# DESS - Semaine du 21/10/2002

**Exercice** 1 : On considere l'algorithme suivant d'exclusion mutuelle entre deux processus. Il a ete publié dans un journal de l'Association for Computing Machinery il y a une vingtaine d'annees.

```
Boolean array b[0..1] initialized to true;
Integer k;

Process(i) with i either 0 or 1
begin
1 : the core program
    b[i] := false;
2 : if (k!=i) then begin
3 : if (not b[1-i]) then goto 3 else k:=i;
    goto 2 end;
    Critical Section
    b[i] := true; goto 1;
end
```

Dans ce programme, sont consideres comme atomiques, les affectations, les evaluations de conditions, les goto.

1. Dessinez l'automate qui correspond au processus $i$.
2. Montrer que le programme presente un "livelock", c.a.d une execution dans laquelle les deux processus restent indefiniment en attente sans qu'aucun ne rentre en section critique.
3. Une fois le probleme precedent corrige, montrer que l'algorithme n'assure pas l'exclusion mutuelle. Quelle instruction faudrait-il rendre atomique pour que l'excusion mutuelle soit assuree.
4. Simplifier l'automate, en fusionnant les transitions de goto avec les transitions qui les precedent, et apres avoir rendu atomique l'instruction de la question precedente.
5. dessiner l'automate global du systeme, et montrer que l'exclusion mutuelle est maintenant assuree.
6. montrer que cet algorithme n'assure pas l'absence de famine, meme en supposant les equites usuelles. C.a.d qu'il existe une execution pour laquelle l'un des processus n'entre jamais en Section Critique, alors qu'il en a formule la demande.

**Exercice** 2 :

Voici le modele d'un programme ecrit en Spin

```
/* Algorithme d'exclusion mutuelle de Peterson 1981 */

#define trans(garde,action,etat) d_step{garde -> action} goto etat

bool req[2];            /* variables globales */
bit tour;

proctype user(bit id)
{
repos:
        if
        :: trans(true,skip,repos)
        :: trans(true,req[id] = true,demande)
        fi;
demande:
        if
        :: trans(true,tour = 1-id,attente)
        fi;
attente:
        if
        :: trans((tour==id || req[1-id]==false),skip,impression)
        :: trans(else,skip,attente)
        fi;
impression:
        if
        :: trans(true,skip,impression)
        :: trans(true,req[id] = false,repos)
        fi
}

init                    /* pid 0 */
{
        atomic {
        tour=0;
        req[0]=false;
        req[1]=false;
        run user(1);        /* pid 1 */
        run user(0)         /* pid 2 */
        }
}
```

```
/*
Propositions atomiques:
*/
#define l1 _last==1
#define l0 _last==2
#define d1 (user[1]@demande)
#define d0 (user[2]@demande)
#define i1 (user[1]@impression)
#define i0 (user[2]@impression)
/*
Specifications:

Jamais 2 impressions simultanees
[](!i0 || !i1) ou !<>(i0 && i1)

Toute demande d'impression doit etre satisfaite.
Sans equite.
[] (d1 -> <> i1) ou !<>(d1 && [] !i1)

Toute demande d'impression doit etre satisfaite.
Equite: chaque processus est execute infiniment souvent.
([]( (<> l1) && (<> l0) )) -> ([] (d1 -> <> i1))

Toute demande d'impression doit etre satisfaite.
Equite: chaque processus est execute infiniment souvent.
Equite: toute impression se termine.
([]( (<> l1) && (<> l0) && (<> !i0) && (<> !i1) )) -> ([] (d1 -> <> i1))
sous forme negative :
[]( <> l1 && <> l0 && <> ! i0 && <> ! i1 ) && <> (d1 && [] ! i1)
*/

never { /* []( <> l1 && <> l0 && <> ! i0 && <> ! i1 ) && <> (d1 && [] ! i1) */
T0_init:
        if
        :: (1) -> goto T0_init
        :: (!i1 && d1) -> goto T0_S32
        fi;
T0_S32:
        if
        :: (!i1) -> goto T0_S32
        :: (l1 && !i1) -> goto T1_S32
        :: (l1 && l0 && !i1) -> goto T2_S32
        :: (l1 && l0 && !i0 && !i1) -> goto accept_S32
        fi;
T2_S32:
        if
        :: (!i1) -> goto T2_S32
        :: (!i0 && !i1) -> goto accept_S32
        fi;
T1_S32:
        if
        :: (!i1) -> goto T1_S32
        :: (l0 && !i1) -> goto T2_S32
        :: (l0 && !i0 && !i1) -> goto accept_S32
        fi;
accept_S32:
        if
        :: (!i1) -> goto T0_S32
        :: (l1 && !i1) -> goto T1_S32
        :: (l1 && l0 && !i1) -> goto T2_S32
        :: (l1 && l0 && !i0 && !i1) -> goto accept_S32
        fi;
}
```

