1 Exercise Sheet - Problems

There are three exercises, yielding a total of 15 points.

Exercise 1. (2+2 points) Consider the following NuSMV program implementing mutual exclusion between two processes.

```
MODULE main
VAR
   semaphore : boolean;
   proc1 : process user(semaphore);
   proc2 : process user(semaphore);
ASSIGN
   init(semaphore) := FALSE;

MODULE user(semaphore)
VAR
   state : {idle, entering, critical, exiting};
ASSIGN
   init(state) := idle;
   next(state) :=
               case
                   state = idle                  : {idle, entering};
                   state = entering & !semaphore : critical;
                   state = critical              : {critical, exiting};
                   state = exiting               : idle;
                   TRUE                          : state;
               esac;
   next(semaphore) :=
               case
                   state = entering : TRUE;
                   state = exiting  : FALSE;
                   TRUE             : semaphore;
               esac;
FAIRNESS
   running
```

This NuSMV program uses the variable `semaphore` to implement mutual exclusion between the two processes `proc1` and `proc2`. Each process has four states: `idle`, `entering`, `critical` and `exiting`. The `entering` state indicates that the process wants to enter its critical region. If the variable `semaphore` is `FALSE`, it goes to the `critical` state, and sets `semaphore` to `TRUE`. On exiting its critical region, the process sets `semaphore` to `FALSE` again.

(a) A safety property P of this program is that "it should never be the case that the two processes `proc1` and `proc2` are at the same time in the `critical` state". Express P as a *CTL formula* and add it as a CTL specification to the above NuSMV program. Verify the CTL specification by running NuSMV on the annotated program. If NuSMV produces a counterexample, explain the counterexample!

```
SPEC AG ! (proc1.state = critical & proc2.state = critical)
```

(b) A liveness property Q of this program is that "whenever process `proc2` wants to enter its `critical` state, it eventually does". Express Q as an *LTL formula* and add it as a LTL specification to the above NuSMV program. Verify the LTL specification by running NuSMV on the annotated program. If NuSMV produces a counterexample, explain the counterexample!

