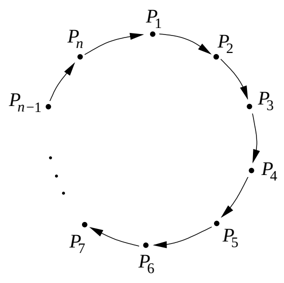
P. Müller and D. Basin

Formal Methods and Functional Programming

Optional Exercises 14: Modeling in Promela

As usual, the solutions can be found at the end of the file.

Assignment A (Leader Election Protocol)



Consider the following leader election protocol. For $n \geq 1$, the processes $P_1, \ldots, P_n$ are located in a ring topology, where each process is connected by an unidirectional channel to its neighbor as outlined in the figure to the right.

To distinguish the processes, each process has a unique identifier id with $1 \leq id \leq n$. The aim is to elect the process with the highest identifier as the leader within the ring. Therefore, each process executes the following algorithm:

```
send message $id$
loop
    receive message $m$
    if $m = id$ then stop
    if $m > id$ then send message $m$
end loop
```

Task A.1. Model this leader election protocol for n processes in Promela.

Hint: Use an array of n channels of length 1, i.e.,

```
#define n 5  // number of processes
#define l 1  // length of channel
chan c[n] = [l] of { byte }
```

Model a process in Promela as

```
proctype pnode(chan _in, out; byte id) {
    /* ... algorithm for electing the leader ... */
}
```

Task A.2. Assume that the channels are of length $n + 1$ instead of length 1 in your Promela model. Is there a state in some execution in which a channel stores more than n messages? Use Spin to verify your claim for some fixed values of n. What happens if the channels have length 0?

Assignment B (Dekker's Algorithm)

Dekker's algorithm is said to be the first known algorithm that (really) solves the mutual exclusion problem for two concurrent processes. In the algorithm, the critical section is protected by two bit-valued flags. The first one is actually a pair that is used to signal that the first and second, respectively, process is interested in entering the critical section. The other flag alternates and is used to decide which process may enter the critical section in case both are interested.

The algorithm guarantees mutual exclusion to the critical section as well as deadlock and starvation freedom. Instead of relying on low-level test-and-set instructions or interrupts, or on signal/wait thread operations, each process uses busy waiting to detect when it may enter the critical section. This makes the algorithm highly portable between different languages and hardware architectures, but also less efficient in case of lots of contention.

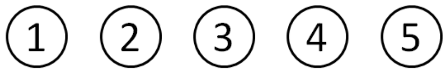
Task B.1. Implement Dekker's algorithm in Promela and verify exclusivity using Spin.

Hint: In order to verify the exclusivity of the critical section, let `mutex` count the number of processes that are currently in the critical section. Then, use the following *supervisor* process (also known as *monitor* or *watchdog*) to assert exclusivity:

```
proctype supervisor() {
    assert(mutex != 2)
}
```

Task B.2. Prove the property that a process enters the critical section of Dekker's algorithm infinitely often.

Assignment C (Mole Game)



Consider the following game between a mole (*Maulwurf*) and a hunter. The mole has five holes as in the above figure. At the beginning of the game, the mole hides in one of these holes. Now the game proceeds in rounds of the following form: the hunter checks one hole to see whether the mole is inside. If it is, the hunter wins the game. If not, the mole must move one hole to the left or right (if it is in the leftmost hole already, it must move to the right and conversely for the rightmost hole). After the mole has moved, the next round starts.

It turns out that the hunter has a strategy to win the game, no matter how the mole moves. One sure winning strategy for the hunter is to check holes in the sequence 2-3-4-2-3-4.

Your task is to model this game in Promela. Your model has to contain the data structure for the holes, the set-up of the initial state (the hiding of the mole), the behaviour of the mole, the implementation of the hunter's winning strategy, and an assertion that ensures that after executing the strategy, the hunter has definitely caught the mole.

Solution of assignment A (Leader Election Protocol)

Task A.1. See `leader_1.pml`. Note that this version doesn't terminate all processes – only the leader. We could also send a special "finished" message to each process to tidy up. As it is, we get "invalid endstate" failures if we search for them (but the computed leader ID is correct).

Task A.2. See `leader_2.pml`. It is not possible for one node to receive more than n messages (one per node). We formalise the property of interest as a single assertion, and placed it into a so-called watchdog process. Spin will test all possible interleavings of the pnode processes and the watchdog process. The resulting set of interleavings will contain those where the single assertion is checked after every possible state change involving the channels. Hence, the assertion must always hold, or Spin will complain.

The assertion will fail if you reduce the channel capacity, e.g., set it back to one again, as used in part a) of the assignment.

In case of a channel length $l = 0$, we have synchronous message sending, which means that sends are blocking. For the protocol under study, $l = 0$ leads to a deadlock (we do not detect this explicitly below, but could do so with extra instrumentation and an LTL formula checking that at it is always the case that at least one process has either terminated or will reach the beginning of its loop again.

Note that, unlike in Task A.1, we do not get invalid endstate errors for this model – this is just because the "watchdog" is never stuck.

Solution of assignment B (Dekker's Algorithm)

Task B.1. See `dekker_1.pml`. A comment about the supervisor thread: Let `mutex` denote the number of processes in the critical section. It might be tempting to model the supervisor as

```
proctype supervisor() {
    do
    :: assert mutex != 2
    od
}
```

in order to capture the idea that the assertion is checked in each possible state. The loop, however, is not necessary, that is,

```
proctype supervisor() {
    assert mutex != 2
}
```

suffices. Assume that there exists a state in which the assertion is violated. Since Spin generates every possible interleaving, it also generates one in which the assertion is executed in exactly this violating state.

In terms of performance, the loop-approach can even be considered a poor solution because it unecessarily increases the state space.

Task B.2. The first try is to introduce a variable `critical` whose value is equal to the id of the process that is inside the critical section (2 if none is) (the new file is called `dekker_2.pml`). Then model check the LTL properties `[]<>(critical==0)` and (`[]<>(critical==1)` for the first and second process, respectively. Both are expected to fail.

The problem with the property is *fairness*: we must ensure that each process gets executed infinitely often (otherwise a process can starve the other). It is possible to model check only fair paths in Spin, using the `-f` switch. We run the analysis again with `-f` and the property verifies.

Another way to have this work is to simulate some notion of fairness into our model. We introduce a pair of counters, one per process. A process decrements its counter when it enters its critical section and resets the counter of the other process. Once a counter is 0, the corresponding process blocks and then the other process may proceed. Modelling fairness this way makes our property go through (it is questionable as to whether a real implementation should work this way, since we could block processes unnecessarily, in cases where other threads are not interested in entering the critical section). A full solution can be found in `dekker_3.pml`.

Instructions for model checking in command line (use `-f` only for fair model checking):

```
spin -a dekker_3.pml
gcc pan.c
./a.exe -a -f
```

In the GUIs, this can be chosen by choosing "With Weak Fairness" in the Verification Options.

Solution of assignment C (Mole Game)

See `mole.pml`.