Course "Formal Methods"
Lab Test

Roberto Sebastiani
DISI, Università di Trento, Italy

May 26^{th}, 2017

769857918

[COPY WITH SOLUTIONS]

1 Spin

Model the *Cigarette Smokers* problem **using** the following specification.

Assume that a cigarette requires three ingredients to be made: `TOBACCO`, `PAPER` and `MATCHES`. There are three smokers around a table, each of which has an infinite supply of only **one** ingredient.

Smoker. Each smoker is in a loop waiting for both of his missing ingredients to be put on the *table*. Whenever that happens, he grabs the two ingredients from the table (which becomes empty), rolls a cigarette and smokes it by printing a message. The smoker must also put on the table one unit of his own resource whenever asked to do so through a channel.

Master Agent. Whenever the table is empty, the *master agent* sends a message demanding a unit of resource to be put on the table to two distinct *smokers* using a channel.

Simulate the system and visually verify that it behaves correctly.

Solution:

```
#define TOBACCO 1
#define PAPER   2
#define MATCHES 4

unsigned table : 3 = 0;

inline smoke()
{
    if
      :: id & TOBACCO -> printf("Tobacco Smokes\n");
      :: id & PAPER   -> printf("Paper Smokes\n");
      :: id & MATCHES -> printf("Matches Smokes\n");
    fi
    table = 0;
};

proctype smoker(mtype id; chan master)
{
  do
    :: atomic {
        master?eval(id) ->                          assert(~(id & table));
        table = (table|id);                         assert(id&table);
       }
    :: table == ((TOBACCO|PAPER|MATCHES) & ~id) ->
        smoke();                                    assert(~table);
  od
}

init
{
  unsigned i : 2 = 0; unsigned j : 2 = 0;
  chan master = [0] of { mtype };

  run smoker(TOBACCO, master);
  run smoker(PAPER, master);
  run smoker(MATCHES, master);

  do
    :: table == 0 ->
        select(i: 0..2); select(j: 0..2);

        if
          :: i == j -> j = (j + 1) % 3;
          :: i == j -> j = (j + 2) % 3;
          :: else -> skip;
        fi;

        master!(1<<i); master!(1<<j);
  od;
}
```

2 nuXmv

Model a battery powered *flying drone*, using the skeleton file `/usr/local/docs/drone.smv`, having the following state variables:

- `state`: can be either `OK` or `FAILURE`, the latter meaning that the drone cannot fly any longer
- `power`: ranges from 0 to 100, measures the remaining charge of the drone's battery
- `x`, `y`, `z`: discrete coordinates of the drone; `x` and `y` range in $[-30, 30]$, `z` ranges in $[0, 30]$
- `vx`, `vy`, `vz`: drone's speed vector; `vx` and `vy` range in $[-1, 1]$, `vz` ranges in $[-2, 1]$

Initially, `state` is `OK`, the battery is fully charged and all other variables are equal to 0.

`state`. Assume that the drone is flying in a room surrounded by a concrete wall in all directions, so that if the drone is in the immediate proximity of a wall with a positive speed in the direction of the wall then it will crash against it. The `state` variable of the drone changes to `FAILURE` whenever in the next state of the execution trace the `drone` crashes against a wall. Otherwise, the `state` variable keeps its value.

`power`. The battery is charged back to full state whenever the drone parks at the origin and is in `OK` state. The drone consumes one unit of power when touches the ground but has vertical speed larger than 0. It also consumes one unit of power when it is flying mid-air with a vertical speed larger than -2 (i.e. larger than *free fall* speed). Otherwise, the power level remains unchanged.

`position`. The position of the robot in the next state is obtained by adding its old position vector (x, y, z) with its velocity vector (vx, vy, vz).

`vx`. (resp. `vy`) changes according to this **sorted** set of rules:

- if the drone has crashed over the `x` axis (resp. `y`) or in the next state touches the ground, `vx` (resp. `vy`) is set to 0.
- if the drone is falling, the speed `vx` (resp. `vy`) does not change
- if the drone is powered, it can freely change by one single unit value its current speed
- otherwise, the speed is 0

Express the following properties, and check their expected value with nuXmv:

- **LTL.** if the drone always flies safe then it will remain in good state forever. (true)
- **LTL.** if the drone is flying above ground-level infinitely often, then the drone charges infinitely often too. (true)
- **CTL.** if the drone experiences a collision, then it is necessarily the case that it will eventually hit the ground with negative speed. (true)
- **CTL.** regardless of its position, if the drone is mid-air in `OK` state and has at least 7% of its battery left, then it has at least one possible safe landing strategy. (true)
- **CTL.** write a property s.t. its counter-example is a safe landing strategy for the drone in the state $(0, 0, 30, 0, 0, 0, 5, OK)$ with $(x, y, z, vz, vy, vz, power, state)$.
- **CTL.** write a property s.t. its counter-example is a flight plan that goes through the ordered sequence of states $s_1 = (30, 0, 0, 0, 0, 0, OK)$, $s_2 = (30, 30, 0, 0, 0, 0, OK)$, $s_3 = (30, 30, 30, 0, 0, 0, OK)$ and $s_4 = (20, 30, 30, 1, 0, 0, OK)$ with $(x, y, z, vx, vy, vz, state)$.
- **BONUS.** use `pick_state -s N.NNN` to jump at the last state in the counter-example found for the previous property, and simulate the system with `simulate -iv -k 30`. What happens to the drone? What is the final position of the drone at the end of the simulation?

