Technische Universität München (I7) **Summer Semester 2019**
J. Esparza / P. Ashok / T. Meggendorfer **27.06.2019**

Model Checking – Exercise sheet 8

Exercise 8.1

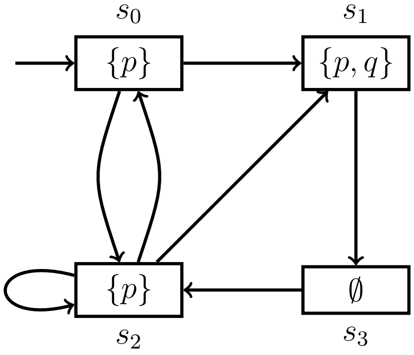
Consider an elevator system that services $N > 0$ floors numbered 0 through $N-1$. There is an elevator door at each floor with a call button and an indicator light that signals whether or not the elevator has been called. In the elevator cabin there are N send buttons (one per floor) and N indicator lights that inform to which floor(s) is going to be sent. For simplicity consider $N = 4$. Present a set of atomic propositions (try to minimize the number of propositions) that are needed to describe the following properties of the elevator system as CTL formulae and give the corresponding CTL formulae

1. The doors are "safe", i.e., a floor door is never open if the cabin is not present at the given floor.
2. The indicator lights correctly reflect the current requests. That is, each time a button is pressed, there is a corresponding request that needs to be memorized until fulfillment (if ever).
3. The elevator only services the requested floors and does not move when there is no request.
4. All requests are eventually satisfied.

(This above exercise is taken from 'Principles of Model Checking')

Exercise 8.2

Create a `NuSMV` model for the following Kripke structure over $AP = \{p, q\}$:



Use `NuSMV` to model check each of the following formulas. Explain in words if the formula holds, or give a counterexample otherwise.

(a) $\mathbf{EG}\, p$,

(b) $\mathbf{AX}\, \mathbf{AF}\, \mathbf{EG}\, p$,

(c) $p\, \mathbf{AU}\, q$,

(d) $\mathbf{AG}(p \to \mathbf{AX}\, p)$,

(e) $\mathbf{EX}(\neg q \land (\neg p\, \mathbf{EU}\, q))$.

Exercise 8.3

Model the following stack system in `NuSMV`:

> The stack system consists of three input interfaces: `push`, `pop`, `in_val`; and one output interface: `out_val`. The values of `push` and `pop` can be either `true` or `false`, while `in_val` and `out_val` can take any number between 0 and 9.
>
> When `push` is `true`, the system takes the input from `in_val` and pushes it onto its internal stack. When `pop` is `true`, the system removes the value on the top of the stack and outputs it via `out_val`. It is forbidden to call `push` and `pop` at the same time. The size of the stack is 5, i.e. the stack is full if there are 5 pushes without a pop. When the stack is full, it ignores `push` and `in_val`. Similarly, the system ignores `pop` when the stack is empty. The value of `out_val` is undefined if the stack is empty or `pop` is `false`.

Write the following properties in CTL and use `NuSMV` to model check the formulas:

(a) The stack cannot be empty and full at the same time.

(b) There exists a path along which the stack is eventually always full.

(c) From any given point of time, there always exists a path in which the stack will be full.

(d) The stack cannot be empty after a push.

(e) The internal stack is correctly updated after a push or pop.

(f) Whenever the stack is full, there exists a path in which the stack stays full forever or it remains full until a pop.

(g) For every push, there exists a path that pops the value without pushing another value.

(h) After every pop, `out_val` holds the correct value.

Solution 8.2

```
MODULE main
VAR
  state : {s0, s1, s2, s3};
ASSIGN
  init(state) := s0;
  next(state) :=
    case
      state = s0 : {s1, s2};
      state = s1 : s3;
      state = s2 : {s0, s1, s2};
      state = s3 : s2;
    esac;
DEFINE
  p := state = s0 | state = s1 | state = s2;
  q := state = s1;

SPEC
  EG p
SPEC
  AX AF EG p
SPEC
  A [p U q]
SPEC
  AG (p -> AX p)
SPEC
  EX (!q & E [!p U q])
```

Solution 8.3

```
MODULE main
VAR
  op : 0..2;
  in_val  : 0..9;
  out_val : 0..9;
  ptr : 0..5;
  arr : array 0..4 of 0..9;

FROZENVAR
  i : 0..4;
  x : 0..9;

DEFINE
  empty := (ptr = 0);
  full  := (ptr = 5);
  push  := (op = 0);
```

```
  pop   := (op = 1);

ASSIGN
  init(ptr) := 0;
  next(ptr) := case
                 push & !full  : ptr + 1;
                 pop  & !empty : ptr - 1;
                 TRUE : ptr;
               esac;

  next(arr[0]) := push & ptr = 0 ? in_val : arr[0];
  next(arr[1]) := push & ptr = 1 ? in_val : arr[1];
  next(arr[2]) := push & ptr = 2 ? in_val : arr[2];
  next(arr[3]) := push & ptr = 3 ? in_val : arr[3];
  next(arr[4]) := push & ptr = 4 ? in_val : arr[4];

  next(out_val) := case
                     pop & !empty : arr[ptr - 1];
                     TRUE : out_val;
                   esac;

--  (a) The stack cannot be empty and full at the same time.
SPEC
  AG !(empty & full)

-- (b) There exists a path along which the stack is eventually always full.
SPEC
  EF EG full

-- (c) From any given point of time, there always exists a path in
-- which the stack will be full.
SPEC
  AG EF full

-- (d) The stack cannot be empty after a push.
SPEC
  AG (push -> AX !empty)

-- (e) The internal stack is correctly updated after a push or a pop.
SPEC
  AG ((push & !full & in_val = x & ptr = i) -> (AX (arr[i] = x)))

SPEC
  AG ((push & !full & ptr = i) -> (AX (ptr = i + 1)))
```

```
SPEC
  AG ((pop & !empty & ptr = i) -> (AX (ptr = i - 1)))

SPEC
  AG ((push & ptr >= 4) -> (AX full))

SPEC
  AG ((pop & ptr <= 1) -> (AX empty))

-- (f) Whenever the stack is full, there exists a path in which the
-- stack stays full forever or it remains full until a pop.
SPEC
  AG (full -> ((EG full) | E[full U pop]))

-- (g) For every push, there exists a path that pops the value without
-- pushing another value.
SPEC
  AG (push -> EX E[!push U pop])

-- (h) After every pop, out_val holds the correct value
SPEC
  AG ((pop & !empty & arr[ptr - 1] = x) -> (AX (out_val = x)))
```