MMT case study: distributed algorithms

Fabricio Chalub Christiano Braga

July 11, 2005

Contents

1 Introduction

This technical report shows the use of the *Maude MSOS Tool* [2] in the specification and verification of distributed algorithms. It is well known that SOS and MSOS are formalisms not only used in the specification of programming languages, but also of concurrent systems [8, 9]. The conversion from MSOS to Rewriting Logic performed by *Maude MSOS Tool* using the

Maude interpreter enables the use of Maude's built in Linear Temporal Logic (LTL) model checker and breadth-first search capabilities.

This report is organized as follows: Section 1.1 defines a model for process execution of distributed processes and Section 1.2 shows the examples from [4] and [3].

1.1 Process execution model

This Section outlines a simple process execution model. We begin with the notion of *processes* and *process identifiers*. The set '`Proc`' represents processes in our specifications:

```
Proc .
```

A process contains an integer as its process identifier (pid) and an abstract data type that represents its local state ('`St`'). The local state is dependent on the algorithm being specified, and will be mostly used on our specifications to record the state of the computation of a process, but it can also store temporary values that are local to a specific process throughout the execution.

```
St .
Proc ::= prc (Int, St) .
```

We follow ideas present in [5, 1] and create a set '`Soup`' that represents an associative-commutative "soup of processes." A single process is a trivial soup. The evolution of the soup is done by selecting non-deterministically a process out of the "floating processes," made using matching modulo associativity and commutativity, evaluating this process, and putting it back into the soup.

```
Soup ::= Proc .
Soup ::= Soup Soup [assoc comm] .
```

The following rule implements the evolution of the soup of processes.

```
                      Proc -{...}-> Proc'
 [exec1] -- -----------------------------------
         (Proc Soup) : Soup -{...}-> Proc' Soup .
```

One could write the left-hand side of the conclusion as '`Soup1 Soup2`', instead of '`Proc Soup`'. This would select non-deterministically an entire portion of the soup to evolve. This extra generality is not necessary on some of the algorithms shown here, since they specify transitions for a particular process, and not a subset of processes. The alternative rule would then recursively apply to itself until '`Soup1`' is a single '`Proc`' to which there are other applicable transitions available, generating unnecessary rewrites, and artificially augmenting the state space of a particular specification.

Finally, we need a rule for the trivial case in which the soup consists of a single process:

```
        Proc -{...}-> Proc'
 -- -------------------------
  Proc : Soup -{...}-> Proc' .
```

1.1.1 Process communication models

This Section describes two possible models for process communication: shared memory and message-passing.

Shared memory model is trivially implemented with the use of a read-write component on the label to store the shared variables of the processes. The remainder of this Section deals with a simple message passing model on an asynchronous network.

The set '`Msg`' represents the *messages* that circulate on the network.

```
Msg .
```

The specific type of message is, as usual, algorithm dependent, but, for this exposition let us assume the following:

```
Msg ::= msg Int from Int to Int .
```

where the first argument is the *value to be transfered*, the second is the *origin* of the message, and the third is the *destination*.

The message passing mechanism in our specification follows Maude's pattern matching capabilities. In this mechanism, messages and processes "float" on the soup and the transition rules will emulate the transmission of a message to a process by matching the destiny argument of the message with the pid present on the process object. For this we need to expand the range of the '`Soup`' object to allow also messages.

```
Soup ::= Msg | Proc .
```

To exemplify the message passing through matching, consider the following fragment in which a message originating from process '`Int'`' with a destination of process '`Int`' is paired with the process of pid '`Int`'.

```
prc (Int, St) (msg C from Int' to Int)
```

Since now processes *and* messages need to interact for the evolution of the soup, we must generalize the interleaving rule to allow the evolution of a portion of the soup.

```
                Soup1 -{...}-> Soup'1
 [exec2] -- ----------------------------------------
        (Soup1 Soup2) : Soup -{...}-> Soup'1 Soup2 .
```

While it is true that this rule has the drawback discussed at the beginning of Section 1.1, its generality allows no *a priori* commitments on the nature of the algorithm. In order words, the relationship of objects and messages is left open for a wide variety of interactions, depending on the specific needs of a particular specification.

1.1.2 Justice

Let us discuss justice in rule '`[exec1]`'. It is easy to notice that there is no specific order in which processes are selected to be evaluated: all possible traces of execution are produced, including those in which a particular process loops forever, not letting any other process evolve.

A very simple way of adding justice to a specification is by controlling which process is chosen to be evaluated through some sort of *scheduling policy*. Let us describe one such policy, the round-robin, or *fair* scheduling of processes. It consists of having a counter that operates modulo the number of processes: the current value of the counter is the pid of the process that is allowed to execute; upon executing one step, the counter is incremented. With this strategy, all processes eventually reach their execution turn.

This is implemented by adding a read-write component indexed by '`fair`' to the label.

```
Label = { fair : Int, fair' : Int, ... } .
```

We now change rule '[exec1]' to reflect the scheduling just described. Let us assume that there is a constant 'n' that will be instantiated later, through an equation, with the number of processes in the soup.

```
Int' := (Int + 1) rem n,
prc (Int, St) -{fair = Int, fair' = Int, ...}-> prc (Int, St')
-- ------------------------------------------------------------
(prc (Int, St) Soup) : Soup -{fair = Int, fair' = Int', ...}->
                                              prc (Int, St') Soup .
```

Even though this solution works, it is far too restricted: all processes receive the same probability of execution, which is not always the case, and the processes always execute on the same order. This last restriction may be lifted by using a pseudo-random number generator and randomly selecting which process to evaluate at a time.

1.2 Examples

1.2.1 Another thread game

Let us begin with a simple specification, based on the *thread game* described in [3]. This specification also demonstrates the problems associated with the justice (or lack thereof) in the specification.