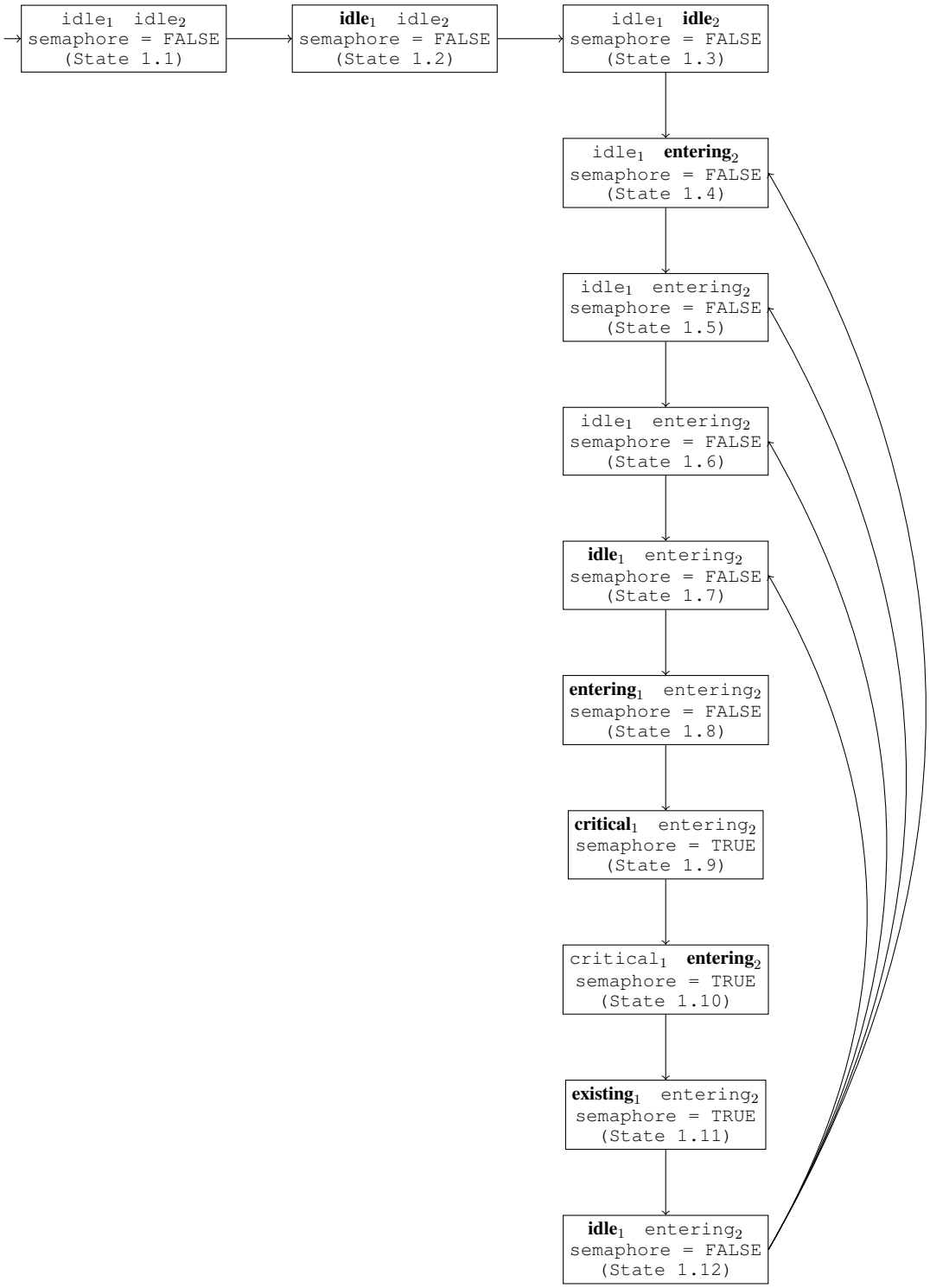
```
LTLSPEC G (proc2.state = entering -> F proc2.state = critical)
```

Counterexample. This liveness property does not hold for the considered system, and NuSMV returns the following counterexample.

```
-> State: 1.1 <-
  semaphore = FALSE
  proc1.state = idle
  proc2.state = idle
-> Input: 1.2 <-
  _process_selector_ = proc1
  running = FALSE
  proc2.running = FALSE
  proc1.running = TRUE
-> State: 1.2 <-
-- Nondeterministically set next(proc1.state) = idle.
-- So, variables semaphore, proc1.state, proce2.state in both
-- states 1.1 and 1.2 have the same values.
-> Input: 1.3 <-
  _process_selector_ = proc2
  proc2.running = TRUE
  proc1.running = FALSE
-> State: 1.3 <-
-- Nondeterministically set next(proc2.state) = idle.
-- So, variables semaphore, proc1.state, proce2.state in both
-- states 1.2 and 1.3 have the same values.
-> Input: 1.4 <-
-- Loop starts here
-> State: 1.4 <-
  proc2.state = entering
-> Input: 1.5 <-
-- The module main has always its own process associated. Hence,
-- there are 3 processes main, proc1, and proc2 in this example.
  _process_selector_ = main
  running = TRUE
  proc2.running = FALSE
-- Loop starts here
-> State: 1.5 <-
-> Input: 1.6 <-
-- Loop starts here
-> State: 1.6 <-
-- States 1.5 and 1.6 are the same
-> Input: 1.7 <-
  _process_selector_ = proc1
  running = FALSE
```

```
  proc1.running = TRUE
-- Loop starts here
-> State: 1.7 <-
-> Input: 1.8 <-
-> State: 1.8 <-
  proc1.state = entering
-> Input: 1.9 <-
-> State: 1.9 <-
  semaphore = TRUE
  proc1.state = critical
-> Input: 1.10 <-
  _process_selector_ = proc2
  proc2.running = TRUE
  proc1.running = FALSE
-> State: 1.10 <-
-> Input: 1.11 <-
  _process_selector_ = proc1
  proc2.running = FALSE
  proc1.running = TRUE
-> State: 1.11 <-
  proc1.state = exiting
-> Input: 1.12 <-
-> State: 1.12 <-
  semaphore = FALSE
  proc1.state = idle
```

