# Playing with Maude

Miguel Palomino, Narciso Martí-Oliet, and Alberto Verdejo

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
{miguelpt,narciso,alberto}@sip.ucm.es

**Abstract.** This paper is an introduction to rule-based programming in Maude. We illustrate in particular the use of operator attributes to structure the state of a system, and the difference between equations and rules. We use mathematical games and puzzles for our examples illustrating the expressive power of Maude.
**Keywords**: Rule-based programming, Maude, puzzles.

## 1 Presentation

Though not a formal branch of Mathematics, mathematical games and puzzles of all sorts constitute an important subclass in the realm of mathematical problems, with a long tradition and extensive literature [1,4,11,12]. Most of them have in common the fact that they are easy to state and understand, which does not mean that a precise solution is always trivial to find.

Here we make use of a collection of these problems to introduce Maude, a specification language that efficiently implements rewriting logic [10], which includes equational logic as a sublogic. We are not concerned with finding neat and concise mathematical solutions, but rather we would like to find out how easy is to express those problems in the rewriting logic formalism underlying Maude, and how far we can go in their resolution by the use of just brute force and as less ingenuity as possible. In this regard, a clear conclusion is that many of these problems can be represented/specified in Maude in a much simpler way than it would be possible in other more conventional languages. Among the main reasons why the rule-based programming paradigm supported by Maude allows so natural a representation of many problems, we would like to mention:

- The syntax is user-definable to a great extent, which allows to choose the more appropriate one for each problem. In particular, operators declared by the user can have attributes like associativity and commutativity, which makes *multiset* rewriting trivial. All the specifications in this paper make essential use of this feature.
- An expressive version of *equational logic* allows a (first-order) version of functional programming to describe the static aspects of a system.
- The dynamic aspects are described by means of rules that represent the possible transitions or *changes* in a system. Those rules need only specify the part of the system that actually changes, which makes them quite simple. This corresponds to the fact that *rewriting logic* is a logic very suitable for expressing concurrent action and change [7] in which the frame problem [6] has been avoided.
- The transitive closure of the relation defined by the rules is automatically computed by the Maude system. This, combined with the flexible `search` command, lets the user explore all computations starting at a given state.

On the other hand, it is also true that some of these examples suffer from the state explosion problem which makes it difficult to solve them just by checking all possible combinations.

Most of the problems introduced here are well-known and can be found (in some form or another) in a number of sources: see [11] for a classic reference on the subject, [4] for a delightful exposition on how to tackle these problems, [1] for an on-line presentation, or even [12] for more algebraic ones. In many cases, a clear mathematical solution exists, but not always, and anyway our goal is to show the ease with which Maude lends itself to the specification of these problems, and to try to solve them without much thinking.

This paper is thus an introduction to rule-based programming in Maude by means of a collection of puzzles showing the language's expressive power. Even though current Maude users no doubt will find the examples here to be very simple, those new to it may still find them attractive and be encouraged to use Maude for more "serious" applications. Anyway, even that would be too ambitious a goal: the main reason why this was written down was, plain and simple, to have some fun. And we hope that you will have some fun while reading it, too. All the examples can be downloaded from `http://maude.sip.ucm.es/games`.

## 2 A Brief Overview on Rewriting Logic and Maude

The motivation for this section is not to provide a crash course on rewriting logic and Maude. On the contrary, we only intend to give the flavor of the underlying theory and provide enough information so that the specifications of the examples in the next sections can be understood. For a thorough treatment we refer the interested reader to the paper in which rewriting logic was first presented [10], to the Maude manual [3], and to [9], where many more papers on rewriting logic are referenced.

Rewriting logic was proposed by Meseguer as a unified model for concurrency in the early nineties. Since then, it has proved its value as a logic *of* change as well as a logical and semantic framework [8]. As a consequence of that success some implementations were developed; the one we use is called Maude and can be obtained at `http://maude.cs.uiuc.edu` free of charge.

The states (or configurations) of a system, its static part, are specified in rewriting logic by means of an equational theory. The transitions, the dynamic part, are specified by means of rules that rewrite some terms (representing parts of a system) into others.

To illustrate both these ideas and Maude syntax consider the following example. We have some natural numbers written on a blackboard and we are allowed, at any given time, to replace any two of them by their arithmetic mean. In this case the static part corresponds to the representation of the blackboard and the numbers themselves. To represent the numbers we have first to declare their sort (or type) and then write the well-known Peano constructors.

```
sort Nat .
op 0 : -> Nat .
op s : Nat -> Nat .
```

Since we will also need to add numbers we declare an operator

```
op _+_ : Nat Nat -> Nat .
```

Note the use of Maude's mixfix syntax, with _ indicating where the arguments are to be written. Its behavior is defined inductively by means of the following two *equations*.

```
vars N M : Nat .
eq N + 0 = N .
eq N + s(M) = s(N + M) .
```

Division `_div_` would be defined analogously. As for the blackboard, it can be represented as a (nonempty) *multiset*, or bag, of numbers.

```
  sort Blackboard .
  subsort Nat < Blackboard .
  op __ : Blackboard Blackboard -> Blackboard [assoc comm] .
```

The `subsort` declaration tells Maude that a single number constitutes a valid representation for the blackboard. Multiset union is represented with empty syntax `__`. Note that this operator has two *attributes*, `assoc` and `comm`, so that terms of sort `Blackboard` are considered *modulo* associativity and commutativity (e.g., `s(0) 0` and `0 s(0)` become indistinguishable).

