Algorithmes distribués – TP noté

Contrôle Continu n°2
Les deux exercices sont indépendants
Durée : 1h40

Consignes :

- les programmes devront être déposés sur moodle sous forme de fichiers séparés, exécutables en Promela. Ils seront préfixés par le numéro de l'exercice (ex : *exo1a.pml*).
- **pour chaque programme, indiquer en commentaire si la vérification syntaxique fonctionne, et le cas échéant, si la vérification est correcte.**

Exercice 1 : distributeur de sucreries (10 points)

Considérons un distributeur de sucreries contenant des barres chocolatées (cout : 0.50€) et des paquets de bonbon (cout : 1 €). Chaque fois qu'un utilisateur introduit le montant exact dans la machine, il obtient la sucrerie demandée.

a) Définir un processus nommé `distributeur` modélisant le comportement du distributeur de sucrerie. Définir aussi un processus `client` représentant un utilisateur du distributeur. Cet utilisateur consommera en continu des sucreries (aussi bien des bonbons que des chocolats). Dans un premier temps, on supposera que le distributeur dispose d'une infinité de sucrerie et que le client a à sa disposition une somme d'agent illimitée.

Vous pouvez considérer la structure de données suivante :

```
mtype = { ... };        /*    types des messages transitant sur les canaux
                              argent_channel et  sucrerie_channel */
chan argent_channel = [1] of { mtype };
chan sucrerie_channel = [1] of { mtype };

proctype client() {
 ...
}

proctype distributeur() {
 ...
}

init { atomic { run client(); run distributeur(); } }
```

b) Modifier l'exercice précédent de telle sorte que le client n'ait à sa disposition qu'un budget de `5` €, et que le nombre de chocolats et de bonbons du distributeur soit limité respectivement à `10` et `5`. Définir une assertion permettant de s'assurer que la valeur des «biens» du client est toujours égale à `5` € (la valeur des «biens» correspondant au budget restant, cumulé avec le montant équivalent aux sucreries achetées). Pour cela il faudra enregistrer le montant équivalent aux sucreries, ainsi que le nombre de chocolats et de bonbons que le client a achetés.

c) Tester le protocole en vérification. Rajouter si besoin des labels pour que la vérification n'indique pas d'erreur. Faire le scénario de simulation aléatoire.

Exercice 2 : attribution de rôles dynamique et indéterministe (10 points)

L'objectif est d'affecter *dynamiquement* un rôle différent à chaque processus d'un ensemble, sachant que l'ensemble des rôles à attribuer a le même cardinal que celui des processus qui s'exécutent. Cette affectation doit être *indéterministe*, et ne doit pas dépendre d'un paramètre du processus tel que son numéro.

a) On considère ici un ensemble de deux processus, qui doivent s'affecter dynamiquement le rôle MAITRE ou ESCLAVE (à la fin de l'exécution un seul processus a le rôle MAITRE et un seul processus a le rôle ESCLAVE). Définir un processus nommé `affecte_role` modélisant le comportement du processus.

Vous pouvez utiliser la structure de données suivante :

```
#define MAITRE 1
#define ESCLAVE 2

proctype affecte_role (…) {
 ...
}

init
{chan AtoB = [1] of {byte} ;
 chan BtoA = [1] of {byte} ;

atomic {
      run affecte_role(AtoB,BtoA) ;
      run affecte_role(BtoA,AtoB) ;
}}
```

b) Rajouter un mécanisme permettant de vérifier qu'à la fin de l'exécution les rôles attribués aux deux processus sont bien différents..

c) Etendre l'exercice à un ensemble de trois processus qui s'affectent dynamiquement un rôle unique parmi l'ensemble des 3 rôles {SERVEUR_PRIMAIRE, SERVEUR_SECONDAIRE, CLIENT}.

d) Rajouter un mécanisme permettant de vérifier qu'à la fin de l'exécution les rôles attribués aux trois processus sont bien différents.