The following diagram illustrates the considered counterexample. To keep the representation simple, we show only the values of variables `semaphore, proc1.state, proc2.state`, and use the sugar syntax $\texttt{val}_i$ for a constraint $\texttt{proc}i\texttt{.state} = val$ where $i \in \{1, 2\}$. Moreover, the bold text shows which process in `proc1` and `proc2` is running.



Intuitively, this counterexample has infinite loops where following constraints hold:

- Variable `proc2.state` always equals `entering`, i.e `proc2.state = entering`.
- Only process `proc1` enters the `critical` state. Therefore, only process `proc1` keeps and releases the semaphore.
- Whenever the fairness condition `running` is applied on process `proc2`, process `proc1` is already in the `critical` state. Therefore, the only possible transition for `proc2` is to keep `proc2.state = entering`.

– A loop can contain transitions from process `main` which do not change values of variables `semaphore, proc1, proc2`.

Submission guidelines: For each taks, submit the annotated NuSMV program together with the output of running NuSMV on it. In case a counterexample was generated, submit your narrative explanation of the counterexample.

Exercise 1.2. (2+2 points) Consider the following NuSMV program implementing a simple, deterministic counter modulo 8.

```
MODULE main
VAR
  y : 0..15;

ASSIGN
  init(y) := 0;

TRANS
  case
    y = 7    :  next(y) = 0;
    TRUE     :  next(y) = (y + 1) mod 16;
  esac
```

(a) Consider the property expressing that "there is a value of y whose next value is 8". Express this property as an LTL formula and add it as a LTL specification to the above NuSMV program. Verify the LTL specification by running NuSMV on the annotated program. If NuSMV produces a counterexample, explain the counterexample!

```
LTLSPEC F (X y = 8)
```

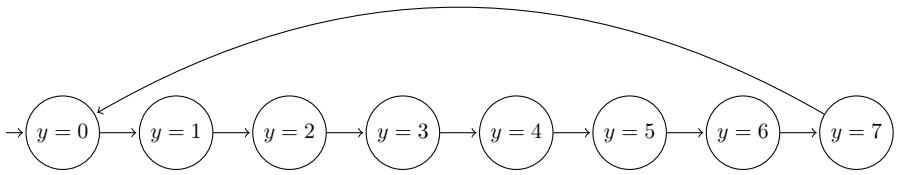
Counterexample. The liveness property does not hold for the considered system, and NuSMV returns the following counterexample

```
-- Loop starts here
-> State: 1.1 <-
  y = 0
-> State: 1.2 <-
  y = 1
-> State: 1.3 <-
  y = 2
-> State: 1.4 <-
  y = 3
-> State: 1.5 <-
  y = 4
-> State: 1.6 <-
  y = 5
-> State: 1.7 <-
  y = 6
-> State: 1.8 <-
  y = 7
-> State: 1.9 <-
  y = 0
```

The above execution path has an infinite loop such that:

- State 1.1 is an initial state where `y` is initialized to 0. Moreover, the infinite loop starts from this state.
- For every $1 \leq k < 8$, we have that the value of `y` in state $1.k$ is not equal to 7. It follows that the evaluation of the operator `case` returns the value of the expression `next(y) = (y + 1) mod 16`.
- Due to `y = 7` in state 1.8, the evaluation of the operator `case` returns the value of the expression `next(y) = 0`. The result is state 1.9 which is the same with state 1.1.

The following diagram illustrates the considered counterexample:



(b) Consider the property expressing that "there is a value of y whose next value is 7". Express this property as an LTL formula and add it as a LTL specification to the above NuSMV program. Verify the LTL specification by running NuSMV on the annotated program. If NuSMV produces a counterexample, explain the counterexample!

```
LTLSPEC F (X y = 7)
```

Submission guidelines: For each taks, submit the annotated NuSMV program together with the output of running NuSMV on it. In case a counterexample was generated, submit your narrative explanation of the counterexample.