Finally, the system's dynamics is specified by the single *rule*

```
  rl [replace] : N M => (N + M) div s(s(0)) .
```

The word in brackets after the keyword `rl` is the rule's name and is optional. Note that it is enough to specify the behavior of the two numbers that are going to be erased, without considering the rest of the numbers in the blackboard.

The `rewrite` command can be used to *execute* the system, by means of an interpreter which applies the rules (using a default internal strategy) and stops when no rule can be applied.

```
Maude> rewrite s(s(s(s(s(s(0)))))) s(s(s(0))) s(s(0)) .
result NzNat: s(s(s(s(0))))
```

But the numbers chosen to be replaced by their mean can be selected arbitrarily, that is, in a nondeterministic way, and this affects the final result. The `search` command can be used to explore the computation tree. It receives the term to be rewritten, the relation used to obtain final states (`=>*` for zero or more rewrites), and the final state (a new variable `N:Nat` of sort `Nat` in this case). The computation tree is traversed in a *breadth-first* way.

```
Maude> search s(s(s(s(s(s(0)))))) s(s(s(0))) s(s(0)) =>* N:Nat .

Solution 1 (state 4)
N --> s(s(s(s(0))))
Solution 2 (state 5)
N --> s(s(s(0)))
No more solutions.
```

## 3 The Hopping Rabbits

Two teams of $n$ rabbits each, wearing T-shirts marked with a cross and a circle respectively, are placed facing each other on a row with $2n + 1$ positions. The `x`-team occupies the first $n$ positions and the `o`-team the last $n$; the middle one is left empty. The goal is to swap the positions of the teams (the players of each team are indistinguishable), with the rabbits moving according to the rules of the game:

1. Rabbits from the `x`-team can only move rightward, and rabbits from the `o`-team can only move leftward.
2. A rabbit is allowed to advance one position if that position is empty.
3. A rabbit can jump over a rival if the position behind it is free.

This puzzle is also known as the *toads and frogs puzzle* or *traffic jam*. It is possible to generalize the puzzle so that the number of elements in each team is different [11].

We represent the state of the game as a nonempty *list* of rabbits, specified by means of an *associative* append operator written with empty syntax `__`; note that associativity is built into the list constructor `__` using the attribute `assoc`. Each rabbit is represented as a constant `x` or `o`, according to its team, and the constant `free` represents the empty position.

The initial state of the game depends on the number $n$ of rabbits in each team. This is specified by means of an operator `initial` that builds the appropriate initial state, as indicated in the equations below to define this operator. Notice how equations are used to define the initial state, while rules are used to represent the transitions corresponding to the legal moves in the game. As pointed out in the introduction, we use two logics, each for a different purpose: equational logic for the static aspects of a system, and rewriting logic for the dynamic aspects.

Since the rules need only specify the parts of the system that change, in this game we only need to consider the positions adjacent to the free position. Thus there are four possible legal moves, and each one is represented by a rule whose label identifies the corresponding move.

The complete specification is then as follows; the second line imports the predefined module `NAT` that specifies the natural numbers with the usual notation and arithmetic operations [3, Section 7.2].

```
mod RABBIT-HOP is
  protecting NAT .
  sorts Rabbit RabbitList .
  subsort Rabbit < RabbitList .

  ops x o free : -> Rabbit .
  op __ : RabbitList RabbitList -> RabbitList [assoc] .
  op initial : Nat -> RabbitList .

  var N : Nat .
  vars L R : RabbitList .
  var B : Rabbit .

  eq initial(0) = free .
  eq initial(s(N)) = x initial(N) o .

  rl [xAdvances] : x free => free x .
  rl [xJumps] : x o free => free o x .
  rl [oAdvances] : free o => o free .
  rl [oJumps] : free x o => o x free .
endm
```

Since we are interested in knowing how to reach the final position, and in general there are several possible rules that can be applied in a given state, we use the `search` command. The example below is with $n = 3$.

```
Maude> search initial(3) =>* o o o free x x x .

Solution 1 (state 71)
empty substitution

No more solutions.
```

The sequence of 15 steps leading to the final position can be obtained as follows, where we only show the beginning of the output.

```
Maude> show path 71 .
state 0, RabbitList: x x x free o o o
===[ rl x free => free x [label xAdvances] . ]===>
state 1, RabbitList: x x free x o o o
===[ rl free x o => o x free [label oJumps] . ]===>
state 4, RabbitList: x x o x free o o
===[ rl free o => o free [label oAdvances] . ]===>
state 9, RabbitList: x x o x o free o
...
```

## 4 The Josephus Problem

As related in [11], Flavius Josephus was a famous Jewish historian who, during the Jewish-Roman war in the first century, was trapped in a cave with a group of 40 Jewish soldiers surrounded by Romans. Legend has it that, preferring death to being captured, the Jews decided to gather in a circle and rotate a dagger around it so that every third remaining person would commit suicide. Apparently, Josephus was too keen to live and quickly found out the safe position.