Two threads continuously attempt to update the value of a shared variable: one process increments the value by one, while the other decrements the value by one. This shared variable is modelled using a read-write component indexed by 'v'.

```
Label = {v : Int, v' : Int, ...} .
```

Let us formalize the behavior of both threads. Process 'prc 0' increments and process 'prc 1' decrements. Let us also limit the value of the shared variable to no less than zero and no more than five, an arbitrary value.

```
   Int < 5, Int' := Int + 1
-- ------------------------------------------------
(prc 0) : Proc -{v = Int, sh' = Int', -}-> prc 0 .

   Int > 0, Int' := Int - 1
-- ------------------------------------------------
(prc 1) : Proc -{v = Int, sh' = Int', -}-> prc 1 .
```

These next two rules keep the system running when the variable is in the established limits.

```
    Int >= 5, Int' := Int
 -- ------------------------------------------------
 (prc 0) : Proc -{v = Int, sh' = Int', -}-> prc 0 .

    Int <= 0, Int' := Int
 -- ------------------------------------------------
 (prc 1) : Proc -{v = Int, sh' = Int', -}-> prc 1 .
```

In order to analyze this specification with Maude's model checker, let us create a proposition '`max(i)`', which holds whenever the shared variable has a value equal or inferior to `i`.

```
 op max : Int -> Prop .

 ceq (< S, { v = I', PR } >) |= max (I) = true
 if I' <= I .
```

If we use the *fair scheduling* of processes described on Section 1.1.2, we will notice that the value of the shared variable will never exceed one.

```
rewrites: 2511 in 26ms cpu (26ms real) (93013 rewrites/second)
reduce in CHECK :
  modelCheck(init,[] max(1))
result Bool :
  true
```

Using the specification without fairness we quickly arrive at a counterexample where process zero always increments the shared variable up to five.

```
reduce in CHECK :
  modelCheck (init, [] max(1))
result ModelCheckResult :
  counterexample(
  { < (prc 0 prc 1), {fair = 0, v = 0} > }
  { < (prc 0 prc 1), {fair = 0, v = 1} > }
  { < (prc 0 prc 1), {fair = 0, v = 2} > }
  { < (prc 0 prc 1), {fair = 0, v = 3} > }
  { < (prc 0 prc 1), {fair = 0, v = 4} > },
  { < (prc 0 prc 1), {fair = 0, v = 5} > })
```

1.2.2 Mutual exclusion using semaphores

This Section specifies a mutual exclusion algorithm using semaphores. It is also an introductory example that shows how a process keeps is internal state using the '`St`' set.

Let us begin with a specification without semaphores and check the race condition problem, in this specification processes have two possible states: either they are inside the critical region ('`cric`') or not, the remainder region ('`rem`').

```
St .
St ::= crit | rem .

Proc .
Proc ::= pid (Int, St) .
```

Processes keep entering and leaving their critical region.

```
prc (Int, rem) : Proc --> prc (Int, crit) .
prc (Int, crit) : Proc --> prc (Int, rem) .
```

This is specification is simple enough and we may search for all possible states. A simple '`search`' command suffices to show all four options:

```
(search
  (< (prc (0, rem) prc (1, rem)) ::: 'Soup,
     { null } >) =>* C:Conf .)

Solution 1
C:Conf <- <(prc(0,rem) prc(1,rem))::: 'Soup,{null}>

Solution 2
C:Conf <- <(prc(0,crit) prc(1,rem))::: 'Soup,{null}>

Solution 3
C:Conf <- <(prc(0,rem) prc(1,crit))::: 'Soup,{null}>

Solution 4
C:Conf <- <(prc(0,crit) prc(1,crit))::: 'Soup,{null}>
```

To avoid the race condition shown on the fourth solution, let us rewrite our rules with a semaphore semantics: before entering the critical region, a process will go through intermediate states '`down`' and '`up`', represented by the read-write component '`sem`'.

```
Label = { sem : Int, sem' : Int, ... } .
```

We need to at '`down`' and '`up`' to our set of possible states, '`St`':

```
St ::= down | up .
```

Before entering a critical region, a process first go to its '`down`' state:

```
prc (Int, rem) : Proc --> prc (Int, down) .
```

Following the semaphore semantics, a process will only access its critical region when the semaphore is zero.

```
   Int' == 0
-- ------------------------------------------------------
prc (Int, down) : Proc -{sem = Int', sem' = Int', -}->
prc (Int, down) .

   Int' > 0, Int'' := Int' - 1
-- ------------------------------------------------------
prc (Int, down) : Proc -{sem = Int', sem' = Int'', -}->
prc (Int, crit) .
```

Moving from the critical region to the remainder, the process first executes its '`up`' action, incrementing the value of the semaphore by one.

```
prc (Int, crit) : Proc --> prc (Int, up) .

   Int'' := Int' + 1
-- ------------------------------------------------------
prc (Int, up) : Proc -{sem = Int', sem' = Int'', -}->
prc (Int, rem) .
```

Now, a search for a configuration where a race condition occurs is unsuccessful.

```
search : <(prc(0,rem)prc(1,rem))::: 'Soup,{sem = 1}> =>*
         <(prc(0,crit)prc(1,crit))::: 'Soup,R:Record > .

No solution.
```

Let us use the model checker to confirm this result. We begin by creating an auxiliar operation '`create-conf(i)`' that creates a configuration with `i` processes. The proposition '`race-condition`' holds whenever is more than one process is its critical zone.

```
rewrites: 817190 in 7638ms cpu (7638ms real)
          (106978 rewrites/second)
reduce in CHECK :
  modelCheck(create-conf(10), [] ~ race-condition)
result Bool :
  true
```