Exercise 1.3. (1+4+2 points) Consider the following puzzle that is an instance of the puzzle known as the "Tower of Hanoi".

There are three poles (`left`, `middle`, `right`) and four ordered disks `d1`, `d2`, `d3`, `d4` of different sizes, with disk `d1` being the biggest one. Initially, all four disks are on the `left` pole in ascending order, the smallest at the top. The goal of the puzzle is to move all four disks to the `right` pole, using the following simple rules:

- Only one disk can be moved at a time;
- Each move consists of taking the upper disk from one of the poles and placing it on top of another pole;
- No disk may be placed on top of a smaller disk.

The NuSMV program below describes the skeleton of a Hanoi tower puzzle with four disks. The skeleton declares the state variables of the puzzle and defines macros for moving a disk.

```
MODULE main
-- Hanoi tower with three poles (left, middle, right), and four
-- ordered disks d1, d2, d3, and d4. Disk d1 is the biggest one.
VAR
  d1 : {left,middle,right};
  d2 : {left,middle,right};
  d3 : {left,middle,right};
  d4 : {left,middle,right};
  move : 1..4; -- possible moves
DEFINE
  move_d1 := move=1;
  move_d2 := move=2;
  move_d3 := move=3;
  move_d4 := move=4;

-- di is on top of a pole iff di!=dj for every j>i
-- These predicates d1, d2, d3, and d4 ensure that
--  (a) Only one disk is on top of every pole.
--  (b) If disk di is on top of pole pk, other disks on pole pk are
--      bigger than di.
  top_d1 :=
           d1 != d2 &
           d1 != d3 &
           d1 != d4;
  top_d2 :=
           d2 != d3 &
           d2 != d4;
  top_d3 :=
           d3 != d4;
  top_d4 := TRUE;
```

Complete the program skeleton above to model the puzzle, ensuring that the puzzle yields a solution (that is, all four disks are on the `right` pole). Your tasks are as follows:

(a) Declare the set of initial states;

```
INIT d1 = left & d2 = left & d3 = left & d4 = left;
```

(b) Formalize the transition relation for the existing variables;

(Hint: declare the transition relation by completing and continuing the following skeleton `TRANS move_d1 -> ...`)

```
TRANS
  -- Move only d4. This action is always possible since d4 is
  --  smallest.
   (move_d4 -> (top_d4          &
                next(d1) = d1   &
                next(d2) = d2   &
                next(d3) = d3   &
                next(d4) != d4))
```

```
 -- Move only d3. This movement must satisfy following
 --  constraints: (a)  top_d3 = TRUE, i.e. d3 is on top of
 --  a pole, and (b) only d3's position will be changed,
 --  and (c) next(d3) != d4, i.e. we do not move d3 to a
 --  pole on which d4 is already since d3 is bigger than d4,
 --  and we cannot leave d3 on d4.
& (move_d3 -> (top_d3         &
                next(d1) = d1  &
                next(d2) = d2  &
                next(d3) != d3 &
                next(d4) = d4  &
                next(d3) != d4))
& (move_d2 -> (top_d2         &
                next(d1) = d1  &
                next(d2) != d2 &
                next(d3) = d3  &
                next(d4) = d4  &
                next(d2) != d3 &
                next(d2) != d4))
& (move_d1 -> (top_d1         &
                next(d1) != d1 &
                next(d2) = d2  &
                next(d3) = d3  &
                next(d4) = d4  &
                next(d1) != d2 &
                next(d1) != d3 &
                next(d1) != d4)) ;
```

(c) Formalize in CTL the requirement that the puzzle has a solution and make sure that your design satisfies it.

```
CTLSPEC ! EF (d1=right & d2=right & d3=right & d4=right)
```

Submission guidelines: Submit the NuSMV program completed with initial states and transition relation and annotated with the proper CTL specification. Submit and explain the output of running NuSMV on your solution.