], ML [**?**], Scheme [**?**], and Lisp [**?**]. In particular, Maude is based on rewriting logic, which allows programmers to define several complex computation models such as concurrent programming or object-oriented programming. For example, Maude allows programmers to specify objects directly in the language, following a declarative approximation to object-oriented programming that is not available in imperative languages like C++ or Java nor in declarative languages such as Haskell.

The development of Maude relies on an international team whose purpose is the design of a common declarative platform for research, teaching, and implementation. More information is available at:

`http://maude.cs.uiuc.edu`

In the following we present the main features of Maude. A complete manual is also available [**?**], as well as a primer [**?**] in the url above. A Maude book [**?**] is also available in the library and online (accessing from the UCM) at:

`http://www.springerlink.com/content/p6h32301712p`

3 Programming in Maude

In this section we introduce Maude syntax by means of examples. We start by presenting the simplest modules, functional modules, and then we move to system modules. For all modules we briefly present the available commands. Once the different modules have been explained we will present how to use searches to check invariants in finite state spaces. Finally, we will briefly present the Maude interpreter, the most common errors and some predefined modules.

3.1 Functional modules

In this section we present functional modules by means of examples: we first specify natural numbers following Peano axioms, then we will present how to specify stacks, and, finally, we will show how to work with lists.

It is worth taking into account that in this section we talk about equations, which have two important constraints:[1] they must be terminating, that is, no infinite computations are allowed, and confluent, that is, they must return the same final result independently of the order in which they are applied.

[1] Actually, they have more constraints that will not be discussed this course.

3.1.1 Natural numbers

Our first example, available in `peano_nat.maude`, are the natural numbers following the Peano notation.[2] First, we create a functional module called `PEANO` using the keyword `fmod`, which indicates that the module starts here (we will see it finishes with `endfm`), followed by the module name, in this case `PEANO`,[3] and finally the keyword `is`:

```
fmod PEANO is
```

We need now a datatype. In this simple example we only define the type `PeanoNat`, which requires the keyword `sort` to state we plan to define a type, followed by the type name, `PeanoNat`, and finished with a dot.

Statements are usually finished in Maude with a whitespace followed by a dot, so pay attention to this fact when programming.

```
  sort PeanoNat .
```

Next, we define constructors for the previous datatypes. In this case the constructors are 0 and successor. 0 is defined with the following syntax:

```
  op 0 : -> PeanoNat [ctor] .
```

First, we use the keyword `op`, which indicates we are defining an *operator*. Then, we write the symbol we want to use to represent our constructor, in this case `0`, although any other symbol like `zero`, `myZero`, or `foo` would be valid as well. After this symbol we find a colon, which indicates that the symbol has finished, and then we would find a list of arguments, empty in this case, and the symbol `->` that indicates this list has finished. Finally, we have the type of the constructor (`PeanoNat`), a whitespace, a list of attributes enclosed in square brackets, and a dot to finish the operator declaration. In this simple case we only have the `ctor` attribute, which indicates the operator is a constructor, but we will present more attributes later. The successor operator is slightly more complex:

```
  op s : PeanoNat -> PeanoNat [ctor] .
```

In this case the constructor receives an argument of type `PeanoNat`. This means that we can create terms like `s(0)` (which stands for 1), `s(s(0))` (2 in the standard notation), etc.

Once we have defined the constructors we can define functions. However, it is worth defining variables first. In this simple case it is enough to define two variables of sort `PeanoNat`:

```
  vars N M : PeanoNat .
```

As shown above, we use the keyword `vars` (`var` is also valid, even for several variables) followed by a list of identifiers separated without commas, a colon `:`, the type name and finishing with a dot. The simplest function we can define for natural numbers is addition:

```
  op _+_ : PeanoNat PeanoNat -> PeanoNat [assoc comm] .
```

[2] Note, as explained in Section 3.6, that Maude has a predefined module for natural numbers, so this module is just an introduction to Maude and will not be used in the future.

[3] By convention we use capital letters for module names.