It is interesting to observe that, since the system does not have justice, there is a possibility that a process may *never* enter its critical region. Let us add a new proposition '`in-crit(i)`' that holds when a process `i` is in its '`crit`' state. The following verification fails with a counterexample where process '`1`' is "stuck" on its '`down`' state.

```
reduce in CHECK :
  modelCheck(create-conf(3), <> in-crit(1))
result ModelCheckResult :
  counterexample

{prc (1, rem) prc (2, rem) prc (3, rem)}
{prc (1, down) prc (2, rem) prc (3, rem)}
{prc (1, down) prc (2, down) prc (3, rem)}
{prc (1, down) prc (2, crit) prc (3, rem)}
{prc (1, down) prc (2, up)  prc (3, rem)}
{prc (1, down) prc (2, up)  prc (3, down)}
{prc (1, down) prc (2, rem) prc (3, down)}
{prc (1, down) prc (2, down) prc (3, down)},
{prc (1, down) prc (2, crit) prc (3, down)}
```

1.2.3 Dining Philosophers

This Section presents a solution to Dijkstra's "Dining Philosophers" problem as described in [4]. This solution is based on breaking the symmetry on the moment in which each philosopher acquires its fork: philosophers with even pids first attempt to acquire the fork at their left, while philosophers with odd pids first attempt to acquire the fork at their right.

By definition, the right fork of a philosopher `i` has number `i`, and the left fork has number `i+1` mod `n`. When there is a competition to acquire a fork, the pids of the competing philosophers are inserted on a queue present in each fork. As each philosopher is done with the fork, it removes its pid from the queue.

The MSDF specification is as follows. First we need to map each fork id to a list of pids to implement the queue on each fork. The set '`Pids`' defines that list of pids, while '`Queue`' defines the map from integers (fork ids) to '`Pids`'. Although a specific queue needs to be shared only between two philosophers, to simplify the specification we opted to make it globally shared by creating a read-write component indexed by '`q`'.

```
Pids = (Int) List .
Queue = (Int, Pids) Map .
Label = {q : Queue, q' : Queue, ...} .
```

The specification is parameterized by a constant '`n`', which should be instantiated through an equation to the correct number of philosophers on the table.

```
Int ::= n .
```

Each philosopher is a process with the following states. Each state will be detailed on the subsequent transitions.

```
St .
St ::= srem
     | stest-right
     | stest-left
     | sleave-try
     | scrit
     | sreset-right
```

```
    | sreset-left
    | sleave-exit
    | stry
    | sexit .
```

Let us show only the transitions for odd-numbered processes. The even-numbered transitions are symmetric to the ones shown here. Initially, all philosophers are hungry and will attempt to acquire their forks, that is all processes are in the state '`stry`'. Odd-numbered processes, selected with the predicate '`odd(i)`', attempt to acquire their right forks (state '`stest-right`').

```
                      odd (Int)
-- ------------------------------------------------
prc (Int, stry) : Proc --> prc (Int, stest-right) .
```

At this point we make a slight modification to the original algorithm. The original rule is the following: if the fork is unavailable, the process put its pid on the queue, and go back to test if its pid reached the beginning of the queue, as the rule below shows. Recall from Section **??** that '`insert-back`' and '`first`' are functions operating on parameterized lists.

```
odd (Int),
Pids := lookup (Int, Queue),
Pids' := if (not Int in Pids)
         then insert-back (Int, Pids) else Pids fi,
Queue' := (Int |-> Pids') / Queue,
St := if first (Pids') == Int
         then stest-left else stest-right fi
-- ------------------------------------------
prc (Int, stest-right) : Proc
   -{q = Queue, q' = Queue', -}-> prc (Int, St) .
```

This *busy waiting* makes verification more complex since, if the algorithm is incorrect, it would enter a *livelock* and not in a *deadlock*. Deadlocks are easier to check with Maude: it needs only to look for a state to which no rule applies, since the system is reactive. We opted to change the algorithm by allowing at most one process in the queue, making it behave as a semaphore. The transition will only happen when a process successfully acquires a fork by putting its pid on the queue and immediately checking that it is at the

beginning of the queue—that is, the queue was empty. If the fork is successfully acquired, the process moves to acquire its left fork by changing its state to '`stest-left`', otherwise it does not change its state.

```
odd (Int),
Pids := lookup (Int, Queue),
Pids' := if (not Int in Pids)
         then insert-back (Int, Pids) else Pids fi,
Queue' := (Int |-> Pids') / Queue,   first (Pids') == Int
-- ------------------------------------------------------
prc (Int, stest-right) : Proc
   -{q = Queue, q' = Queue', -}-> prc (Int, stest-left) .
```

Not only is this rule simpler than the previous one, it also has the advantage of creating a *deadlock* instead of a *livelock* if the specification has any problems.

The rule for the '`stest-left`' state is similar to the rule for '`stest-right`'. The difference is that, when the left fork is acquired, the process moves to '`sleave-sty`'.

```
odd (Int),
Pids := lookup (((Int + 1) rem n), Queue),
Pids' := if (not Int in Pids)
         then insert-back (Int, Pids)
         else Pids fi,
Queue' := ((((Int + 1) rem n)) |-> Pids') / Queue,
                                     first (Pids') == Int
-- ------------------------------------------------------
prc (Int, stest-left) : Proc
   -{q = Queue, q' = Queue', -}-> prc (Int, sleave-try) .
```

One in the '`sleave-sty`' state, a process moves to is critical region.

```
                    odd (Int)
-- ---------------------------------------------
prc (Int, sleave-try) : Proc --> prc (Int, scrit) .
```

After accessing its critical region, a process moves to the '`sexit`' state, in which it first puts the right fork down, and then the left.

```
                      odd (Int)
-- ---------------------------------------------
prc (Int, scrit) : Proc -{-}-> prc (Int, sexit) .

   odd (Int)
-- ---------------------------------------------
prc (Int, sexit) : Proc --> prc (Int, sreset-right) .
```