Solution:

```
MODULE main ()
VAR
  state: { OK, FAILURE }; power: 0..100;
   x: -30..30;  y: -30..30;  z: 0..30;
  vx: -1..1;   vy: -1..1;   vz: -2..1;

...

ASSIGN
  init(x)  := 0; init(y)  := 0; init(z)  := 0;
  init(vx) := 0; init(vy) := 0; init(vz) := 0;
  init(power) := 100; init(state) := OK;

  next(state) := case
      next(collision) : FAILURE;
      TRUE            : state;
    esac;

  next(power) := case
      x = 0 & y = 0 & z = 0 &
      vz = 0 & state = OK     : 100;
      power = 0               : 0;
      touch_ground  & vz > 0  : power - 1;
      !touch_ground & vz > -2 : power - 1;
      TRUE                    : power;
    esac;

  next(x) := case
      -30 <= (x + vx) & (x + vx) <= 30 : x + vx;
      x + vx < -30                     : -30;
      x + vx > 30                      : 30;
    esac;
  next(y) := case
      -30 <= (y + vy) & (y + vy) <= 30 : y + vy;
      (y + vy) < -30                   : -30;
      (y + vy) > 30                    : 30;
    esac;
  next(z) := case
        0 <= (z + vz) & (z + vz) <= 30 : z + vz;
      (z + vz) < 0                     : 0;
      (z + vz) > 30                    : 30;
    esac;

  next(vx) := case
      crash_x | next(touch_ground) : 0;
      is_falling                   : vx;
      has_power & vx = 1           : { vx, vx - 1 };
      has_power & vx = -1          : { vx, vx + 1 };
      has_power                    : { vx, vx + 1, vx - 1 };
      TRUE                         : 0;
    esac;
```

```
  next(vy) := case
      crash_y | next(touch_ground) : 0;
      is_falling                   : vy;
      has_power & vy = 1           : { vy, vy - 1 };
      has_power & vy = -1          : { vy, vy + 1 };
      has_power                    : { vy, vy + 1, vy - 1 };
      TRUE                         : 0;
    esac;
  next(vz) := ...

...

-- if the drone always flies safe then it will remain in good state forever [LTL, TRUE]
LTLSPEC G (safe_flight) -> G state = OK;

-- if the drone is flying above ground-level infinitely often, then the drone charges
-- infinitely often too. [LTL, TRUE]
LTLSPEC G F (!touch_ground) -> G F (power = 100);

-- if the drone experiences a collision, then it is necessarily the case that it will
-- eventually hit the ground with negative speed [CTL, TRUE]
CTLSPEC AG (collision -> AF (z = 0 & vz < 0));

-- regardless of its position, if the drone is mid-air in OK state and has at least 7% of
-- its battery left, then it has at least one possible safe landing strategy. [CTL, TRUE]
CTLSPEC AG ((state = OK & power >= 7 & z > 0) -> EF good_parking);

-- write a property s.t. its counter-example is a safe landing strategy for the drone in
-- the state (0, 0, 30, 0, 0, 0, 5, OK) with (x, y, z, vx, vy, vz, power, state) [CTL]
CTLSPEC AG ((x = 0 & y = 0 & z = 30 & vx = 0 & vy = 0 & vz = 0 & power = 5 & state = OK)
            -> AF (state = FAILURE));

-- write a CTL property s.t. its counter-example is a flight plan that goes through the
-- ordered sequence of states s1, s2, s3, s4 (see above definitions) [CTL]
CTLSPEC !( EF ( s1 & EF (s2 & EF (s3 & EF s4)))) ;
-- or, equivalently,
CTLSPEC ! E [ TRUE U (s1 &
          E [ TRUE U (s2 &
          E [ TRUE U (s3 &
          E [ TRUE U  s4 ] ) ] ) ] ) ];

-- BONUS: use 'pick_state -s N.NNN' to jump at the last state in the counter-example
-- found for the previous property, and simulate the system with 'simulate -iv -k 30'.
-- - What happens to the drone?
-- It falls due to lack of power, eventually crashing against the X wall
-- - What is the final position of the drone at the end of the simulation?
-- (30, 30, 0)
```