The problem of finding that safe position can be modeled very easily in Maude. The circle representation becomes a (circular) *list* once the beginning position is chosen. The operator `__` is used to build nonempty lists of (nonzero) natural numbers (sort `NzNat` in Maude's predefined module `NAT`) representing the original positions of the soldiers in the circle; its associativity is specified with the attribute `assoc`. Though it is not explicitly represented, we assume that the dagger is initially at position 1.

The idea then consists in continually taking the first two elements in the list and moving them to the end of it while "killing" the third one; when only two are left, the one who initially has the dagger has to commit suicide. Note that in this way the dagger remains always implicitly located at the beginning of the list. Since we need to keep track of both the actual start and end of the list, we enclose it using the operator `{_}`. In this way, rewriting takes place only at the top of the term that represents the state.

As in the previous example, the operator `initial` and the corresponding equations are used to build the initial state. Then the rules correspond to the system transitions; we have got three rules for the cases when there are two, three, or more soldiers in the circle. Notice that there is no rule corresponding to a single soldier list, because this is the situation in which the last remaining soldier decides not to follow the rules of the game. In Maude modules, several rules can have the same label, and comments are introduced with `---`.

```
mod JOSEPHUS is
  protecting NAT .
  sorts Moriturem Circle .
  subsort NzNat < Moriturem .

  op __ : Moriturem Moriturem -> Moriturem [assoc] .
  op {_} : Moriturem  -> Circle .
  op initial : NzNat -> Moriturem .

  var M : Moriturem .
  vars I1 I2 I3 N : NzNat .
  eq initial(1) = 1 .
  eq initial(s(N)) = initial(N) s(N) .

  rl [kill] : { I1 I2 I3 M } => { M I1 I2 } .
  rl [kill] : { I1 I2 I3 } => { I1 I2 } . --- This rule is necessary
                                          --- because M cannot be empty
  rl [kill] : { I1 I2 } => { I2 } .
endm
```

Had we been in the same position as Josephus (and had we had a laptop to run Maude on it), we could have found out the safe spot by executing the command:

```
Maude> rewrite { initial(41) } .
result Circle: {31}
```

Note that at any moment until the end only one of the three rules can be applied, thus the final state is reached deterministically.

It is also easy to modify the program so that every $i$-th person commits suicide, where $i$ is a parameter. The idea is the same, but because of the parameter now it is necessary to explicitly represent the dagger. For that, we use the constructor `dagger : NzNat NzNat -> Moriturem`, whose second argument stores the value of $i$ while the first one acts as a counter: each time an element is moved from the beginning of the list to the end, the first argument is decreased by one; once it reaches 1, the element that is currently the head of the list is "killed" (removed from the list).

```
mod JOSEPHUS-GENERALIZED is
  protecting NAT .
  sorts Moriturem Circle .
  subsort NzNat < Moriturem .

  op dagger : NzNat NzNat -> Moriturem .
  op __ : Moriturem Moriturem -> Moriturem [assoc] .
  op {_} : Moriturem  -> Circle .
  op initial : NzNat NzNat -> Moriturem .

  var M : Moriturem .
  vars I I1 I2 N : NzNat .

  eq initial(1, I)  = dagger(I, I) 1 .
  eq initial(s(N), I) = initial(N, I) s(N) .

  rl [kill] : { dagger(1, I) I1 M } => { dagger(I, I) M } .
  rl [kill] : { dagger(s(N), I) I1 M } => { dagger(N, I) M I1 } .
  rl [kill] : { dagger(N, I) I1 } => { I1 } .  --- The last one throws the dagger away!
endm

Maude> rewrite { initial(41, 3) } .
result Circle: {31}
```

## 5 The Three Basins Puzzle

The following is a classic puzzle with a recent cameo in the 1995 Hollywood hit *Die Hard: With a Vengeance*. In the movie, McClane and Zeus have to deactivate a bomb by placing 4 gallons of water on a balance. The supply of water is unlimited, but they only have three basins with capacities of 3, 5, and 8 gallons, respectively.

The problem can be specified in Maude as follows. A basin is represented by means of the constructor `basin` with two natural numbers as arguments: the first one is the basin capacity and the second one is how much it is filled. We can think of a basin as an object with two attributes. This way of thinking leads to an *object-based* style of programming, where objects change their attributes as result of interacting with other objects; these interactions are represented as rules on *configurations* that are nonempty *multisets* of objects [3, Chapter 8]. The multiset constructor is written with empty syntax and declared with attributes `assoc` and `comm`. The constant `initial` defines the initial configuration.