In order to put the right fork down, it must remove itself from the queue on that fork. Since the queue only had the pid of the process, it will be empty after this operation.

```
odd (Int), Pids := lookup (Int, Queue),
Pids' := remove (Int, Pids),
Queue' := (Int |-> Pids') / Queue
-- ---------------------------------
prc (Int, sreset-right) : Proc
   -{q = Queue, q' = Queue', -}->
              prc (Int, sreset-left) .
```

The same process is make for the left fork.

```
odd (Int), Pids := lookup (((Int + 1) rem n), Queue),
Pids' := remove (Int, Pids),
Queue' := ((((Int + 1) rem n)) |-> Pids') / Queue
-- -----------------------------------------------------
prc (Int, sreset-left) : Proc
   -{q = Queue, q' = Queue', -}-> prc (Int, sleave-exit) .
```

One the left fork is taken down, a process goes to its '`srem`' state, which models the philosopher thinking.

```
                      odd (Int)
-- ---------------------------------------------
prc (Int, sleave-exit) : Proc --> prc (Int, srem) .
```

After thinking for a while a philosopher gets hungry again and returns to its '`stry`' state.

```
                    odd (Int)
 -- ---------------------------------------
 prc (Int, srem) : Proc --> prc (Int, stry) .
```

Searching for a final state on with the '`search`' command with the '`=>!`' predicate relation is a good way of finding a deadlock on the algorithm, since a final state is a state in which no rule applies, meaning that the entire pool of processes is stopped and cannot continue to evolve.

The auxiliar function '`initial-conf`' creates an initial configuration with the desired number $\mathtt{n}$ of philosophers. For $\mathtt{n} = 4$, the algorithm takes 3.6 seconds to find that there is no final state, as we expect on a correct configuration.

```
rewrites: 760825 in 3604ms cpu (3646ms real)
          (211079 rewrites/second)
search in SEARCH : initial-conf =>! C:Conf .

No solution.
```

When $\mathtt{n} = 6$, the search takes two minutes.

```
rewrites: 26197002 in 127450ms cpu (127420ms real)
          (205547 rewrites/second)
search in SEARCH : initial-conf =>! C:Conf .

No solution.
```

We may further test the specification using more searches. For example we know that, in a configuration with four philosophers, two philosophers may eat at the same time (that is, be at their respective '`scrit`' states), however, a philosopher may never eat concurrently with its neighbor. We may verify this by asking '`search`' to return all states in which two philosophers are in their '`scrit`' states:

```
search in SEARCH : initial-conf =>*
 < (prc(I1:Int,scrit)prc(I2:Int,scrit) S:Soup)::: 'Soup,
   R:Record > .

I1:Int <- 0 ; I2:Int <- 2
```

```
I1:Int <- 1 ; I2:Int <- 3
I1:Int <- 2 ; I2:Int <- 0
I1:Int <- 3 ; I2:Int <- 1
```

Another search confirms that three philosophers never eat at the same time in a four-philosopher configuration.

```
search in SEARCH : initial-conf =>*
 <(prc(I1:Int,scrit)prc(I2:Int,scrit)prc(I3:Int,scrit)
   S:Soup)::: 'Soup,R:Record > .

No solution.
```

Because the specification does not have justice, it is possible that a particular philosopher may never have the chance to eat, as the following model checking shows. The proposition '`state(i,s)`' holds when process `i` is in state `s`. Looking at the counterexample, we notice that process '`0`' is "stuck" in '`sleave-try`' while process '`2`' keeps entering and leaving its critical region indefinitely.

```
rewrites: 3539 in 60ms cpu (60ms real) (58983 rewrites/second)
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> state(0,scrit))
result ModelCheckResult :
  counterexample(
  { prc(0,stry) prc(1,stry) prc(2,stry) prc(3,stry)}...,

  { prc(0,sleave-try) prc(2,srem) ... }
  { prc(0,sleave-try) prc(2,stry) ... }
  { prc(0,sleave-try) prc(2,stest-left) ... }
  { prc(0,sleave-try) prc(2,stest-right) ... }
  { prc(0,sleave-try) prc(2,sleave-try) ... }
  { prc(0,sleave-try) prc(2,scrit) ... }
  { prc(0,sleave-try) prc(2,sexit) ... }
  { prc(0,sleave-try) prc(2,sreset-left) ... }
  { prc(0,sleave-try) prc(2,sreset-right) ... }
  { prc(0,sleave-try) prc(2,sleave-exit) ... }
```

1.2.4 Dining Philosophers, terminating specification

This Section presents a variant of the specification in which each philosopher, after eating, prints out its pid and stops. This specification was inspired by the one present on [3].

The specification is similar to the one shown in Section 1.2.3 with some modifications. The first is the addition of a write-only component '`Int*`', indexed by '`out'`', to model the output of information by the processes.

```
Label = {out' : Int*, q : Queue, q' : Queue, ...} .
```

We also change the rule for the '`scrit`' state, making the process output its pid.

```
                      odd (Int)
-- ------------------------------------------------------
prc (Int, scrit) : Proc -{out' = Int, -}-> prc (Int, sexit) .
```

The following rule for the '`srem`' state is removed, since, a philosopher no longer gets hungry again after thinking.

```
              odd (Int)
-- ----------------------------------------
prc (Int, srem) : Proc --> prc (Int, stry) .
```

This slight modification of the algorithm allows for more interesting verifications. Searching for all final states using the '`search`' command, we must arrive in states in which the '`out'`' component contains all the pids of the processes in the configuration.

```
search in SEARCH : initial-conf =>! C:Conf .

Solution 1
C:Conf <- <( prc(0,srem) prc(1,srem) prc(2,srem) prc(3,srem))
 {..., out' = 0,1,2,3}>

Solution 2
C:Conf <- <( prc(0,srem) prc(1,srem) prc(2,srem) prc(3,srem))
 {...,out' = 0,1,3,2}>
```

```
Solution 3
C:Conf <- <( prc(0,srem) prc(1,srem) prc(2,srem) prc(3,srem))
 {...,out' = 0,3,1,2}>
...
```

There are several possible variations of the contents of the '`out`' component, since the order in which a philosopher eats is non-deterministic.

