Course "Formal Methods"
Lab Test

Roberto Sebastiani
DISI, Università di Trento, Italy

July 03^{rd}, 2018

769857918

[COPY WITH SOLUTIONS]

1 Spin

Two friends play the famous *Rock-Paper-Scissors* game to establish who is going to pay for dinner.

The friendly match has 10 rounds; the player which wins the largest number of rounds wins the game and pays for dinner. If neither player wins more rounds than the other, it is a tie and they split the bill in two equal halves.

The `Waiter` is responsible for managing the game and keeping the score. For each round, it sends a sequence of 3 messages to each player: "1", "2" and "3".

Both players do nothing upon receival of messages "1" and "2". When a `Player` receives message "3", it chooses a `hand` value among { `ROCK, PAPER, SCISSORS` } and sends it back to the `Waiter` for scrutiny. Each `Player` has a different likelyhood of choosing one `hand` value over the other:

- `Player 1` - `ROCK`: 1/4, `PAPER`: 1/2, `SCISSORS`: 1/4
- `Player 2` - `ROCK`: 2/5, `PAPER`: 1/5, `SCISSORS`: 2/5

At each round, the `Waiter` receives the `hand` value picked by each `Player` and keeps track of the score table according to the following rules:

- `PAPER` beats `ROCK`
- `ROCK` beats `SCISSORS`
- `SCISSORS` beat `PAPER`
- picking the same value results in a tie

example #1. `Player 0` plays `PAPER` and `Player 1` plays `ROCK`, thus `Player 0` gains 1 point.

example #2. Both `Player 0` and `Player 1` pick `SCISSORS`. Neither score is incremented.

After 10 rounds, the game ends and all processes **must** gracefully reach the end of their execution (i.e. no process should be forever blocked on an instruction which is not executable).

Model the *Rock-Paper-Scissors* game in *Promela* and find –using *Spin*– an execution trace in which the two friends split the bill equally.

(optional). Make appropriate use of `printf()` so that one (your teacher!) can easily follow the moves played by the two friends at each round and see the final score table of the game.

Solution:

```
mtype = { ROCK, PAPER , SCISSORS };
byte score[2];

inline print_hand(v)
{
    if
        :: v == PAPER    -> printf("paper   ");
        :: v == ROCK     -> printf("rock    ");
        :: v == SCISSORS -> printf("scissors");
    fi;
};

proctype player (chan chin, chou)
{
    mtype hand;
    byte cc;
    do
        :: chin?1 -> skip;
        :: chin?2 -> skip;
        :: chin?3 -> cc = cc + 1;
            if
                :: true      -> hand = ROCK;
                :: true      -> hand = PAPER;
                :: true      -> hand = SCISSORS;
                :: 2 == _pid -> hand = ROCK;
                :: 1 == _pid -> hand = PAPER;
                :: 2 == _pid -> hand = SCISSORS;
            fi;
            chou!hand;
        :: 10 == cc ->
            break;
    od;
}

active proctype waiter() {
    chan chou[2] = [1] of { byte };
    chan chin[2] = [1] of { mtype };
    mtype hand[2];
    byte i;

    run player(chou[0], chin[0]);
    run player(chou[1], chin[1]);

    printf("\n\t\t   Player 0        |\tPlayer 1\n");

    for (i: 0 .. 9) {
        chou[0]!1; chou[1]!1;
        chou[0]!2; chou[1]!2;
        chou[0]!3; chou[1]!3;
```

```
        chin[0]?hand[0]; chin[1]?hand[1];

        printf("Round %d: ", i);
        print_hand(hand[0]);
        printf("|");
        print_hand(hand[1]);

        if
            :: hand[0] == hand[1] ->
                printf(": tie\n");
            :: hand[0] % 3 == ((hand[1] + 2) % 3) ->
                printf(": 0 wins\n");
                score[0] = score[0] + 1;
            :: else ->
                printf(": 1 wins\n");
                score[1] = score[1] + 1;
        fi;
    }

    printf("\n      Summary: %d vs %d\n\n", score[0], score[1]);
};

ltl p0 { ! <> [] (score[0] == score[1]) };

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

2 nuXmv

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