At any given time we can either empty one of the basins, or fill it completely; the rules `empty` and `fill` below take care of this. When there is enough space in one of the basins to hold the current content of another, we can transfer all the water from this second one by using rule `transfer1`. Note that this is a *conditional rule* (introduced with keyword `crl`), with the condition at the end, after keyword `if`. The case when, after pouring one basin over another, there is still some water left is dealt with by the conditional rule `transfer2` (where the operator `sd` denotes the subtraction operation over natural numbers). These last two rules could be combined into a single one, but the result would not be so clear.

```
mod DIE-HARD is
  protecting NAT .
  sorts Basin BasinSet .
  subsort Basin < BasinSet .

  op basin : Nat Nat -> Basin .     --- Capacity / Content
  op __ : BasinSet BasinSet -> BasinSet [assoc comm] .
  op initial : -> BasinSet .

  vars M1 N1 M2 N2 : Nat .

  eq initial = basin(3, 0) basin(5, 0) basin(8,0) .

  rl [empty] : basin(M1, N1) => basin(M1, 0) .
  rl [fill] : basin(M1, N1) => basin(M1, M1) .
  crl [transfer1] : basin(M1, N1) basin(M2, N2) =>
                    basin(M1, 0) basin(M2, N1 + N2) if N1 + N2 <= M2 .
  crl [transfer2] : basin(M1, N1) basin(M2, N2) =>
                    basin(M1, sd(N1 + N2, M2)) basin(M2, M2) if N1 + N2 > M2 .
endm
```

We can now find out the shortest solution with the help of the `search` command, due to the breadth-first way of searching (the argument `[1]` tells Maude to look only for one solution). Notice that the *pattern* used after the arrow `=>*` represents any set of basins with one of them having 4 gallons.

```
Maude> search [1] initial =>* basin(N:Nat, 4) B:BasinSet .

Solution 1 (state 75)
B:BasinSet --> basin(3, 3) basin(8, 3)
N:Nat --> 5
```

The sequence of actions that leads to the solution can be seen with `show path 75`, where we omit part of the information about the rules used.

```
Maude> show path 75 .
state 0, BasinPack: basin(3, 0) basin(5, 0) basin(8, 0)
```

```
===[ rl ... fill ]===>
state 2, BasinPack: basin(3, 0) basin(5, 5) basin(8, 0)
===[ crl ... transfer2 ]===>
state 9, BasinPack: basin(3, 3) basin(5, 2) basin(8, 0)
===[ crl ... transfer1 ]===>
state 20, BasinPack: basin(3, 0) basin(5, 2) basin(8, 3)
===[ crl ... transfer1 ]===>
state 37, BasinPack: basin(3, 2) basin(5, 0) basin(8, 3)
===[ rl ... fill ]===>
state 55, BasinPack: basin(3, 2) basin(5, 5) basin(8, 3)
===[ crl ... transfer2 ]===>
state 75, BasinPack: basin(3, 3) basin(5, 4) basin(8, 3)
```

## 6 Crossing the Bridge

The four components of U2, the famous band of rock music, are in a tight situation. The concert starts in 17 minutes and in order to get to the stage they must first cross an old bridge through which only a maximum of two persons can walk over at the same time. It is already dark and, because of the bad condition of the bridge, to avoid falling into the darkness it is necessary to cross it with the help of a flashlight. Unfortunately, they only have one. Knowing that Bono, Edge, Adam, and Larry take 1, 2, 5, and 10 minutes, respectively, to cross the bridge, is there a way that they can make it to the concert on time?

The current state of the group can be represented in Maude by a *multiset* consisting of singers, the flashlight, and a watch to keep record of the time. The flashlight and the singers have a `Place` associated to them, indicating whether their current position is to the left or to the right of the bridge; each singer, in addition, also carries the time it takes him to cross the bridge. As in the previous example, this specification follows an object-based style of programming. We have an auxiliary operation `changePos` that is defined by means of two equations.

The traversing of the bridge is modeled by two rewrite rules: the first one for the case in which a single person crosses it, and the second one for when there are two. Note that for somebody to be allowed to cross, their position relative to the bridge must be the same as for the flashlight, which is represented by having the same variable `P` twice on the lefthand side of the rules. Also, since `__` is commutative, the condition in the second rule amounts to no loss of generality.

```
mod U2 is
  protecting NAT .
  sorts Singer Object Group Place .
  subsorts Singer Object < Group .

  ops left right : -> Place .
  op changePos : Place -> Place .
  op flashlight : Place -> Object .
  op watch : Nat -> Object .
  op singer : Nat Place -> Singer .
  op __ : Group Group -> Group [assoc comm] .
  op initial : -> Group .

  var P : Place .
  vars M N N1 N2 : Nat .
  eq initial = watch(0) flashlight(left)
               singer(1, left) singer(2, left) singer(5, left) singer(10, left) .

  eq changePos(left) = right .
  eq changePos(right) = left .

  rl [one-crosses] : watch(M) flashlight(P) singer(N, P) =>
                     watch(M + N) flashlight(changePos(P)) singer(N, changePos(P)) .
  crl [two-cross] : watch(M) flashlight(P) singer(N1, P) singer(N2, P) =>
                    watch(M + N1) flashlight(changePos(P)) singer(N1, changePos(P))
                    singer(N2, changePos(P))
                  if N1 > N2 .
endm
```