CORRIGE

Exercice 1

a)

```
mtype = { euro1, cents50, chocolat, bonbon };
chan argent_channel = [1] of { mtype };
chan sucrerie_channel = [1] of { mtype };

proctype client() {
     do
     :: argent_channel!cents50 -> sucrerie_channel?chocolat;
     :: argent_channel! euro1 ->  sucrerie_channel?bonbon;
     od
}
proctype distributeur() {
     do
     :: argent_channel? cents50 -> sucrerie_channel!chocolat;
     :: argent_channel? euro1 -> sucrerie_channel!bonbon;
     od
}
init { atomic { run client(); run distributeur(); }}
```

b)

```
mtype = { chocolat, bonbon };
chan argent_channel = [1] of { short };
chan sucrerie_channel = [1] of { mtype };
short budget = 500, argent_dis = 0;

proctype client() {
end0:     do
     ::      budget>=      50      ->      argent_channel!50      ->
sucrerie_channel?chocolat;
     ::      budget>=      100      ->      argent_channel!100      ->
sucrerie_channel?bonbon;
     :: budget<50 -> break
     od
}
proctype distributeur() {
     byte chocolats = 10, bonbons = 5;
     assert(budget+argent_dis == 500);

end1:     do
     ::    ((chocolats    >    0)    &&    argent_channel?[50])    ->
argent_channel?50;
          argent_dis = argent_dis + 50;
          budget=budget-50;
          sucrerie_channel!chocolat;
          chocolats = chocolats-1;
          assert(budget+argent_dis == 500);
     :: ((bonbons > 0) && argent_channel?[100]) ->
          argent_channel?100;
          argent_dis = argent_dis + 100;
```

```
          budget=budget-100;
          sucrerie_channel!bonbon;
          bonbons = bonbons-1;
          assert(budget+argent_dis == 500);
     :: (chocolats==0) && (bonbons==0) -> break
     od
}
init { atomic { run client(); run distributeur(); }}
```

Exercice 2

a. Affectation dynamique de deux rôles

```
#define MAITRE 1
#define ESCLAVE 2

short som_role ;

proctype affecte_role (chan inc, out)
{byte m1, m2, role, fin ;

do
/* :: (role == 0) -> */
  ::
   if /* choix aleatoire de role */
   :: m1=MAITRE ;
   :: m1=ESCLAVE ;
   fi ;
   out!m1 ; inc ?m2 ;
   if
   ::(m1 != m2) -> role=m1 ;
      som_role = som_role + role ; out!fin ; break ;
                    /* out!fin permet de valider la verification */
   :: else -> skip ;
   fi
od ;
printf(« le processus %d a le role : %d \n »,_pid,role) ;
inc?fin -> assert(som_role==3) ; /* inc ?fin est une balise assurant
                    que l'autre processus a aussi fini son calcul */
}


init
{chan AtoB = [1] of {byte} ;
 chan BtoA = [1] of {byte} ;

atomic {
     run affecte_role(AtoB,BtoA) ;
     run affecte_role(BtoA,AtoB) ;
}}
```

c. Affectation dynamique de trois rôles

```
#define SERVEUR_PRIMAIRE 1
#define SERVEUR_SECONDAIRE 2
#define CLIENT 3


short som_role;

proctype affecte_role (chan inx, iny, outx, outy)
{byte m1, m2,m3, role, fin;

do
/*:: (role == 0) -> */
:: if /* choix aleatoire de role */
   :: m1= SERVEUR_PRIMAIRE;
   :: m1= SERVEUR_SECONDAIRE;
   :: m1= CLIENT;
   fi;
   outx!m1; outy!m1; inx?m2; iny?m3;
   if
   ::(m1 != m2) && (m1 != m3) && (m2 != m3) -> role=m1;
     som_role = som_role + role; outx!fin; outy!fin; break;
                /* out!fin permet de valider la verification */
   :: else -> skip;
   fi
od;
printf("le processus %d a le role : %d \n",_pid,role);
(inx?[fin]) && (iny?[fin]) -> assert(som_role==6);
                /* (inx?fin) && (iny?fin) est une balise assurant que
                les autres processus ont aussi fini leur calcul */
}


init
{chan AtoB = [1] of {byte};
 chan AtoC = [1] of {byte};
 chan BtoA = [1] of {byte};
 chan BtoC = [1] of {byte};
 chan CtoA = [1] of {byte};
 chan CtoB = [1] of {byte};

atomic {
     run affecte_role(BtoA,CtoA,AtoB,AtoC);
     run affecte_role(AtoB,CtoB,BtoA,BtoC);
     run affecte_role(AtoC,BtoC,CtoA,CtoB);
}}
```