```
    def mode(arr, ret):
        fr = [0, 0, 0]
        cc = 0
L0:     while (cc < 5):
L1:         fr[arr[cc]] = fr[arr[cc]] + 1
L2:         cc = cc + 1
L3:     max = fr[1]
        ret = 1
        cc = 1
L4:     while (cc <= 3):
L5:         if (fr[cc] > max):
L6:             ret = cc
                max = fr[cc]
L7:         cc = cc + 1
L8:     return                      # self-loop here!
```

Declare, inside the **main** module, the following variables:

- `arr`, an array of 5 elements with domain in $[1, 3]$, initialized to { `3, 1, 3, 1, 3` }
- `ret`, a variable with domain in $[1, 3]$
- `me` is an instance of `mode(arr, ret)` taking as input `arr` and `ret`

Hints:

- provide an appropriate initial value for `max` and `ret`, so as to reduce the number of viable initial states

Verify that the following properties are `true`:

- In the final state the mode is equal to 3
- Sooner or later, the value of `cc` will remain equal to 4 forever
- Invariant: the value of `max` is always smaller than 4

Solution:

```
MODULE main()
VAR
    arr : array 0..4 of 1..3;
    ret : 1..3;
    me  : mode(arr, ret);

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

MODULE mode(arr, ret)
VAR
    fr  : array 1..3 of 0..5;
    cc  : 0..5;
    max : 0..5;
    pc  : { L0, L1, L2, L3, L4, L5, L6, L7, L8 };

INIT
    pc = L0 & cc = 0 & max = 0 & ret = 1 &
    fr[1] = 0 & fr[2] = 0 & fr[3] = 0;

ASSIGN
    next(pc) := case
        L0 = pc & cc < 5           : L1;
        L0 = pc                    : L3;
        L1 = pc                    : L2;
        L2 = pc                    : L0;
        L3 = pc | L7 = pc          : L4;
        L4 = pc & cc <= 3          : L5;
        L4 = pc                    : L8;
        L5 = pc & 1 <= cc &
             cc <= 3 & fr[cc] > max : L6;
        L5 = pc | L6 = pc          : L7;
        TRUE                       : L8;
    esac;

    next(cc) := case
        (L2 = pc | L7 = pc) & cc < 5 : cc + 1;
        L3 = pc                      : 1;
        TRUE                         : cc;
    esac;

    next(max) := case
        L3 = pc                      : fr[1];
        L6 = pc & 1 <= cc & cc <= 3 : fr[cc];
        TRUE                         : max;
    esac;

    next(ret) := case
        L3 = pc                      : 1;
        L6 = pc & 1 <= cc & cc <= 3 : cc;
        TRUE                         : ret;
    esac;
```

```
TRANS (pc != L1 | (cc < 5 & arr[cc] != 1)) -> next(fr[1]) = fr[1];
TRANS (pc != L1 | (cc < 5 & arr[cc] != 2)) -> next(fr[2]) = fr[2];
TRANS (pc != L1 | (cc < 5 & arr[cc] != 3)) -> next(fr[3]) = fr[3];

TRANS (pc = L1 & (cc < 5 & arr[cc] = 1)) -> next(fr[1]) = fr[1] + 1;
TRANS (pc = L1 & (cc < 5 & arr[cc] = 2)) -> next(fr[2]) = fr[2] + 1;
TRANS (pc = L1 & (cc < 5 & arr[cc] = 3)) -> next(fr[3]) = fr[3] + 1;

-- In the final state the mode is equal to 3
LTLSPEC G (pc = L8 -> ret = 3);

-- Sooner or later, the value of cc will remain equal to 4 forever
LTLSPEC F G (cc = 4);

-- Invariant: the value of max is always smaller than 4
INVARSPEC max <= 3;
```