Validation des systèmes embarqués : Model Checking

TD4 : NuSMV

Nga Nguyen

Question 1 : Implémenter les exemples vus en cours (compteur modulo 4 avec reset, three-bit counter, cell (red, green, blue), inverter ring).

Question 2 : Deux processus veulent accéder à une ressource commune et ils ont 3 états : *idle, trying, critical.* L'état *trying* indique que le processus veut entrer en section critique. La variable booléenne *turn* est utilisée pour implémenter l'exclusion mutuelle. Compléter le code NuSMV suivant pour modéliser le système :

```
MODULE main
VAR
        p1: {idle, trying, critical};
        p2: {idle, trying, critical};
        turn: {1, 2};
ASSIGN
        init(p1) := …
        next(p1) := …
```

Puis vérifier les 2 spécifications suivantes :

1) **Exclusion mutuelle** (safety): à un moment donné, un seul processus peut être dans la section critique.
2) **Pas de famine** (liveness): si un processus veut entrer en section critique, il va finir par y entrer.

Question 3 : Le programme suivant utilise un sémaphore pour implémenter l'exclusion mutuelle entre deux processus asynchrones. Chaque processus a 4 états : *idle, entering, critical* et *exiting*. L'état *entering* indique que le processus veut entrer en section critique. Si le sémaphore est False, il rentre en état *critical*, et met le sémaphore à True. En quittant la section critique (*exiting*), le processus remet le sémaphore à False.

```
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;
```

Exprimer puis vérifier les spécifications suivantes :

a) **Exclusion mutuelle** (safety): à un moment donné, un seul processus peut être dans la section critique.

b) **Pas de famine** (liveness): Si un processus veut entrer en section critique, il va finir par y entrer. Qu'est-ce qu'il faut ajouter pour que la propriété soit vraie ?

Question 4 : (*Source : Anton Hedin, UPPSALA UNIVERSITET*) Consider the following situation: Adam and Eve are sitting at a table to enjoy a nice meal. They each have a plate of food, but there is only one fork at the table. Thus only one of them can be eating at any given time. To make sure that both will be able to finish their meal there is a waiter that controls access to the fork. Adam and Eve may at any time request to use the fork, and if it is available the person requesting it should have it. This situation can be modeled by the following code. The main module represents the waiter and determines whose turn it is, based on received requests. Your first task is to fill in the gaps in the code.

```
MODULE proc(other-person, turn, myname)
   VAR
        status: {no-fork, request-fork, eating};
  ASSIGN
        init(status) := ….;   -- complete here
        next(status) := case
            (status = request-fork) & (other-person=request-fork) & (turn=myname) : eating;
            (status = request-fork) &   -- complete here
            (status = no-fork)   -- complete here
            (status = eating) : {eating,no-fork};
            TRUE : status;
        esac;

   FAIRNESS running
```

```
MODULE main
  VAR
      Adam : process proc(Eve.status,turn,FALSE);
      Eve : process proc(Adam.status,turn,TRUE);
      turn : boolean;
  ASSIGN
      init(turn) := FALSE;
      -- You can leave the evolution empty:
      -- in this way turn changes randomly.
```

Now load the model and formalize and check the following properties:

1. Adam and Eve are never able to eat at the same time (safety);
2. If Adam (Eve) requests the fork he (she) should eventually be allowed to eat (liveness).

A fairness constraint restricts the attention only to fair execution paths. When evaluating specifications, the model checker considers path quantifiers to apply only to fair paths. The fairness condition running restricts attention to paths along which the module in which it appears is selected for execution infinitely often. Try to find a fairness condition making the second formula above true.