In this case the operator has two underlines (`_`). This element is not part of the operator identifier, but a placeholder that indicates where arguments are placed. This function receives two arguments of type `PeanoNat` that will be placed instead of the underscores, so we can write, for example, `s(0) + s(s(0))`. Finally, this operator is not a constructor, but it has two attributes:

- The attribute `assoc` indicates that the function is associative, that is, $(N+M)+P = N+(M+P)$ and no parentheses are required.
- The attribute `comm` indicates that the function is commutative, that is, $N+M = M+N$. This attribute simplifies the definition of the behavior of functions, because just one stands many others. We will discuss more complex examples later.

The semantics of functions is defined by means of *equations*. Equations are usually defined by following structural induction in one argument, defining an equation for each constructor. In this case, we can apply induction indistinctly in both arguments (because the function is commutative) and we need two equations (because we have two constructors):

```
*** Case 1: constructor 0
eq [s1] : 0 + N = N .

*** Case 2: constructor sucesor
eq [s2] : s(N) + M = s(N + M) .
```

where we have used comments to identify the cases.[4] Moreover, we have used labels to identify each equation. These labels are enclosed by square brackets and followed by `:` and are optional; the module would work in the same way if we do not define them. We observe that equations are defined with the keyword `eq` and using `=` to identify the terms; it finishes with a dot, as usual. Equation `s1` indicates that adding `0` to any natural number, identified by the variable `N`, returns the same number. Equation `s2` indicates that, if the first argument is defined by means of the successor constructor, then we can add the argument (which is obviously smaller than the whole term) with the other argument and apply the successor constructor to the result. In this way the argument will eventually reach the base case in `s1`.

We define multiplication in a similar way:

```
op _*_ : PeanoNat PeanoNat -> PeanoNat [assoc comm] .
eq 0 * N = 0 .
eq s(N) * M = M + (N * M) .
```

By default, Maude imports the predefined module `BOOL`, which allows specifiers to use the sort `Bool` for Boolean values (more information is available in Section 3.6). Then, we can define a Boolean function that, given a natural numbers, indicates whether the number is positive:

```
op isPositive : PeanoNat -> Bool .
```

The equations for this function distinguish between the two constructors and use `true` and `false` as result:

[4] Block comments are defined with syntax `***( ... )`.

```
eq isPositive(0) = false .
eq isPositive(s(N)) = true .
```

Finally, the module finishes with `endfm`:

```
endfm
```

The functions defined in functional modules are executed (see Section 3.4 to know how to start Maude) with the `reduce` command, abbreviated as `red`. Hence, we can add 1 and 2 (written as `s(0) + s(s(0))` in our notation) as follows: (**The command also finishes with a dot**):

```
Maude> red s(0) + (s(s(0))) .
reduce in Peano : s(0) + s(s(0)) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result PeanoNat: s(s(s(0)))
```

In this first example it is interesting to see how Maude works. In this case we have the following derivation, where we show for each step the equation applied and the subterm involved in the reduction:

$$\underline{\mathtt{s}(0) + \mathtt{s}(\mathtt{s}(0))} \overset{\mathtt{s2}}{\rightsquigarrow} \mathtt{s}(\underline{0 + \mathtt{s}(\mathtt{s}(0))}) \overset{\mathtt{s1}}{\rightsquigarrow} \mathtt{s}(\mathtt{s}(\mathtt{s}(0)))$$

We can also execute `isPositive` as follows:

```
Maude> red isPositive(s(s(0))) .
reduce in Peano : isPositive(s(s(0))) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

3.1.2 Stacks

Once we have mastered the basic Maude syntax we can create stacks of natural numbers (the code is available as `stack.maude`). We first create the `STACK` module:

```
fmod PILA is
```

We first require natural numbers. Although we defined basic natural numbers in the previous section it is more convenient to use the predefined module for natural numbers. The `NAT` module, as explained in Section 3.6, defines the type `Nat` to work with natural numbers, so we import it with the keyword `pr`:

```
pr NAT .
```

Then, we define the type `Stack`, which has `mt` and `push` as constructors, and define variables for numbers and stacks:

