Course "Formal Methods"
Lab Test

Roberto Sebastiani
DISI, Università di Trento, Italy

February 15^{th}, 2018

769857918

[COPY WITH SOLUTIONS]

1 Spin

As a junior, unnamed, engineer of a spaceship that just landed on an unexplored planet, you are given the task to solve a puzzle given by an alien technology artefact that is blocking the way of your not-so-bright commander.

The puzzle consists of five `ControlPillars`, numbered from 0 to 4, controlling a gate. Initially, pillars 1, 3 and 4 are in `ON` (i.e. `true` or 1) state, while the remaining two pillars are OFF (i.e. `false` or 0). That is, the initial configuration is { `OFF, ON, OFF, ON, ON` }. The gate opens when all the pillars are contemporarily set to `ON`.

Each pillar is in a loop waiting for a command on an input channel `ctl` (passed as argument). Whenever a pillar receives a command, it changes its own state –and the state of its immediate left and right neighbours– to the opposite value[1]. **E.g.** starting from the initial configuration, if pillar 1 is given a command then the final configuration is { `ON, OFF, ON, ON, ON` }. In this regard, you can assume that the pillars are *placed* in a *virtual circle*, so that pillars 0 and 4 are considered neighbours of each other. Each pillar exits its control loop whenever the gate opens.

The `Commander` is stuck in a loop, continuously checking whether the gate has opened or not. If that is the case, the `Commander` prints a "Walk in..." message and exits the loop. Otherwise, the `Commander` chooses a random pillar, prints its *ID* on screen, and interacts with it by sending a message through its associated communication channel.

Model the system using one instance of `Commander()` process, five instances of `ControlPillar(id, ctl)` processes, and conveniently initialise the system using the `init()` process.

Use your *model checking* skills to aid the commander getting inside the gate, by writing a property `p1` s.t. its counter-example is a sequence of button-switches that will open the gate.

Also, briefly explain, as a comment on your solution, how you can use **Spin** in order to find [in finite time] the **minimum-length** sequence of switches that opens the gate. What is the sequence?

[1] **Hint:** ensure that both the receipt of a command and the change in the system configuration are executed as a single atomic sequence.

Solution:

```
#define OFF 0
#define ON  1

chan ps[5] = [0] of { bit };
bit  state[5] = { OFF, ON, OFF, ON, ON };

#define gate_is_open (state[0] & state[1] & state[2] & state[3] & state[4])

proctype ControlPillar(byte id; chan ctl)
{
    bit in;
    do
        :: atomic { ctl?in ->
            state[(id - 1 + 5) % 5] = ! state[(id - 1 + 5) % 5];
            state[id]               = ! state[id];
            state[(id + 1) % 5]     = ! state[(id + 1) % 5];
           }
        :: gate_is_open -> break;
    od
}

active proctype Commander()
{
    byte id;
    do
        :: !gate_is_open ->
            select(id:  0 .. 4);
            printf("E (%d)\n", id);
            ps[id]!1;
        :: else ->
           printf("Walk in..\n"); break;
    od;
}

init
{
    run ControlPillar(0, ps[0]); run ControlPillar(1, ps[1]);
    run ControlPillar(2, ps[2]); run ControlPillar(3, ps[3]);
    run ControlPillar(4, ps[4]);
}

ltl p1 { [] ! gate_is_open }

// ~$ spin -search -bfs elaaden_vault.pml # sequence is E(3), E(4)
```

2 nuXmv

Encode the following *sorting algorithm* for an array of length 5 in NuSMVor nuXmv as a **module** `gnomeSort(arr, len)`:

```
    procedure gnomeSort(arr, len):
l0:     pos := 0
l1:     while (pos < len):
l2:         if (pos == 0 or arr[pos] >= arr[pos - 1]):
l3:             pos := pos + 1
            else:
l4:             swap(arr[pos], arr[pos - 1])
                pos := pos - 1
l5:     return                  # self-loop here!
```

Declare, inside the **main** module, a variable `arr` and a variable `sorter`:

- `arr` is an array of 5 elements with domain in $[1, 10]$, initialised to { `9, 7, 5, 3, 1` }
- `sorter` is an instance of `gnomeSort(arr, len)` taking as input `arr` and its corresponding length, 5.

Verify that the following properties are `true`:

- the algorithm always terminates
- eventually in the future, the array will be sorted forever
- eventually the array is sorted, and the algorithm is not done until the array is sorted

Solution:

```
MODULE main()
VAR
    arr    : array 0..4 of 1..10;
    sorter : gnomeSort(arr, 5);

INIT arr[0] = 9 & arr[1] = 7 & arr[2] = 5 & arr[3] = 3 & arr[4] = 1;

MODULE gnomeSort(arr, len)
VAR
  pos : 0..len;
  pc  : { l0, l1, l2, l3, l4, l5 };

INIT pc = l0 & pos = 0;

ASSIGN
  next(pc) := case
      pc = l0 | pc = l3 | pc = l4                                                  : l1;
      pc = l1 & pos < len                                                          : l2;
      pc = l2 & ((pos = 0) | (pos > 0 & pos < len & arr[pos] >= arr[pos - 1]))     : l3;
      pc = l2                                                                      : l4;
      (pc = l1 & pos >= len) | pc = l5                                             : l5;
    esac;

  next(pos) := case
      pc = l3 & pos < len : pos + 1;
      pc = l4 & pos > 0   : pos - 1;
      TRUE                : pos;
    esac;

TRANS pc != l4 ->
            (next(arr[0]) = arr[0] & next(arr[1]) = arr[1] & next(arr[2]) = arr[2] &
             next(arr[3]) = arr[3] & next(arr[4]) = arr[4]);
TRANS (pc = l4 & pos = 1) ->
            (next(arr[0]) = arr[1] & next(arr[1]) = arr[0] & next(arr[2]) = arr[2] &
             next(arr[3]) = arr[3] & next(arr[4]) = arr[4]);
TRANS (pc = l4 & pos = 2) ->
            (next(arr[0]) = arr[0] & next(arr[1]) = arr[2] & next(arr[2]) = arr[1] &
             next(arr[3]) = arr[3] & next(arr[4]) = arr[4]);
TRANS (pc = l4 & pos = 3) ->
            (next(arr[0]) = arr[0] & next(arr[1]) = arr[1] & next(arr[2]) = arr[3] &
             next(arr[3]) = arr[2] & next(arr[4]) = arr[4]);
TRANS (pc = l4 & pos = 4) ->
            (next(arr[0]) = arr[0] & next(arr[1]) = arr[1] & next(arr[2]) = arr[2] &
             next(arr[3]) = arr[4] & next(arr[4]) = arr[3]);

DEFINE
    done := pc = l5;
```

```
-- The algorithm always terminates
CTLSPEC AF AG done ;

-- Eventually in the future, the array will be sorted forever
CTLSPEC AF AG (arr[0] <= arr[1] & arr[1] <= arr[2] &
               arr[2] <= arr[3] & arr[3] <= arr[4]) ;

-- Eventually the array is sorted, and the algorithm is not done until the array is sorted
CTLSPEC A[!done U (arr[0] <= arr[1] & arr[1] <= arr[2] &
                   arr[2] <= arr[3] & arr[3] <= arr[4])] ;
```