Following the example in [3] let us model check this specification using a proposition '`check(i)`' which holds when the component '`out`' contains all numbers less than `i`.

```
rewrites: 505248 in 2450ms cpu (2440ms real)
          (206223 rewrites/second)
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> check (n - 1))
result Bool :
  true
```

In this case, since a process eventually stops, all processes eventually eat. The example below shows the case of process '`0`'. Recall that '`state(i,s)`' holds then process `i` is in state `s`.

```
rewrites: 176389 in 1750ms cpu (1750ms real)
          (100793 rewrites/second)
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> state (0,scrit))
result Bool :
  true
```

1.2.5 Dining Philosophers, fair scheduling

It is interesting to see what is the effect of adding a fair scheduling, according to the discussion on Section 1.1.2, on the specification. Let us make these changes to the terminating specification (Section 1.2.4), but they are easily adapted to the looping specification. Besides the scheduling rules, of course, the only change to that specification is the addition of the following rule:

```
                  odd (Int)
 -- ---------------------------------------
 prc (Int, srem) : Proc --> prc (Int, srem) .
```

This is necessary because a process needs to pass its turn to the next process when it is in its stopped mode.

The interesting result of this change is that the verification capacity is greatly enhanced. For example, let us check a 200-philosopher configuration for a deadlock. Notice that there is no final state, now that, upon termination, process keep "passing the turn" indefinitely.

```
rewrites: 86864 in 10442ms cpu (10501ms real)
          (8318 rewrites/second)
search in SEARCH : initial-conf =>! C:Conf .

No solution.
```

The model checking of the '`check(i)`' proposition is also successful.

```
rewrites: 1661019 in 42601ms cpu (43419ms real)
          (38989 rewrites/second)
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> check(n - 1))
result Bool :
  true
```

As it was expected with a fair scheduling of the execution, a process will now eventually eat.

```
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> state(0,scrit))
result Bool :
  true
...
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> state(199,scrit))
result Bool :
  true
```

1.2.6 An incorrect solution for the dining philosophers

This Section shows how a deadlock is detected in an incorrect specification, which, as we outlined on Section 1.2.3, is one that does not break the symmetry on the order on which each philosopher acquire its fork.

We may "break" any of the specifications described so far by removing the subset of the rules that applies to odd (or even) processes and, of course, removing the predicate '`odd(i)`' (or '`even(i)`') from the condition on the rules. Clearly, this should lead to a deadlock. In what follows we attempt to verify this deadlock with each of the several variants of the solution we developed.

Let us begin with the looping specification, in which each philosopher gets hungry again after thinking. A search for a final state with the '`search`' command with a configuration with four philosophers finds the deadlocked state: all philosophers are "stuck," holding their left forks.

```
search in SEARCH : initial-conf =>! C:Conf .

Solution 1
C:Conf <-
   < prc (0,stest-left) prc (1,stest-left)
     prc (2,stest-left) prc (3,stest-left))
     { q = (0 |->[0] +++ 1 |->[1] +++
            2 |->[2] +++ 3 |->[3]) } >

No more solutions.
```

With the terminating specification, the search finds not only the states in which the philosophers successfully eat, but also the deadlock state ('`Solution 1`').

```
search in SEARCH : initial-conf =>! C:Conf .

Solution 1
C:Conf <-
  < prc (0, stest-left) prc (1, stest-left)
    prc (2, stest-left) prc (3, stest-left)),
    { fair = 0,out' = (),
      q = (0 |-> [0] +++ 1 |-> [1] +++
```

```
          2 |-> [2] +++ 3 |-> [3]) }>

Solution 2
C:Conf <-
  < prc (0, srem) prc (1, srem)
    prc (2, srem) prc (3, srem),
    { fair = 0, out' = 0,1,2,3,
      q =(0 |-> [] +++ 1 |-> [] +++
          2 |-> [] +++ 3 |-> []) }>

Solution 3
C:Conf <-
  < prc (0, srem) prc (1, srem)
    prc (2, srem) prc (3, srem),
    { fair = 0, out' = 0,1,3,2,
      q =(0 |-> [] +++ 1 |-> [] +++
          2 |-> [] +++ 3 |-> []) }>
...
```

The specification with the round-robin scheduling is also prone to the deadlock.

```
rewrites: 671 in 20ms cpu (20ms real) (33550 rewrites/second)
search in SEARCH : initial-conf =>! C:Conf .

Solution 1
C:Conf <-
   < prc (0,stest-left) prc (1,stest-left)
     prc (2,stest-left) prc (3,stest-left)),
     { fair = 0, out' = (),
       q =(0 |->[0]+++ 1 |->[1]+++ 2 |->[2]+++ 3 |->[3]) }>
```

Recall that, with a fair scheduling policy, the model checking of a proposition that states that eventually a process will enter its critical region succeeds. The following shows that this is no longer the case, and presents as counterexample the same deadlocked situation, omitted here for brevity.

```
rewrites: 2520 in 50ms cpu (50ms real) (50400 rewrites/second)
```

```
reduce in MODEL-CHECK :
  modelCheck(initial-conf,<> state(0,scrit))
result ModelCheckResult :
  counterexample(...)
```

1.2.7 Bakery algorithm

This Section presents a specification of Lamport's Bakery Algorithm, described in [4]. Its primary objective it to give a verification example of an unbounded algorithm using an abstraction [6, 7]. Intuitively, the algorithm simulates a bakery (in Lamport's conception of how a bakery works) where customers wait for their turn by drawing tickets when they enter and are served in the order of their ticket numbers.

