TP1 Promela

1. Algorithme d'exclusion mutuelle de Dekker (vu en cours)

```
#define true 1
#define false 0
#define Aturn false
#define Bturn true

bool x, y, t;

proctype A()
{       x = true;
        t = Bturn;
        (y == false || t == Aturn);
        /* critical section */
        x = false
}

proctype B()
{       y = true;
        t = Aturn;
        (x == false || t == Bturn);
        /* critical section */
        y = false
}

init { run A(); run B() }
```

Coder en `Promela` l'algorithme de Dekker. Vérifier à l'aide du mécanisme «assert» que cet algorithme assure bien l'exclusion mutuelle.

2. Sémaphore de Dijkstra (vu en cours)

```
#define p 0
#define v 1

chan sema = [0] of {bit};

proctype dijkstra()
{
 do
 :: sema!p ->
    sema?v;
 od
}

proctype user()
{
 do
 :: sema?p ->
    skip; /* critical section */
    sema!v;
    skip; /* non critical section */
 od
}

init{ atomic { run dijkstra();
               run user(); run user(); run user()} }
```

Coder en `Promela` le sémaphore de Dijkstra. Vérifier à l'aide du mécanisme «assert» que cet algorithme assure bien l'exclusion mutuelle.

3. Boucles en Promela

a) Ecrire en `Promela` un processus qui prend en paramètre un entier `n`, puis affiche la somme des `n` premiers entiers, en émulant le comportement de la boucle «`while`».

b) Ecrire en `Promela` un processus qui affiche la valeur maximum d'un tableau de `n` entiers.

4. Nombre aléatoire

a) La conditionnelle en `Promela` étant non déterministe, servez-vous en pour écrire une fonction (ou un processus) générant un nombre aléatoire (soit `0` soit `1`).

b) Ecrire un processus générant un nombre aléatoire à `n` bits.

5. Jeu de Nim (allumettes)

Ce jeu se joue à deux. Les joueurs sont initialement devant un certain nombre d'allumettes `N` (qui peut varier d'une partie à l'autre). A tour de rôle, chaque joueur doit enlever `1`, `2` ou `3` allumettes.
On supposera ici un tirage aléatoire du nombre d'allumettes a retirer.
Le joueur qui retire la dernière allumette gagne.

a) Modéliser le fonctionnement du jeu et lancer une simulation.

b) Vérifier les propriétés suivantes :
   i) le nombre d'allumettes retiré + le nombre d'allumettes restant = `N` (invariant)
   ii) à la fin, le nombre d'allumettes restant est égal à `0`.

6. Arbre binaire équilibré

Définir un code `Promela` permettant de générer un arbre binaire équilibré de profondeur `n`.

Chaque nœud a un canal parent, pouvant recevoir le message « `create` » demandant de créer deux nœuds feuilles et sur lequel il peut écrire un message « `created` » indiquant que les nœuds feuilles ont été créées.

Chaque nœud non feuille a deux nœuds fils (gauche et droit) avec lesquels il peut communiquer par message.

7. Réseau de Petri en Promela

Les réseaux de Petri se modélisent de façon simple en `Promela`. On considère le réseau de Petri suivant :

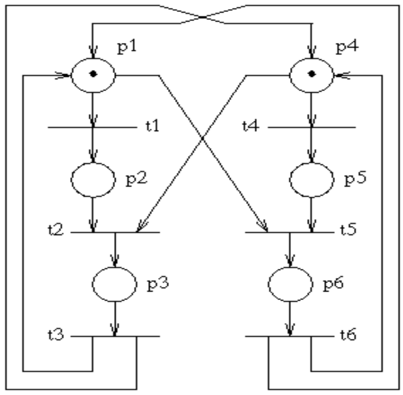

Figure 5 — Petri Net with Hang State

```
#define Place    byte  // assume at most 255 tokens per place
                           // file ex_4.pml
Place p1=1, p2, p3         // place p1 has an initial token
Place p4=1, p5, p6         // place p4 has an initial token

#define inp1(x)  x>0 -> x--
#define inp2(x,y)      x>0 && y>0 -> x--; y--
#define out1(x)  x++
#define out2(x,y)      x++; y++

init
{
  do
  :: inp1(p1)    -> out1(p2) // t1
  :: inp2(p2,p4) -> out1(p3) // t2
  :: inp1(p3)    -> out2(p1,p4)    // t3
  :: inp1(p4)    -> out1(p5) // t4
  :: inp2(p1,p5) -> out1(p6) // t5
  :: inp1(p6)    -> out2(p4,p1)    // t6
  od
}
```

Coder ce programme, puis lancer une vérification. Commenter le résultat obtenu.