A solution can now be found quickly by looking for a state in which all singers (and the flashlight) are to the `right` of the bridge. Notice how the `search` command is invoked with a `such that` clause that allows to introduce a condition that solutions have to fulfill.

```
Maude> search [1] initial =>* flashlight(right) watch(N:Nat) singer(1, right)
                              singer(2, right) singer(5, right) singer(10, right)
       such that N:Nat <= 17 .

Solution 1 (state 402)
N --> 17
```

The solution takes exactly 17 minutes (a happy ending after all!) and the complete trace can be shown as follows:

```
Maude> show path 402 .
state 0, Group: flashlight(left) watch(0) singer(1, left) singer(2, left)
                singer(5, left) singer(10, left)
===[ crl ... two-cross ]===>
state 5, Group: flashlight(right) watch(2) singer(1, right) singer(2, right)
                singer(5, left) singer(10, left)
===[ rl ... one-crosses ]===>
state 15, Group: flashlight(left) watch(3) singer(1, left) singer(2, right)
                 singer(5, left) singer(10, left)
===[ crl ... two-cross ]===>
state 71, Group: flashlight(right) watch(13) singer(1, left) singer(2, right)
                 singer(5, right) singer(10, right)
===[ rl ... one-crosses ]===>
state 158, Group: flashlight(left) watch(15) singer(1, left) singer(2, left)
                  singer(5, right) singer(10, right)
===[ crl ... two-cross ]===>
state 402, Group: flashlight(right) watch(17) singer(1, right) singer(2, right)
                  singer(5, right) singer(10, right)
```

After sorting out the information, it becomes clear that Bono and Edge have to be the first to cross. Then Bono returns with the flashlight, which gives to Adam and Larry. Finally, Edge takes the flashlight back to Bono and they cross the bridge together for the last time.

Note that, in order for the `search` command to stop, we need to tell Maude to look only for one solution. Otherwise, it will continue exploring all possible combinations, increasingly taking a larger amount of time, and it will never end.

## 7 The Looping Chips

In the next game, taken from [1], four chips of different colors have been placed in consecutive places on a $12 \times 1$ board whose ends have been glued together. Each chip can be moved 5 places from its current location, either clockwise or counterclockwise, assuming the final position is empty. The goal is to arrange the chips in reverse order, over the original four squares.

The state is again represented by a *multiset* of `Places`, with each `Place` determined by its position in the board and the color of the chip on it or `e` if empty[1]. As in previous examples, places can be understood as objects and the state of the game at each moment is given by a configuration of objects. The constants `initial` and `final` represent the initial and final configurations.

There are two possible legal moves in the game, but taking advantage of the circularity of the board, it is possible to represent both together in one rule, as shown below; notice how the condition of the rule considers the two possible directions of the move.

```
mod CHIPS is
  protecting NAT .
  sorts Place Board Chip .
  subsort Place < Board .

  ops r b g y e : -> Chip .
  op place : Nat Chip -> Place .
  op __ : Board Board -> Board [assoc comm] .
  ops initial final : -> Board .

  eq initial = place(0,r) place(1,b) place(2,g) place(3,y)
               place(4,e) place(5,e) place(6,e) place(7,e)
               place(8,e) place(9,e) place(10,e) place(11,e) .

  eq final = place(0,y) place(1,g) place(2,b) place(3,r)
             place(4,e) place(5,e) place(6,e) place(7,e)
             place(8,e) place(9,e) place(10,e) place(11,e) .

  vars I J : Nat .
  var C : Chip .

  crl [move] : place(I,C) place(J,e) => place(I,e) place(J,C)
           if ((I + 5) rem 12 == J) or ((J + 5) rem 12 == I).
endm
```

Then, we can use the command

```
Maude> search initial =>* B:Board such that B:Board == final .

No solution.
```

to prove that it is not possible, by using the allowed moves, to reverse the original order of the chips. Notice that the constant `final` is not a pattern (because it can be reduced), and therefore cannot be used after the arrow in the `search` command; the clause at the end allows us to say the same.

[1] In this example, we could also use a list representation for the state; this would simplify the representation of places, but instead the corresponding rules would be more complex.

## 8 The Khun Phan Puzzle

The Khun Phan puzzle is one of those typical puzzles consisting of a rectangular board over which some pieces can be slid. The goal is to move the pieces so as to reach a certain configuration in which sometimes a picture becomes clear or other times a piece understood as some character is freed from his guards. Figure 1 shows the initial configuration that we will consider. The board is a $4 \times 5$ rectangle, there is one $2 \times 2$ piece, five rectangular pieces of size $2 \times 1$, and four smaller squares with dimension $1 \times 1$; there are only two empty spaces that must be used to slide the pieces. The goal we consider is to move the pieces so as to put the big square in the position where the small ones are initially. An additional twist would be to reach a completely symmetric position with respect to the original.