1. notez les explications, la semaine prochaine vous ecrirez du Spin
2. dessiner l'automate du modele
3. dessinez l'automate du never claim

**<u>Exercice</u>** 3 : Voici un autre exemple d'exclusion mutuelle

```
/* Algorithme d'exclusion mutuelle avec tour. */

#define trans(garde,action,etat) atomic{d_step{garde -> action}; goto etat}

bit tour;               /* variable globale */

proctype user(bit id)
{
repos:
        if
        :: trans(true,skip,repos)
        :: trans(true,skip,demande)
        fi;
demande:
        if
        :: trans(tour == id,skip,impression)
        :: trans(else,skip,demande)
        fi;
impression:
        if
        :: trans(true,skip,impression)
        :: trans(true,tour = 1-id,repos)
        fi
}

init                    /* pid 0 */
{
        atomic {
        run user(1);        /* pid 1 */
        run user(0)         /* pid 2 */
```

}
}

## Une télévision

On considère une télévision qu'on abstrait en considérant un module pour chacun des composants suivants :

- la télécommande
- le récepteur infrarouge
- le son

On ne considère que les actions sur le volume. Vu de la télécommande elles sont **push_up, push_down, push_off,push_on**.

L'émission et la reception d'un signal $s$ sont notées respectivement par les actions $!s$ et $?s$.

1 Donner une modélisation de la télécommande

Pour modéliser les signaux entre les composants cablés de la télévision, on utilise des variables booléennes partagées par les systèmes concernés par la communication. L'envoi d'un signal correspond alors à une affectation d'une variable à **vrai** et la reception n'est possible que si la variable a pour valeur **vrai** et a pour effet de lui donner **faux** pour valeur.

On souhaite toutefois utiliser les mêmes conventions de notation que pour la télécommande.

2 Si $x_s$ est la variable associée au signal cablé $s$ donner l'étiquette explicite correspondant aux actions $!s$ et $?s$.

Sur réception d'un signal interne **vol_up** ou **vol_down** le volume décroit ou croit en restant dans un niveau compris entre 0 et 4. Lorsque le volume est en sourdine (réception de **vol_off**), la réception d'un des signaux **vol_up**,**vol_down**,**vol_on** n'a pour effet que de remettre le volume à son niveau d'origine.

3 Modéliser le module son et le récepteur infrarouge

On désire enrichir la modélisation de la télévision en y incluant l'écran (!!).

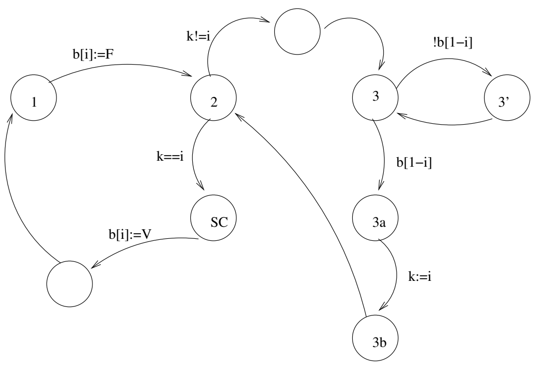
Lorsque le son du téléviseur a été modifié, l'écran affiche la valeur courante du volume (y compris en sourdine). Cet affichage débute sur reception d'un signal **d_on**. On supposera que la valeur $i$ du volume est bel et bien affichée lorsque l'écran est en mode $\mathbf{display}_i$.

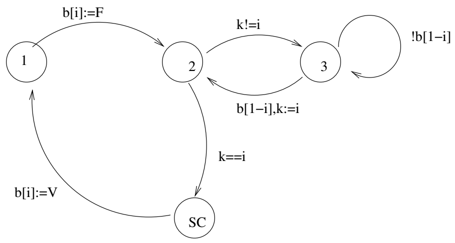
Sur reception d'un signal **timeout** provenant d'un module non modélisé ici, l'écran passe en mode **undisplay**.

4 Donner une modélisation du module ecran, en modifiant éventuellement les autres modules.

5 Préciser quelles sont les variables partagées, et par quels modules.

6 Discuter les pertes de messages.




<u>**Correction**</u> 1 :

1. c'est l'automate de la figure 1

2. initialement, si $k! = 0, 1$, en $(2, 2)$ on a $b[0, 1] = [F, F] \Rightarrow (3, 3')^*$
3. voici un contre exemple : $(1, 1), (1, 2), (1, 3), (1, 3a), (2, 3a), (SC, 3a), (SC, 3b), (SC, 2), (SC, SC)$
4. voir figure 4
5. C'est un produit asynchrone. Les gardes entrantes de $(SC, SC)$ doivent etre $\perp$
6. L'equite "chacun a la main infiniment souvent" n'empeche pas que si les 2 sont en attente, alors c'est toujours le meme qui rentre.