Université Paris 7
U.F.R d'informatique

DESS Logiciels Fondamentaux
Année 2002-2003

TP 6 – SMV

1 Syntaxe de SMV

Après Spin nous allons découvrir SMV, pour lequel il existe deux grandes méthodes de modélisation d'automates. Elles utilisent respectivement les mots clés ASSIGN et TRANS.

1.1 ASSIGN

Voici un exemple pour la syntaxe ASSIGN, dans lequel, comme pour spin, on distingue 3 parties :

- Un module
- Un programme principal
- Une spécification

Le programme principal déclare deux variables de type process qui s'executeront de manière concurrente. Chacun des deux process est une instance de *user*. Un *user* possède une variable locale *state* a valeur dans un domaine a trois éléments. Initialement la variable *state* est égale a *repos*. Dessinez l'automate non déterministe associe naturellement a un *user*

```
-- Probleme de l'exclusion mututelle.
-- modules
MODULE user
VAR
        state:{repos,demande,impression};
ASSIGN
        init(state):=repos;
        next(state):=case
                (state=repos)       : {repos,demande};
                (state=demande)     : impression;
                (state=impression)  : {impression, repos};
        esac;
FAIRNESS
        running

FAIRNESS
        !state=impression
-- Main
MODULE main
VAR
        p0   : process user;
```

```
        p1   : process user;
-- Specifications
SPEC
        !EF (p0.state=impression & p1.state=impression)
```

Tapez le programme précèdent et lancer *xNuSMV*. L'onglet *processAll* vous permet de sélectionner votre fichier. Le *TraceViewer* vous permet de retrouver un contre exemple, de sélectionner les variables que vous voulez voir afficher, le *Simulate* vous permet de simuler vous même un comportement, le résultat sera visible dans le *TraceViewer*.

1.2 TRANS

Voici une seconde version de l'exclusion mutuelle qui utilise la modélisation avec *TRANS*

```
-- Probleme de l'exclusion mututelle.
-- Modules
MODULE user(t,id)
VAR
        state:{repos,demande,impression};
INIT
        state=repos
TRANS
        !t=id & next(state)=state
|        t=id &
(         state=repos & next(state) in {repos,demande}
  |       state=demande & next(state)=impression
  |       state=impression & next(state) in {impression, repos}
)
FAIRNESS
        t=id
FAIRNESS
        !state=impression
-- Main
MODULE main
VAR
        t : 0..1;
        p0   : user(t,0);
        p1   : user(t,1);
INIT
        t in {0,1}
TRANS
--      Round Robin scheduling
--      next(t) = t+1 mod 2
--      Asynchronous scheduling
        next(t) in {0,1}
-- specifications
```

```
SPEC    !EF (p0.state=impression & p1.state=impression)
```

TRANS est vu comme la table de transition du module. Elle est présentée comme une grande disjonction (le choix parmi toutes les transitions), et chacune est une formule liant les gardes les actions et les franchissements. La table de transition globale est la conjonction des différentes tables des modules.

Dessinez l'automate a compteurs associe a un *user*.

Évaluez le modelé a l'aide de *xNuSMV*, avec les deux scheduling proposes. Écrivez ce modèle avec la syntaxe *ASSIGN*.

On veut que le tour propose par le scheduler ne puisse changer que lorsque les utilisateurs sont au repos. Cette modification est elle suffisante pour assurer l'exclusion mutuelle. Vérifiez le.

2 L'algorithme auto-stabilisateur de Dijkstra (74)

La principale différence pour le modelisateur, est que les variables de SMV sont toutes a domaine fini, et qu'il n'y a pas de canaux de communication. Vous devrez donc implementer ces canaux quand vous en aurez besoin.

Pour vous faire votre propre opinion lorsqu'il n'y a pas de canaux, on reprend le protocole autostabilisant de Dijkstra. Effectuez les deux modélisation *TRANS* et *ASSIGN*, puis comparez avec celle que vous avez faite en *Spin* (éventuellement terminez la maintenant que vous savez écrire des spécifications en *LTL*).

Rappel : Il s'agit d'un algorithme d'exclusion mutuelle pour des systèmes distribués disposés en anneaux. Les n processus, $P_0 \ldots P_{n-1}$ peuvent chacun demander des informations à leur voisin de droite ($P_{i-1 \ mod \ n}$) immédiat. Pour chaque P_i, le privilège d'être en section critique est matérialisé par une configuration de ses variables propres et de celles de son voisin de gauche.

Chaque processus possède une variable *drapeau* qu'il peut lire et ecrire, et il peut également lire le contenu de *drapeau* pour le processus à sa droite.

Le protocole proposé n'est pas symétrique, et P_0 y joue un rôle particulier.

Protocole pour P_0 :
attendre $drapeau == P_{n-1}.drapeau$
< Section critique>
drapeau = drapeau +1 mod k

Protocole pour $P_i, i \neq 0$:
attendre $drapeau \neq P_{i-1}.drapeau$
< Section Critique >
$drapeau = P_{i-1}.drapeau$

Initialement chaque *drapeau* est mis à 0 et $k > n$.

1. Réaliser un modèle *SMV* pour ce protocole.

2. Après simulation, on peut observer quelques particularités de ce protocole. Donner en langue naturelle un ensemble de spécifications significatives.
3. Essayer de faire croître la taille de l'anneau pour trouver les limites de l'outil.
4. Cet algorithme est dit autostabilisant car si on n'impose plus que chaque drapeau soit mis à zéro initialement, on parvient au bout d'un nombre fini de transitions dans un état où de toutes façons un seul processus possède le privilège d'être en section critique. Cette propriété permet d'envisager par exemple un problème électrique temporaire, et une correction automatique. Pouvez vous intégrer ce défaut de façon réaliste dans votre modèle ?