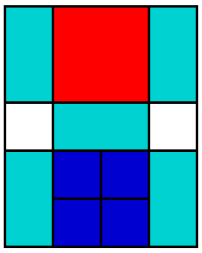

**Fig. 1.** Khun Phan puzzle

The state of the board is also represented as a *multiset* of pieces with the operator `__`. There is a different constructor for each piece, `bigsq`, `hrect`, `vrect`, and `smallsq`, and another one, `empty`, to indicate an empty space (that is considered to be just a special kind of piece). These constructors take two natural numbers as arguments that correspond to the coordinates of the upper left corner of the piece; the origin $(1,1)$ is located at the upper left corner of the board.

The representation of the moves as rewrite rules is then immediate: each involves a piece and at least one empty space. For each kind of piece there are four rules, corresponding to the four possible directions. For example, moving the big square one position to the right is captured by the rule `Sqr` below. Again we think of pieces as objects and rules as interactions among them. The complete specification is as follows.

```
mod KHUN-PHAN is
  protecting NAT .
  sorts Piece Board .
  subsort Piece < Board .

  op __ : Board Board -> Board [assoc comm] .
  ops empty bigsq smallsq hrect vrect : Nat Nat -> Piece .
  op init : -> Board .

  vars X Y : Nat .

  eq initial = vrect(1,1)       bigsq(2,1)          vrect(4,1)
               empty(1,3)       hrect(2,3)          empty(4,3)
               vrect(1,4) smallsq(2,4) smallsq(3,4) vrect(4,4)
                          smallsq(2,5) smallsq(3,5) .
  rl [sqr] : smallsq(X,Y) empty(s(X),Y) => empty(X,Y) smallsq(s(X),Y) .
  rl [sql] : smallsq(s(X),Y) empty(X,Y) => empty(s(X),Y) smallsq(X,Y) .
  rl [squ] : smallsq(X,s(Y)) empty(X,Y) => empty(X,s(Y)) smallsq(X,Y) .
  rl [sqd] : smallsq(X,Y) empty(X,s(Y)) => empty(X,Y) smallsq(X,s(Y)) .

  rl [Sqr] : bigsq(X,Y) empty(s(s(X)),Y) empty(s(s(X)),s(Y)) =>
             empty(X,Y) empty(X,s(Y)) bigsq(s(X),Y) .
  rl [Sql] : bigsq(s(X),Y) empty(X,Y) empty(X,s(Y)) =>
             empty(s(s(X)),Y) empty(s(s(X)),s(Y)) bigsq(X,Y) .
  rl [Squ] : bigsq(X,s(Y)) empty(X,Y) empty(s(X),Y) =>
             empty(X,s(s(Y))) empty(s(X),s(s(Y))) bigsq(X,Y) .
  rl [Sqd] : bigsq(X,Y) empty(X,s(s(Y))) empty(s(X),s(s(Y))) =>
             empty(X,Y) empty(s(X),Y) bigsq(X,s(Y)) .

  rl [hrectr] : hrect(X,Y) empty(s(s(X)),Y) => empty(X,Y) hrect(s(X),Y) .
  rl [hrectl] : hrect(s(X),Y) empty(X,Y) => empty(s(s(X)),Y) hrect(X,Y) .
  rl [hrectu] : hrect(X,s(Y)) empty(X,Y) empty(s(X),Y) =>
                empty(X,s(Y)) empty(s(X),s(Y)) hrect(X,Y) .
  rl [hrectd] : hrect(X,Y) empty(X,s(Y)) empty(s(X),s(Y)) =>
                empty(X,Y) empty(s(X),Y) hrect(X,s(Y)) .

  rl [vrectr] : vrect(X,Y) empty(s(X),Y) empty(s(X),s(Y)) =>
                empty(X,Y) empty(X,s(Y)) vrect(s(X),Y) .
  rl [vrectl] : vrect(s(X),Y) empty(X,Y) empty(X,s(Y)) =>
                empty(s(X),Y) empty(s(X),s(Y)) vrect(X,Y) .
  rl [vrectu] : vrect(X,s(Y)) empty(X,Y) => empty(X,s(s(Y))) vrect(X,Y) .
  rl [vrectd] : vrect(X,Y) empty(X,s(s(Y))) => empty(X,Y) vrect(X,s(Y)) .
endm
```

Then we can use the command

```
Maude> search initial =>* B:Board bigsq(2,4) .
```

to get all possible 964 solutions to the game. The final state used, `B:Board bigsq(2,4)`, represents any final situation such that the upper left corner of the big square is at coordinates $(2,4)$. No wonder it takes some time to find a solution: close examination of the first one, corresponding to the shortest path leading to the final configuration due to the breadth-first search, reveals that it consists of 112 moves!

Similarly, the command

```
Maude> search initial =>* vrect(1,1) smallsq(2,1) smallsq(3,1) vrect(4,1)
                                     smallsq(2,2) smallsq(3,2)
                          empty(1,3)        hrect(2,3)        empty(4,3)
                          vrect(1,4)        bigsq(2,4)        vrect(4,4) .

No solution.
```

shows that it is not possible to reach a position symmetric to the initial one.

## 9 Crossing the River

A shepherd needs to transport to the other side of a river a wolf, a goat, and a cabbage. He has only a boat with room for the shepherd himself and another item. The problem is that in the absence of the shepherd the wolf would eat the goat, and the goat would eat the cabbage.