Let us begin the formal description by defining two read-write components: '`ch`' models whether a process is choosing its number or not; '`nm`' holds the chosen number. Both components are of the same type, '`IntM`', which is a map from integers (the pids) to integers (the chosen numbers).

```
IntM = (Int, Int) Map .
Label = {ch : IntM, ch' : IntM,
         nm : IntM, nm' : IntM, ...} .
```

A process may go through the following states, explained throughout the transition rules.

```
St .
St ::= choosing (Int, Int)
     | waiting (Int)
     | rem
     | crit
     | try
     | exit .
```

When a process wants to go into its critical region, it tells others that it is doing so by changing its entry on the '`ch`' component to '`1`'. The process then choose a number that is greater than all the number chosen by other processes. This is done in the '`choosing(i,m)`' state, which `i` contains the number of processes left to check and `m` the greatest number found so far. Let us assume that the constant '`n`' will be bound, by an equation, to the number of processes currently running.

```
           IntM' := (Int |-> 1) / IntM
-- ------------------------------------------------
prc (Int, try) : Proc -{ch = IntM, ch' = IntM', -}->
               prc (Int, choosing (n - 1, -1)) .
(Int1 >= 0), (Int1 =/= Int), (Int2' := lookup (Int1, IntM)),

Int3 := if Int2' > Int2 then Int2' else Int2 fi
-- ------------------------------------------------------
prc (Int, choosing (Int1, Int2)) : Proc
                      -{nm = IntM, nm' = IntM, -}->
                      prc (Int, choosing (Int1 - 1, Int3)) .
```

During the choosing process, a process must ignore its own number.

```
prc (Int, choosing (Int, Int2)) : Proc -->
               prc (Int, choosing (Int - 1, Int2)) .
```

When $\mathtt{i} = -1$, the greatest number found is $\mathtt{m}$. The process then chooses as its own number $\mathtt{m} + 1$ and goes to the next phase of the algorithm.

```
Int'' := (Int' + 1), IntM'1 := (Int |-> 0) / IntM1,
IntM'2 := (Int |-> Int'') / IntM2
-- ------------------------------------------------
prc (Int, choosing (-1, Int')) : Proc
               -{ch = IntM1, ch' = IntM'1,
                 nm = IntM2, nm' = IntM'2, -}->
                           prc (Int, waiting (0)) .
```

On this phase, a process keeps a constant watch on the other processes, iterating through the '`waiting(i)`' state, where $0 \leq \mathtt{i} \leq \mathtt{n} - 1$ (recall that $\mathtt{n}$ is the number of processes). It waits until its number if the lowest of all in order to access its critical region and avoids comparing with any process that is currently choosing its own number.

Since there is a possibility that *several* processes begin the choosing process at the same time, it may happen that processes choose the same number. In order to deal with this, the comparison to find the lowest number is made lexicographically using $(\mathtt{i}, \mathtt{p})$ where $\mathtt{i}$ is the process number and $\mathtt{p}$ its pid. This is formalized by the transition below, which specifies that process '`Int`' is comparing its number with process '`Int''`'. The predicate '`Int1' == 0`'

first makes sure that process 'Int'' is not choosing a number. If the chosen number of process 'Int'' is zero ('Int2' == 0'), process 'Int'' just left the critical region and process 'Int' may access it directly, otherwise the lexicographical comparison is made.

```
prc (Int, waiting (Int)) : Proc -->
    prc (Int, waiting ((Int + 1) rem n)) .

  Int' =/= Int,
  (Int1' := lookup (Int', IntM1)),
  (Int2' := lookup (Int', IntM2)),
  (Int2  := lookup (Int, IntM2)),
  St := if Int1' == 0 and
           (Int2' == 0 or
            ((Int2 < Int2') or
             (Int2 == Int2' and Int < Int')))
        then crit else waiting ((Int' + 1) rem n)
        fi
 -- ------------------------------------------
 prc (Int, waiting (Int')) : Proc
                -{ch = IntM1, ch' = IntM1,
                   nm = IntM2, nm' = IntM2, -}->
                                  prc (Int, St) .
```

Upon exiting its critical region, a process, as we said, changes its chosen number to zero and moves to its 'rem' state. Once in its 'rem' state, a process attempts to access the critical region again by moving to its 'try' state.

```
                IntM2' := (Int |-> 0) / IntM2
 -- ------------------------------------------------------
 prc (Int, crit) : Proc -{nm = IntM2, nm' = IntM2', -}->
                                            prc (Int, rem) .

 prc (Int, rem) : Proc --> prc (Int, try) .
```

Unfortunately, this algorithm does not have an upper bound on the chosen number. Also, the apparently trivial solution of using integers modulo some very large b also fails.