```
sort Stack .

op mt : -> Stack [ctor] .
op push : Nat Stack -> Stack [ctor] .

var N : Nat .
var P : Stack .
```

It is worth paying attention to the `push` constructor, which receives a natural number as first argument. A stack that only contains 7 is represented as `push(7, mt)`, while the stack that puts a 3 on top of the 7 is represented as `push(3, push(7, mt))`.

The function `pop` removes the top of the stack. If the stack is empty it remains as `mt`:

```
op pop : Stack -> Stack .
eq pop(mt) = mt .
eq pop(push(N, P)) = P .
```

However, dealing with errors is not so easy for the `top` function. We can define a partial operator instead of a total one; Maude will consider that the functions fails when no equations can be applied. To define a partial function instead of `->` in the definition we need to use `~>`:

```
op top : Stack ~> Nat .
eq top(push(N, P)) = N .
endfm
```

We can test the module as follows:

```
Maude> red top(pop(push(3, push(7, mt)))) .
reduce in STACK : top(pop(push(3, push(7, mt)))) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 7
```

In this case the derivation is:

$$\texttt{top}(\underline{\texttt{pop(push(3, push(7, mt)))}}) \rightsquigarrow \underline{\texttt{top(push(7, mt))}} \rightsquigarrow 7$$

3.1.3 Lists

Our last functional module, available as `lists.maude`, presents how to specify lists of persons. We start defining the module `PERSON`:

```
fmod PERSON is
 pr STRING .

 sort Person .

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

 vars S S' : String .
 vars N N' : Nat .
```

We use the predefined module `STRING` to define persons as pairs with name, of sort `String`, and the money they have, of sort `Nat`. We also define the appropriate variables. We define next operators for obtaining the maximum and the minimum of two persons with respect to the money they carry. These functions are commutative so it is enough with one conditional equation to define each of them:

```
ops max min : Person Person -> Person [comm] .

ceq [max] : max(< S, N >, < S', N' >) = < S, N >
 if N >= N' .
ceq [min] : min(< S, N >, < S', N' >) = < S, N >
 if N <= N' .
```

As we have said, the equations `max` y `min` are conditional. Conditional equations are declared with the keyword `ceq` instead of `eq`. After defining the identity we must give a list of conditions, composed with the `/\` operator.

👀 Maude also provides a `if_then_else_fi` operator, which must be distinguished from conditional equations. In order to use this operator we use non-conditional equations[5] and use it on the right-hand side of the equation. For the functions above we would have:

```
ops max min : Person Person -> Person [comm] .

eq [max] : max(< S, N >, < S', N' >) = if N >= N'
                                       then < S, N >
                                       else < S', N' >
                                       fi .
eq [min] : min(< S, N >, < S', N' >) = N <= N'
                                       then < S, N >
                                       else < S', N' >
                                       fi .
```

We plan to define ordered lists in our next module using the money as ordering criteria, so it is convenient for us to define the notion of "smaller than or equal to" and "bigger than":

```
  ops _<=_ _>_ : Person Person -> Bool .
  eq < S, N > <= < S', N' > = N <= N' .
  eq < S, N > > < S', N' > = N > N' .
```

Finally, we define some constants to ease testing:

```
  ops a b c d : -> Persona .
  eq a = < "a", 100 > .
  eq b = < "b", 80 > .
  eq c = < "c", 150 > .
  eq d = < "d", 10 > .
endfm
```

The `LIST` module imports the `PERSON` module and defines the types `List`, standing for lists, and `ListOrd`, standing for ordered lists:

```
fmod LIST is
 pr PERSON .

sorts List ListOrd .
```

We introduce now the notion of *subtype*. Subtypes, defined in Maude with the keywrod `subsort` and the operator `<`, state that all the elements defined for the lesser sort has the bigger sort as well. In this particular case we indicate that an element with sort `Person` is also a singleton ordered list. Likewise, an ordered list is a general list (👀 subsorts also finish with dot):