We represent with constants `left` and `right` the two sides of the river. The shepherd and his belongings are represented as objects with an attribute indicating the side of the river in which each is located; the constant `initial` denotes the initial situation in which we assume that all the objects are located in the left riverbank. The rules represent the ways of crossing the river that are allowed by the capacity of the boat; an auxiliary operation `change` is used to modify the corresponding attributes.

The interesting decision we have made in our specification is to use equations to represent the facts that the wolf eats the goat when they are alone, or that the goat eats the cabbage. Note that the statement of the problem is underspecified; it is not clear what exactly should happen were the wolf, the goat, and the cabbage left alone. In the specification below we have decided that the goat is not fast enough and gets eaten by the wolf before it can take a bite of the cabbage. Note that we use conditional equations, introduced with keyword `ceq`.

```
mod RIVER-CROSSING is
  sorts Side Group .

  ops left right : -> Side .
  op change : Side -> Side .
  ops s w g c : Side -> Group .
  op __ : Group Group -> Group [assoc comm] .
  op initial : -> Group .

  vars S S' : Side .

  eq change(left) = right .
  eq change(right) = left .

  ceq w(S) g(S) s(S') = w(S) s(S') if S =/= S' .
  ceq c(S) g(S) w(S') s(S') = g(S) w(S') s(S') if S =/= S' .

  eq initial = s(left) w(left) g(left) c(left) .

  rl [shepherd-alone] : s(S) => s(change(S)) .
  rl [wolf] : s(S) w(S) => s(change(S)) w(change(S)) .
  rl [goat] : s(S) g(S) => s(change(S)) g(change(S)) .
  rl [cabbage] : s(S) c(S) => s(change(S)) c(change(S)) .
endm
```

By using the command `search` we can confirm that there is only one way the shepherd can safely take his belongings to the other side.

```
Maude> search initial =>* w(right) s(right) g(right) c(right) .
```

One might think about using rules instead of equations to represent the "eating transitions," but this would not be correct because it would allow paths in which the shepherd leaves for example the goat and the cabbage alone and later comes back to find that the cabbage is still there.

## 10 Dominoes on the Chessboard

We are given an $8 \times 8$ board and 31 dominoes, each of which can be used to cover exactly two squares of the board. Is it possible to arrange the dominoes on the board so as to leave uncovered the upper left and the lower right corners?

The answer is no, and a neat solution is given in [4, Chapter 1] among other places. It is enough to imagine the board painted like a chessboard and realize that each domino necessarily covers both a black and a white square: since the corners to be left uncovered are of the same color, such a covering is not possible. This solution, however, requires some ingenuity and, given our present lazy approach, that is not a desirable characteristic. Therefore, we are going to model the problem in Maude and try to solve it by sheer force.

Again, the state of the board is represented as a *multiset* of squares. Each square has three arguments: the first two are its coordinates (column/row) and the last one indicates whether it is already covered or still empty. Since the position of the squares in the board is fixed, the attribute `comm` for `__` could be thought to be unnecessary. This, however, allows a more homogeneous and simple presentation of the rules taking care of positioning the dominoes *both* horizontally and vertically, by focusing only on those two squares involved in placing the domino. Having the board represented as a list by removing the attribute `comm` would force us to represent all the squares in between them in one of the rules.

```
mod CHESS-COVER is
  protecting NAT .
  sorts Status Pos Board State .
  subsort Pos < Board .

  ops e c : -> Status .
  op sq : Nat Nat Status -> Pos .
  op __ : Board Board -> Board [assoc comm] .
  ops initial final : -> Board .

  vars I J I1 J1 : Nat .
  var B : Board .

  eq initial = sq(1,1,e) sq(2,1,e) sq(3,1,e) sq(4,1,e) sq(5,1,e) ...
               ... sq(4,8,e) sq(5,8,e) sq(6,8,e) sq(7,8,e) sq(8,8,e) .

  eq final = sq(1,1,e) sq(2,1,c) sq(3,1,c) sq(4,1,c) sq(5,1,c) ...
             ... sq(4,8,c) sq(5,8,c) sq(6,8,c) sq(7,8,c) sq(8,8,e) .

  rl [hor] : sq(I,J,e) sq(s(I),J,e) => sq(I,J,c) sq(s(I),J,c) .
  rl [ver] : sq(I,J,e) sq(I,s(J),e) => sq(I,J,c) sq(I,s(J),c) .
endm
```

Now, the command

```
Maude> search initial =>* B:Board such that B:Board == final .
```

should return the answer. This time, however, a state explosion problem occurs and in our computer the program runs out of memory before producing any result. To solve it, we are forced to use some ingenuity after all. Note that instead of placing the dominoes in an arbitrary order we could do it starting either from the top of the board towards the bottom, or from the left towards the right, or even in a diagonal manner beginning at the upper left corner. The first two approaches still do not return an answer, but the third does. To implement it, we need an auxiliary operator `cDiag` that checks whether all positions in the board that come before a given square according to the diagonal order have been already covered. Also, as we did in Section 4, we need to have full control of all the elements in the board and for that we enclose it inside the constructor `{_}`.

```
  ceq cDiag(I,J,sq(I1,J1,e) B) = false if (I1 + J1 < I + J) /\ (I1 + J1 >= 3) .
  eq cDiag(I,J,B) = true [owise] .

  crl [hor] : { B sq(I,J,e) sq(s(I),J,e) } => { B sq(I,J,c) sq(s(I),J,c) }
              if cDiag(I,J,B) .
  crl [ver] : { B sq(I,J,e) sq(I,s(J),e) } => { B sq(I,J,c) sq(I,s(J),c) }
              if cDiag(I,J,B) .
```