We may verify that there is no upper bound on the chosen number using a '`search`', showing that, with two processes, the chosen number can easily reach ten (or any other natural). The problem happens when a process chooses a number while the other process is in its critical region. A process only zeroes its chosen number *after* leaving the critical region. It works as follows: process '`0`', with a chosen number of 2, is in its critical region; process '`1`' chooses 3 as its number; when this process is in its critical region, process '`0`' gets another number, which is 4, and so on.

```
search [1] in BAKERY : initial-conf =>*
    < S:Soup,{PR:PreRecord,n = (0 |-> 10 +++ 1 |-> I:Int)} > .


...
< (prc(0, waiting(1)) prc(1, crit)),
  {n = (0 |-> 2 +++ 1 |-> 1)} >
< (prc(0, waiting(1)) prc(1, rem)),
  {n = (0 |-> 2 +++ 1 |-> 0)} >
...
< (prc(0, crit) prc(1, choosing(-1, 2))),
  {n = (0 |-> 2 +++ 1 |-> 0)} >
...
< (prc(0, waiting(1)) prc(1, crit)),
  {n = (0 |-> 4 +++ 1 |-> 3)} >
...
< (prc(0, crit) prc(1, choosing(-1, 4))),
  {n = (0 |-> 4 +++ 1 |-> 0)} >
...
< (prc(0, waiting(1)) prc(1, crit)),
  {n = (0 |-> 6 +++ 1 |-> 5)} >
...
< (prc(0, crit) prc(1, choosing(-1, 6))),
   {n = (0 |-> 6 +++ 1 |-> 0)} >
...
< (prc(0, waiting(1)) prc(1, crit)),
  {n = (0 |-> 8 +++ 1 |-> 7)} >
...
< (prc(0, crit) prc(1, choosing(-1, 8))),
  {n = (0 |-> 8 +++ 1 |-> 0)} >
```

```
...
< (prc(0, choosing(1, -1)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 9)} >
< (prc(0, choosing(0, 9)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 9)} >
< (prc(0, choosing(-1, 9)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 9)} >
< (prc(0, waiting(0)) prc(1, waiting(0))),
  {n = (0 |-> 10 +++ 1 |-> 9)} >
```

Let us naively modify the algorithm so that the chosen number is incremented modulo, say, 2267. The algorithm fails for the same reason: chosen numbers gets increasingly high and, using arithmetic modulo 2267, they will eventually be zero, a number that is obviously smaller than all other numbers, as the following search for a race condition shows:

```
search [1] in BAKERY : initial-conf =>*
    < (prc(0, crit) prc(1, crit)) ::: 'Soup,{PR:PreRecord} > .

Solution 1 (state 183534)
states: 183535  rewrites: 5607219 in 34110ms cpu
    (34130ms real) (164386 rewrites/second)
PR:PreRecord --> ch = (0 |-> 0 +++ 1 |-> 0),
                      n = (0 |-> 2266 +++ 1 |-> 0)


...
< (prc(0, try) prc(1, choosing(-1, 2264))) ,
  {n = (0 |-> 0 +++ 1 |-> 0)} >
< (prc(0, choosing(1, -1)) prc(1, choosing(-1, 2264))),
  {n = (0 |-> 0 +++ 1 |-> 0)} >
< (prc(0, choosing(1, -1)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 2265)} >
< (prc(0, choosing(0, 2265)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 2265)} >
< (prc(0, choosing(-1, 2265)) prc(1, waiting(0))),
  {n = (0 |-> 0 +++ 1 |-> 2265)} >
< (prc(0, waiting(0)) prc(1, waiting(0))),
  {n = (0 |-> 2266 +++ 1 |-> 2265)} >
```

```
< (prc(0, waiting(1)) prc(1, waiting(0))),
  {n = (0 |-> 2266 +++ 1 |-> 2265)} >
< (prc(0, waiting(1)) prc(1, crit)),
  {n = (0 |-> 2266 +++ 1 |-> 2265)} >
< (prc(0, waiting(1)) prc(1, rem)),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, rem)),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, try)),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, choosing(1, -1))),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, choosing(0, -1))),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, choosing(-1, 2266))),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, waiting(0))),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
< (prc(0, crit) prc(1, crit)),
  {n = (0 |-> 2266 +++ 1 |-> 0)} >
```

In order to make this algorithm amenable to verification, we must create an abstraction that captures the essence of the algorithm, but does not have the infinite number of states of the original. The solution follows the ideas described in [6], in which a two-process abstraction is defined and proved to correctly simulate the original specification.

The key to find the correct abstraction in this case is to realize that the actual *absolute value* of the chosen number is not important, but its *relative value* with regard to the other numbers.

We begin with the following two equations: a process changes its number to zero after leaving the critical zone, so the number chosen by the other process in this case does not need to grow indefinitely: choosing number one is sufficient.

```
ceq (< S:Soup, { n = (0 |-> 0 +++ 1 |-> I), PR } >)
  = < S:Soup, { n = (0 |-> 0 +++ 1 |-> 1), PR } >
if I > 1 .
```

```
ceq (< S:Soup, { n = (0 |-> I +++ 1 |-> 0), PR } >)
  = < S:Soup, { n = (0 |-> 1 +++ 1 |-> 0), PR } >
if I > 1 .
```

Next, the following equations keep the chosen numbers of both processes from growing indefinitely, while keeping their relative values.

```
ceq (< S:Soup, { n = (0 |-> I +++ 1 |-> I'), PR } >)
  = < S:Soup, { n = (0 |-> 2 +++ 1 |-> 1), PR } >
if (I' < I) /\ not (I' == 1 and I == 2) .

ceq (< S:Soup, { n = (0 |-> I +++ 1 |-> I'), PR } >)
  = (< S:Soup, { n = (0 |-> 1 +++ 1 |-> 1), PR } >)
if not (I' < I) /\ not (I' == 1 and I == 1) .
```

With these abstractions we may now try a search for a race condition.

```
rewrites: 4195 in 61ms cpu (61ms real) (67671 rewrites/second)
search in CHECK :
   initial-conf =>* <(prc(0,crit)prc(1,crit))::: 'Soup,
   {PR:PreRecord}> .

No solution.
```

Also, both processes eventually reach their critical region, according to the results of the two searches below:

```
rewrites: 3463 in 36ms cpu (36ms real) (93609 rewrites/second)
search in CHECK :
   initial-conf =>* <(prc(0,crit)prc(1,St:St))::: 'Soup,
   {PR:PreRecord}> .

Solution 1
PR:PreRecord <- ch =(0 |-> 0 +++ 1 |-> 0),
                n =(0 |-> 1 +++ 1 |-> 1);
St:St <- waiting(0)
rewrites: 3376 in 26ms cpu (26ms real) (125055 rewrites/second)

search in CHECK :
```

```
   initial-conf =>* <(prc(1,crit)prc(0,St:St))::: 'Soup,
   {PR:PreRecord}> .

Solution 1
PR:PreRecord <- ch =(0 |-> 0 +++ 1 |-> 0),
                n =(0 |-> 2 +++ 1 |-> 1);
St:St <- waiting(0)
Bye.
```

