Course "Formal Methods"
Lab Test

Roberto Sebastiani
DISI, Università di Trento, Italy

June 07^{th}, 2018

769857918

[COPY WITH SOLUTIONS]

1 Spin

An *archery contest* is being held in town, with two participants.

Each `Bowman` fires 7 arrows aiming to the same `Target` pole which is positioned straight in front of them. When aiming, the hand of the first –more skilled– `Bowman` shakes less than that of the second `Bowman`. For this reason, the direction that is being imprinted to the arrow is selected in the interval $[-1, 1]$ for the first `Bowman` and in $[-2, 2]$ for the second `Bowman`. An `ARROW` can be fired with too much, or not enough, power. In this case, the `ARROW` does not reach the `Target` no matter its direction. The first `Bowman` has a likelihood of 1 over 4 to miss the `Target` in this way, whereas the second `Bowman` has a chance of 1 over 3. Successfully firing an arrow, i.e. with the right amount of power so as to hit the target, corresponds to sending an `ARROW` message through a channel. Each message contains as a payload the `direction` of the `ARROW` and the unique identifier of the `Bowman` who fired it.

In this scenario, the `Target` is a wooden pole thick 3 units. We assume that the `Target` is listening on the other side of the channel for any incoming message. The `Target` keeps track of the numbers of arrows that it received from each `Bowman` and their score. When an `ARROW` message is received, the `Target` assigns a score to the `Bowman` that fired it. Hitting the pole in the center (value: 0) awards 2 points, whereas hitting the left/right edges of the pole (values: -1 or 1) scores 1 point. Missing the pole gives no reward.

To make things more interesting, we take into account the effect of the *wind*, which can displace the direction of each `ARROW` by a randomly selected factor in the interval $[-1, 1]$. We assume that neither `Bowman` is able to compensate for the *wind* in advance, and that the wind can change direction at any time. To this extent, the `Target` "measures" the values of the wind right after it receives an `ARROW`, and takes that into account when it assigns the score.

example #1. The `Target` receives an `ARROW` with direction -1 from the first bowman, when the wind is measured to be -1. Thus, the overall direction of the arrow is -2 and no point is assigned.

example #2. The `Target` receives an `ARROW` with direction -2 from the second bowman, when the wind is measured to be $+1$. Thus, the overall direction of the arrow is -1 and the score of the second bowman is increased by 1.

When all arrows have been fired, the `Target` prints the score table and proclaims the winner.

Model the *archery contest* in *Promela* and verify using *Spin* that there exists no execution in which at least one bowman gets the maximum score (14).

Solution:

```
mtype = { ARROW };
chan bow = [1] of { mtype, pid, short };

byte score[2];
byte cc[2];

active [2] proctype bowman()
{
    byte i;
    short tremor;
    for (i: 1 .. 7) {
        select(tremor: -1 - _pid .. 1 + _pid);
        if
            :: true      -> bow!ARROW(_pid, tremor);
            :: true      -> bow!ARROW(_pid, tremor);
            :: _pid == 0 -> bow!ARROW(_pid, tremor);
            :: true;
        fi;
    }
};

active proctype target()
{
    short dir, wind, res;
    pid idx;
    do
        :: bow?ARROW(idx, dir) ->
            select(wind: -1 .. 1);
            res = dir + wind;
            score[idx] = score[idx] + (res == 0  -> 2
                                     : (res == 1  -> 1
                                     : (res == -1 -> 1
                                     :  0)));
            cc[idx] = cc[idx] + 1;
        :: timeout -> break;
    od;

    assert(score[0] != 14 && score[1] != 14);

    printf("Scores: [%d (%d) / %d (%d)]\n", score[0], cc[0], score[1], cc[1]);

    if
        :: score[0] > score[1] -> printf("Hans wins.\n");
        :: score[0] < score[1] -> printf("Henry wins.\n");
        :: else                -> printf("Draw.\n");
    fi;
};

// Verification:
// ~$ spin -a kcd1403.pml ; gcc -o run pan.c ; ./run -a
```

2 nuXmv

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

```
def min_max(arr, len):
    min = arr[0]
    max = arr[0]
    for cc in range(0, len):     # ~ [0, 1, 2, 3, 4]
        if arr[cc] < min:
            min = arr[cc]
        elif max < arr[cc]:
            max = arr[cc]
    return                       # self-loop here!
```

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

- `arr` is an array of 5 elements with domain in $[1, 10]$, initialized to { `5, 7, 9, 3, 2` }
- `mm` is an instance of `min_max(arr, len)` taking as input `arr` and its corresponding length, 5.

Hints:

- transform the `for` loop into a `while` loop before labeling the control points

Verify that the following properties are `true`:

- The execution does not reach the final state until the loop counter is equal to the array length
- Eventually in the future, the value of `min` will be equal 2 and the value of `max` will be equal 9
- The value of `max` is never larger than every value contained in the array at the same time
- Invariant: the value of `min` is smaller or equal `max`

Solution:

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

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

MODULE min_max(arr, len)
VAR
    pc  : { L0, L1, L2, L3, L4, L5, L6 };
    cc  : 0..len; min : 1..10; max : 1..10;

INIT pc = L0 & cc = 0 & min = arr[0] & max = arr[0];

ASSIGN
    next(pc) := case
        L0 = pc & cc < len                 : L1;
        L1 = pc & cc < len & arr[cc] < min : L2;
        L1 = pc                            : L3;
        L3 = pc & cc < len & max < arr[cc] : L4;
        L2 = pc | L3 = pc | L4 = pc        : L5;
        L5 = pc                            : L0;
        TRUE                               : L6;
    esac;

    next(cc) := case
        L5 = pc & cc < len : cc + 1;
        TRUE               : cc;
    esac;

    next(min) := case
        L2 = pc & cc < len : arr[cc];
        TRUE               : min;
    esac;

    next(max) := case
        L4 = pc & cc < len : arr[cc];
        TRUE               : max;
    esac;

-- The execution does not reach the final state until the loop counter has traversed the whole array
LTLSPEC (L6 != pc U cc = len);

-- Sooner or later, the minimum is equal 2 and the maximum is equal 9
LTLSPEC F (min = 2 & max = 9);

-- The value of max is never larger than every value contained in arr at the same time
LTLSPEC G !(max > arr[0] & max > arr[1] & max > arr[2] & max > arr[3] & max > arr[4]);

-- Invariant: the value of min is smaller or equal max
INVARSPEC max >= min;
```