The “otherwise” attribute, `owise`, is just a convenient way of specifying the behavior of `cDiag` in all remaining cases without having to write equations for them. The result is still an equational theory since the `owise` attribute is just a shorthand for a conditional equation [3, Section 4.5.4].

Finally, the result of the command

```
Maude> search { initial } =>* { B:Board } such that B:Board == final .

No solution.
```

proves that such a covering is not possible.

## 11 By Way of Conclusion

We have specified several other games and puzzles, but we think that by now the pattern by which these problems are modeled and solved in Maude should be clear. There are however several advanced features available in Maude that can be useful in some examples, and that we have not considered here with the idea of keeping an introductory level.

The first one is the possibility of using membership axioms [3, Chapter 4] to refine the representation of the state. For example, the multiset constructor allows repetition of elements, but this should be forbidden in some situations; as another example, in the Khun Phan puzzle a piece cannot be stacked on top of another. In the puzzles above we have not made use of this, but memberships are the right tool to make sure all the elements are different.

Another important feature is the availability of a model checker for linear temporal logic [3, Chapter 9]. For example, we have modeled the river crossing puzzle without the “eating equations” by using the model checker instead of the `search` command to represent a safe path by means of a temporal formula.

To gain some perspective we have also carried out a small comparison with other rule-based programming languages, namely, ELAN [2], CHR [5], and ASF+SDF [13]. We acknowledge beforehand that none of us is an expert in any of them, so our conclusions should be taken with a grain of salt. ELAN, which is also based on rewriting logic, is the most similar to Maude, offering a clear distinction between the statics and the dynamics of a system, by means of two different kinds of rules. Like Maude, ELAN also supports rewriting modulo associativity and commutativity, though not modulo associativity only, so the examples involving lists are more cumbersome to describe. The examples involving multiset rewriting can be easily specified; however, since in most cases the rules are nonterminating and the predefined search strategy in ELAN is depth-first, it cannot be used to find a solution (a breadth-first strategy could be specified but by no means in a straightforward manner). Regarding efficiency, for the only problem we were able to translate almost verbatim from Maude, the Josephus problem in Section 4, the performance of the Maude interpreter was much better than even the compiled ELAN version (in some of our experiments, Maude finished within seconds while ELAN took several hours). On the other hand, after trying to specify some of the games in CHR and ASF+SDF, as far as we know there is no distinction between equations and rules and no support for equational attributes, both of which have been essential in our examples. All this suggests to us that the representation of transition systems in these two formalisms is not as natural and straightforward as in Maude. In addition, breadth-first search in CHR seems to present the same problems as in ELAN.

As we mentioned in the introduction, the main goal of this paper was to have some fun while introducing rule-based programming in Maude; we hope that our games have whetted the appetite of the reader for more Maude applications.

## References

1. A. Bogomolny. Interactive mathematics miscellany and puzzles. `http://www.cut-the-knot.org/games.shtml`.
2. P. Borovanský, C. Kirchner, H. Kirchner and P.-E. Moreau. ELAN from the rewriting logic point of view *Theoretical Computer Science*, 285(2):155–185, 2002.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude manual (version 2.1). `http://maude.cs.uiuc.edu/manual/`, 2004.
4. D. Fomin, S. Genkin, and I. Itenberg. *Mathematical Circles: The Russian Experience*, volume 7 of *Mathematical World*. American Mathematical Association, 1996.
5. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming.* Springer, 2003.
6. P. J. Hayes. What the frame problem is and isn't. In Z. W. Pylyshyn, editor, *The Robot's Dilemma: The Frame Problem in Artificial Intelligence*, pages 123–137. Ablex Publishing Corp., 1987.
7. N. Martí-Oliet and J. Meseguer. Action and change in rewriting logic. In R. Pareschi and B. Fronhöfer, editors, *Dynamic Worlds: From the Frame Problem to Knowledge Management*, pages 1–53. Kluwer Academic Press, 1999.
8. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Second Edition, Volume 9*, pages 1–87. Kluwer Academic Publishers, 2002.
9. N. Martí-Oliet and J. Meseguer. Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.
10. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
11. W. W. Rouse Ball and H. S. M. Coxeter. *Mathematical Recreations and Essays.* Dover, 1987.
12. D. O. Shklarsky, N. N. Chentzov, and I. M. Yaglom. *The USSR Olympiad Problem Book.* Dover, 1993.
13. A. van Deursen, J. Heering, and P. Klint. *Language Prototyping: An Algebraic Specification Approach.* AMAST Series in Computing, 5, World Scientific, 1996.