1.2.8 Leader election on an asynchronous ring

This Section specifies the algorithm for *leader election* on an unidirectional, asynchronous ring. It is used as an example of a specification that uses the message-passing model and provides us with more complex model checking examples. The intuitive idea behind this algorithm is to elect as leader the process that has the highest pid of all processes in the ring. Each process forwards its own pid to its neighbor. A process, upon receiving a pid that is greater than its own, forwards it to its neighbor. The greatest pid will eventually circle the ring arriving back at its origin. When a process receives its own pid from a neighbor, it knows it is the leader. It may initiate now, for example, a broadcast announcing the leader election.

Let us begin the formal description of the algorithm by defining the format of the messages. It contains as first argument a pid and as the second argument the destination of the message. There is no need to keep track of the source of the message, as we are dealing with a known network topology.

```
Msg ::= m Int to Int .
```

The ring network is modelled in this specification by having each process knowing the pid for its neighbor. Only one neighbor is known, hence communication in this specification is made in only one direction throughout the ring.

```
Proc ::= prc (Int, Int', St) .
```

As usual, we show the states of a process, while explaining their meaning on the subsequent transitions.

```
St ::= start
     | waiting
     | leader .
```

At the beginning of the algorithm, each process sends its pid to its neighbor.

```
prc (Int, Int', start) : Soup -->
               prc (Int, Int', waiting) (m Int to Int') .
```

When a process receives a message from a neighbor, it compares its pid `i` with the pid `i'` received from its neighbor. If $\mathtt{i}' > \mathtt{i}$, it forwards the message to its own neighbor.

```
   Int'' > Int
-- ------------------------------------------------------
prc (Int, Int', waiting) (m Int'' to Int) : Soup -->
            prc (Int, Int', waiting) (m Int'' to Int') .
```

If $\mathtt{i}' < \mathtt{i}$, it removes the message from the ring.

```
   Int'' < Int
-- ------------------------------------------------------
prc (Int, Int', waiting) (m Int'' to Int) : Soup -->
            prc (Int, Int', waiting) .
```

When $\mathtt{i}' = \mathtt{i}$ the process know it is the leader.

```
   Int'' == Int
-- ------------------------------------------------------
prc (Int, Int', waiting) (m Int'' to Int) : Soup -->
                                prc (Int, Int', leader) .
```

In order to verify the correctness of the specification, let us make some verifications using Maude's model checker on a configuration with four processes. We begin by creating an operation '`leaders(S)`' that computes the number of leaders in a soup `S`.

```
op leaders : Soup -> Int .

eq leaders (S S') = leaders (S) + leaders (S') .
eq leaders (prc (I, I', leader)) = 1 .
eq leaders (prc (I, I', waiting)) = 0 .
eq leaders (prc (I, I', start)) = 0 .
eq leaders (m I to I') = 0 .
```

The proposition '`one-leader`' holds when there is exactly one leader on the configuration, while '`no-leader`' holds when there is no leader on the configuration.

```
op one-leader : -> Prop .
op no-leader : -> Prop .

eq < S ::: 'Soup, R > |= one-leader = (leaders (S) == 1) .
eq < S ::: 'Soup, R > |= no-leader = (leaders (S) == 0) .
```

We may now model check our first formula: in all executions of the specification, there is always one leader.

```
rewrites: 7339322 in 56444ms cpu (56447ms real)
          (130027 rewrites/second)
reduce in CHECK :
  modelCheck(init, <> [] one-leader)
result Bool :
  true
```

There is no execution in which a leader is not elected.

```
rewrites: 7204804 in 56551ms cpu (57278ms real)
          (127402 rewrites/second)
reduce in CHECK :
  modelCheck(init, ~ [] no-leader)
result Bool :
  true
```

In all executions, there is no leader until a leader is selected.

```
rewrites: 7340679 in 57867ms cpu (58915ms real)
          (126853 rewrites/second)
reduce in CHECK :
  modelCheck(init, [] (no-leader U one-leader))
result Bool :
  true
```

References

[1] G. Berry and G. Boudol. The chemical abstract machine. In *Conf. Record 17th ACM Symp. on Principles of Programmming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan. 1990*, pages 81–94. ACM Press, New York, 1990.

[2] Fabricio Chalub. An Implementation of Modular Structural Operational Semantics in Maude. Master's thesis, Universidade Federal Fluminense, 2005. `http://www.ic.uff.br/~frosario/dissertation.pdf`.

[3] Azade Farzan, Feng Chen, José Meseguer, and Grigore Roşu. Formal analysis of Java programs in JavaFAN. In Rajeev Alur and Doron A. Peled, editors, *CAV*, Lecture Notes in Computer Science. Springer, 2004.

[4] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[5] José Meseguer. Rewriting as a unified model of concurrency. Technical Report SRI-CSL-90-02R, SRI International, Computer Science Laboratory, February 1990. Revised June 1990. Appendices on functorial semantics have not been published elsewhere.

[6] José Meseguer, Miguel Palomino, and Narciso Martí-Oliet. Notes on model checking and abstraction in rewriting logic. http://maude.cs.uiuc.edu/.

[7] José Meseguer, Miguel Palomino, and Narciso Martí-Oliet. Equational abstractions. In Franz Baader, editor, *Automated Deduction - CADE-19. 19th International Conference on Automated Deduction, Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volume 2741 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[8] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[9] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.