```
  subsort Person < ListOrd < List .
```

[5] Of course, it is possible to use the `if` statement with conditional equations, but we leave this combination for more advanced users.

Once we have defined these sorts we face a problem: the empty list (defined as `lv`) is ordered but, which constructor could we define for non-empty ordered lists? The only option is to use the same constructor used for general lists and define later ordered list by means of their properties:

```
op lv : -> ListOrd [ctor] .
op __ : List List -> List [ctor assoc id: lv] .

vars P P' : Person .
vars L L' : List .
var  LO : ListOrd .
```

The constructor for non-empty lists has some interesting details:

- It receives as arguments two elements of sort `List`. However, which is the form of these elements? As we explained above, singleton lists are just persons. Once we have several persons we can create larger lists. The proper way to create these lists is explained below.
- The syntax `__` indicates that 2 lists put together (empty syntax) create a new list. That is, given the persons `a` and `b` from the `PERSON` module we can create the list `a b`. The intuition is that we place `a` in the position given by the first underscore, then we write a whitespace, and then we place `b` in the position given by the second underscore. If we want to add now a third person `c` we would create `a b c` (the list `a b` corresponds to the first underscore, while `c` corresponds to the second one).
- The empty list `lv` works as the unit element for lists. This means that Maude considers that any list `L` is equal to the lists `L lv` and `lv L`. This feature is useful to define simpler equations, as we will see later.

Now we need to define the properties that ordered lists must fulfill. We will use *membership axioms* to define them, written in Maude with keywords `mb` and `cmb`, for unconditional and conditional ones, respectively. Then we will write a term, `:`, and the corresponding type. In our case we need a conditional axiom stating that (i) the first element is smaller than or equal to the second one and (ii) the rest of the list is also ordered. This second condition is, in turn, a membership condition, with syntax `:`, as we used for the axiom itself:

```
cmb P P' L : ListOrd
 if P <= P' /\
    P' L : ListOrd .
```

Note that the empty list is ordered by definition, while singleton lists are ordered by the subtype definition.

We can start defining now the functions in the module. The head of a list is a partial function defined as follows:

```
op head : List ~> Person .
eq head(P L) = P .
```

In this equation we implicitly use the unit element. Why does this equation work for singleton lists? Because the person is bound to the variable `P` and is returned, while the variable `L` is bound to the empty list. This equation can be only applied if the list is not empty. Likewise, the tail of a list is computed as follows:

```
op tail : List ~> List .
eq tail(P L) = L .
```

The size of the list is defined as:

```
op size : List -> Nat .
eq size(lv) = 0 .
eq size(P L) = s(size(L)) .
```

The function `member?`, which checks whether an element is a member of the list, can be defined in a tricky way. If the element is part of the list it is enough to use variables surrounding the element we are looking for to match the rest of the list. Otherwise, we return `false`, which is indicated by using the `owise` attribute, which stands for *otherwise*:

```
op member? : List Person -> Bool .
eq member?(L P L', P) = true .
eq member?(L, P) = false [owise] .
```

In a similar way, we can define bubblesort: if the consecutive elements are not ordered we can interchange them and keeping ordering. Once all elements are ordered the list is returned:

```
op order : List -> List .
ceq order(L P P' L') = order(L P' P L')
 if P > P' .
eq order(L) = L [owise] .
```

Finally, we show how to insert in an ordered fashion. If we insert in the empty list the singleton list is returned. Otherwise, we compare the value to be inserted with the head of the list and distinguish cases:

```
op insert-sort : ListaOrd Persona -> ListaOrd .
eq insert-sort(lv, P) = P .
eq insert-sort(P L, P') = if P <= P'
                          then P insert-sort(L, P')
                          else P' P L
                          fi .
```

We can use the function above to define the order-by-insertion function:

```
op sort-by-insertion : ListaOrd -> ListaOrd .
eq sort-by-insertion(lv) = lv .
ceq sort-by-insertion(P L) = insert-sort(L', P)
 if L' := sort-by-insertion(L) .
endfm
```

It is worth noticing the pattern matching condition, with syntax `:=`. This condition is used as an assignment, that is, it binds, if possible, the values to the variables and then the evaluation of the rest of conditions continues.

3.2 System modules

System modules extend function modules with *rewrite rules*, which stand for state transitions. Roughly speaking, functional modules are used to define the data structures while system modules are used to define the behavior of the system. The main advantage of rules with respect to equations is that, in contrast with the restrictions given in Section 3.1, rules can be non-terminating and non-confluent.

As system module example we present the jar problem, as presented in Die Hard 3©:

`http://www.wikihow.com/Solve-the-Water-Jug-Riddle-from-Die-Hard-3`

The problem is defined as follows: we have 3 jars with capacity for 3, 5, and 8 liters, and we have an infinite amount of water. We want to obtain 4 liters in any of the jars but we can only fill them, empty them, and transfer water from one to the other (without exceeding the capacity of the jars). Which are the movements we are supposed to perform to reach the solution? To solve this problem (the code is available in `die-hard.maude`) we define the `DIE-HARD` module as follows:

```
mod DIE-HARD is
 protecting NAT .
 sorts Jar JarSet .
 subsort Jar < JarSet .

 op jar : Nat Nat -> Jar [ctor] .   *** Capacity / Current content
 op __ : JarSet JarSet -> JarSet [ctor assoc comm] .

 vars M1 N1 M2 N2 : Nat .
```

Note first that the module starts with the keyword `mod` but the rest of the elements are equal to the ones in functional modules. In our case jars are pairs of natural numbers, where the first element stands for the capacity and the second one for the current amount. Moreover, the sort `JarSet` is defined for sets of jars. We define a constant `initial` of sort `JarSet` stating that all jars are empty in the initial state:

```
 op initial : -> JarSet .
 eq initial = jar(3, 0) jar(5, 0) jar(8,0) .
```

State transitions are defined by means of rewrite rules. Rules have a syntax similar to equations, using `rl` instead of `eq` (`crl` for conditional rules) and `=>` instead of `=`. Labels are not required but very useful for later analyses. In our problem we start defining a rule `empty` for emptying a jar:

```
 rl [empty] : jar(M1, N1) => jar(M1, 0) .
```

We follow the same idea to fill a jar:

```
 rl [fill] : jar(M1, N1) => jar(M1, M1) .
```

We also have 2 rules to transfer water between jars. The first one, `transfer1`, illustrates the case where the jar has enough capacity to receive all the water from the other one:

```
 crl [transfer1] : jar(M1, N1) jar(M2, N2)
  => jar(M1, 0) jar(M2, N1 + N2)
  if N1 + N2 <= M2 .
```

In the second rule, `transfer2`, we fill the jar while the other one still has some water remaining:

```
 crl [transfer2] : jar(M1, N1) jar(M2, N2)
  => jar(M1, sd(N1 + N2, M2)) jar(M2, M2)
  if N1 + N2 > M2 .
endm
```

The module finishes, as shown above, with the keyword `endm`. Once the modules is loaded we can execute it in different ways. The basic command is `rewrite`, abbreviated as `rew`, that applies equations and rules to the term given as argument. However, in this example the execution does not finish (for example, it is always possible to empty a jar), so we need a more powerful command. For example, we can ask Maude to apply at most 10 rewrite steps writing that number in square brackets:

```
Maude> rew [10] initial .
rewrite [10] in DIE-HARD : initial .
rewrites: 139 in 0ms cpu (0ms real) (1022058 rewrites/second)
result JarSet: jar(3, 0) jar(5, 0) jar(8, 0)
```

The result is not very interesting, because it seems nothing happened. In fact, it is very likely that the system just applied `empty` 10 times, which does not modify the initial term. In order to apply different rules we can use the `frew` command, which performs a *fair rewrite*:

```
Maude> frew [10] initial .
frewrite in DIE-HARD : initial .
rewrites: 18 in 0ms cpu (0ms real) (~ rewrites/second)
result JarSet: jar(3, 0) jar(5, 5) jar(8, 8)
```

This time we obtained a new term, but it is not very useful to analyze how to reach a given configuration. For this analysis we need the `search` command, which checks whether a given result is reachable. If reachable, it shows the path (that is, the sequence of rules) that must be used. In our case we want just one solution (indicated as `[1]`) with any of the jars containing 4 liters (given as a pattern); moreover, we indicate that we want the search to apply 0 or more rewrite steps (search arrow `=>*`), so if the initial state fulfills the condition it would be returned as solution (other options would be 1 or more steps, with the arrow `=>+`, and search for final states, with `=>!`). We use variables for the pattern (👀 these variables must be defined with its type, it is not possible to use variables from the module) when some values can be omitted, for example for the jar capacity.

👀 The variable `B:JarSet` is very important but most of the people forgets about it. This variable is in charge of "collecting" the rest of the jars. This kind of variables are often used to abstract those parts of the solution that are not interesting for the search.

```
Maude> search [1] initial =>* jar(N:Nat, 4) B:JarSet .
search in DIE-HARD : initial =>* B:JarSet jar(N:Nat, 4) .

Solution 1 (state 75)
states: 76  rewrites: 2134 in 0ms cpu (8ms real) (~ rewrites/second)
B:JarSet --> jar(3, 3) jar(8, 3)
N:Nat --> 5
```

We see in the result that we obtained 4 liters in the 5 liters jar and that the solution was found in the 75 state. We can now use the `show path` command to print the path:

```
Maude> show path 75 .
state 0, JarSet: jar(3, 0) jar(5, 0) jar(8, 0)
===[ rl jar(M1, N1) => jar(M1, M1) [label fill] . ]===>
state 2, JarSet: jar(3, 0) jar(5, 5) jar(8, 0)
===[ crl jar(M1, N1) jar(M2, N2) => jar(M1, sd(M2, N1 + N2))
  jar(M2, M2) if N1 + N2 > M2 = true [label transfer2] . ]===>
state 9, JarSet: jar(3, 3) jar(5, 2) jar(8, 0)
===[ crl jar(M1, N1) jar(M2, N2) => jar(M1, 0) jar(M2, N1 + N2)
  if N1 + N2 <= M2 = true [label transfer1] . ]===>
state 20, JarSet: jar(3, 0) jar(5, 2) jar(8, 3)
===[ crl jar(M1, N1) jar(M2, N2) => jar(M1, 0) jar(M2, N1 + N2)
  if N1 + N2 <= M2 = true [label transfer1] . ]===>
state 37, JarSet: jar(3, 2) jar(5, 0) jar(8, 3)
===[ rl jar(M1, N1) => jar(M1, M1) [label fill] . ]===>
state 55, JarSet: jar(3, 2) jar(5, 5) jar(8, 3)
===[ crl jar(M1, N1) jar(M2, N2) => jar(M1, sd(M2, N1 + N2))
  jar(M2,  M2) if N1 + N2 > M2 = true [label transfer2] . ]===>
state 75, JarSet: jar(3, 3) jar(5, 4) jar(8, 3)
```

The `search` command is very powerful, as illustrated in Section 3.3. It also supports condition using the keyword `such that` (abbreviated as `s.t.`). The command above could be rewritten using this syntax as:

```
Maude> search [1] initial =>* jar(N:Nat, M:Nat) B:JarSet s.t. M:Nat == 4 .
search in DIE-HARD : initial =>* B:JarSet jar(N:Nat, 4) .

 Solution 1 (state 75)
 states: 76  rewrites: 2134 in 0ms cpu (8ms real) (~ rewrites/second)
 B:JarSet --> jar(3, 3) jar(8, 3)
 N:Nat --> 5
```

3.3 Checking invariants by search

One of the main advantages of using Maude to specify a system is that we can automatically verify most of its properties. Although many different verification techniques are available, we will focus in checking invariants by using search.

As we know, an *invariant* is a property that holds for all states in a computation. We will analyze invariants for systems with a finite state space. It is easy to check this kind of properties by using the `search` command: it is enough to find a state that does not fulills the property; otherwise the invariant holds.

In the example in Section 3.2 we can define the invariant "we always have 3 jars". To check it we define the module `INVARIANT`, which imports the `DIE-HARD` module:

```
mod INVARIANTE is
  pr DIE-HARD .

  var JS : JarSet .
  var J : Jar .
  var O : Qid .
```

We define the invariant with the `ok` function, which checks that the number of jars is always 3:

```
  op ok : JarSet -> Bool .
  eq ok(JS) = numJars(JS) == 3 .
```

This functions uses an auxiliary function in charge of counting jars. Because we have no unit element we define an equation for non-singleton sets and another one for the singleton set:

```
  op numJars : JarSet -> Nat .
  eq numJars(J JS) = 1 + numJars(JS) .
  eq numJars(J) = 1 .
endm
```

We can now check the invariant. As explained above, we look for states that do *not* fulfill it:

```
Maude> search initial =>* JJ:JarSet s.t. not ok(JJ:JarSet) .

No solution.
```

In fact, there is no state that fulfills the negation of the invariant, that is, it holds for all states and hence it is correct.

3.4 Using Maude

As explained in Section 2, Maude is available (Linux and Mac) at:

`http://maude.cs.uiuc.edu`

There exists a Windows version available at:

`http://moment.dsic.upv.es/`

Both pages have an installation manual. Once installed, we start Maude and find a welcome screen similar to this one:

```
           \||||||||||||||||||/
         --- Welcome to Maude ---
           /||||||||||||||||||\
     Maude 2.7 built: Feb  7 2014 15:12:51
     Copyright 1997-2014 SRI International
            Mon Sep 1 12:02:38 2014
```

We can now load files with the `load` command. For example, `stack.made` is loaded as:

```
Maude> load stack.made
```

If the file is correctly loaded Maude displays the modules that have been introduced. It is also possible to load modules with the `in` command. Moreover, this command can be used in `maude` files to load dependent modules. Finally, it is also possible to start Maude with a list of files to be loaded as follows:

```
$ maude stack.maude
```

Commands will be executed in the module that was loaded in the last time. This is important because, when loading several independent modules, some functions might not be available in the last one. It is possible to change the module with `in`. For example, if we want to execute a command in module `M` we need to use:

```
Maude> red in M : 3 + 4 .
```

Morever, this will change the current module to `M`. Finally, we leave the Maude system with `q`:

```
Maude> q
Bye.
```

3.5 Frequent errors

In this section we present the most frequent errors found when working with Maude:

- Variable names and attributes are separated by whitespaces, not by commas.
- When declaring several operators in the same line the keyword `ops` is required. Otherwise, Maude will understand that a single operator using whitespaces is being declared.
- Commands finish with a dot.
- Infix operators require whitespaces. For example, the term `0+0` is wrong, being the correct term `0 + 0`.
- In some case an unbalanced parenthesis makes Maude to wait for extra input. If you think this might be your case try to execute a simple command, like reducing 1, to check whether it works or the system is blocked.

3.6 Predefined modules

In this section we list some of the most used built-in Maude modules. Remember that they are available in the `prelude.maude` file. The most common Maude modules are:

`BOOL`. The `BOOL` module is imported by default by all Maude modules. It defines the sort `Bool`, the constructors `true` and `false`, and provides Boolean functions such as equality (`_==_`), the unary function `not`, and the binary functions `_and_` and `_or_`.

`NAT`. The `NAT` module provides the types `Zero`, which identifies the number 0; `NzNat`, which identifies positive natural numbers; and `Nat`, which is the union of the previous sets. Morevoer, it provides the standard functions on natural numbers, like addition (`_+_`) or multiplication (`_*_`).

Natural numbers do not implement difference but symmetric difference (`sd`), which subtracts the smallest number from the larger one. For example, we have $\texttt{sd}(3,4) = \texttt{sd}(4,3) = 1$.

`STRING`. The `STRING` module provides the sort `String` to represent strings of characters enclosed by (`""`). For example, the terms `"hello"` and `"123"` have sort `String`.

`QID`. The `QID` module provides the sort `Qid`, which implements quoted identifiers. For example, the terms `'hello` and `'123` have sort `Qid`.