Synchronisation - Sémaphores - Modélisation formelle et vérification de propriétés

Ensimag 2A

1 Sémaphores

1.1 Présentation

Un sémaphore `s` est un objet de synchronisation avec les attributs suivants :

- Un compteur entier *privé*, noté `s.c`, qui est initialisé à une valeur `c0` à la création du sémaphore.
- Une file d'attente *privée* pour gérer les processus bloqués, notée `s.f`.

Un sémaphore possède également deux méthodes *publiques* qui sont exécutées en *exclusion mutuelle* et qui vont utiliser les attributs privés :

```
s.P() {
   s.c--;
   if (s.c < 0)
      f.wait();
      // Le processus se bloque dans s.f
}

s.V() {
   s.c++;
   if (s.c <= 0)
      f.signal();
      // Débloque un processus de s.f
}
```

A partir de la valeur courante c d'un sémaphore s, on peut dire que :

- $c >= 0$ est le nombre de ressources que peuvent prendre des processus appelant $P()$ sans se bloquer.
- $c < 0$ est l'opposé du nombre de processus bloqués dans $s.f$.

Dans le cadre de ce TD, les processus sont débloqués dans un ordre FIFO.

1.2 Modélisation de programmes utilisant des sémaphores

On considère que les programmes utilisant des sémaphores sont exécutés sur une machine monoprocesseur, grâce à l'ordonnanceur préemptif d'un système d'exploitation simple. Typiquement la préemption d'un processus se fait sur une base de temps. Le système d'exploitation sauvegarde correctement le contexte des processus lors de la préemption, ce qui permet d'utiliser des variables *locales* dans le code des processus, sans craindre qu'elles soient écrasées lors du changement de contexte. Les primitives des sémaphores sont rendues *atomiques*, de telle sorte qu'un processus ne puisse pas être interrompu lorsqu'il est en train d'exécuter le code de $P()$ ou $V()$. On ne fait aucune autre hypothèse.

Dans ces conditions, il est possible de représenter par des automates le fonctionnement de chaque processus, et par un *produit asynchrone* d'automates le comportement de l'ensemble processus+mécanisme de synchronisation. Le produit (nous allons voir que c'est une variante de produit cartésien), représente *tous les états possibles*. En l'inspectant, on peut repérer les deadlocks, les famines, les états qui devraient être inaccessibles et qui pourtant ne le sont pas (comme lorsqu'on fait une erreur en essayant de programmer l'exclusion mutuelle).

1.2.1 Les automates de base

Un automate représentant un processus séquentiel est un n-uplet (Q, q_0, T) où :

- Q est l'ensemble des états
- $q_0 \in Q$ est l'état initial
- $T \subseteq Q \times L \times Q$ est l'ensemble des transitions. Chaque transition est un triplet (q, ℓ, q'), où q est l'état de départ, q' est l'état d'arrivée, et $\ell \in L$ est l'étiquette de la transition. L'ensemble L des étiquettes possibles représente les "pas" atomiques que peut exécuter un processus séquentiel.

La figure 1 donne un exemple de programme utilisant des sémaphores, et l'automate correspondant. Il est très important de noter que l'hypothèse sur l'atomicité de $P()$ et $V()$ est prise en compte : chacune de ces actions correspond à une transition. Dans l'exemple on suppose aussi que `A` et `B` représentent des instructions atomiques.

```
while (1) { A; s.P(); B; s.V(); }
```

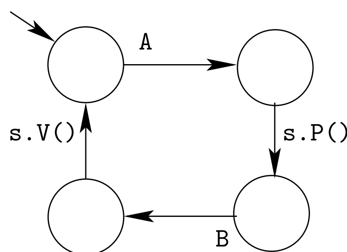


FIGURE 1 – Exemple de programme utilisant des sémaphores, et automate correspondant

1.2.2 Le produit asynchrone (pur)

Etant donnés deux automates $A_i = (Q_i, q_o^i, T_i)$, $i \in \{1,2\}$, l'automate produit est défini par : $A = A_1 \times A_2 = (Q_1 \times Q_2, (q_0^1, q_0^2), T)$, où l'ensemble T des transitions du produit est construit à partir de T_1 et T_2 en appliquant uniquement les règles suivantes :

— S'il existe $(q_1, \ell_1, q_1') \in T_1)$, et $q_2 \in Q_2$ alors $((q_1, q_2), \ell_1, (q_1', q_2)) \in T$

— S'il existe $(q_2, \ell_2, q_2') \in T_2)$, et $q_1 \in Q_1$ alors $((q_1, q_2), \ell_2, (q_1, q_2')) \in T$

La figure 2 donne un exemple de produit. `A, A', B, B'` représentent des instructions atomiques quelconques, éventuellement des appels de primitives des sémaphores.

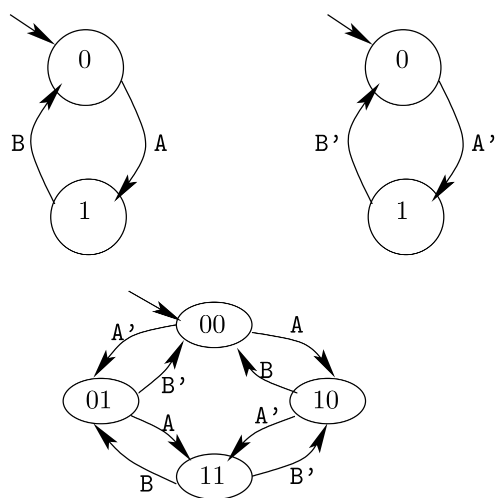


FIGURE 2 – Exemple de produit asynchrone

Le produit asynchrone représente tous les états globaux possibles que peut atteindre l'ensemble des deux processus. Dans l'exemple ils n'utilisent aucun moyen de synchronisation, et donc le produit cartésien complet des états est effectivement accessible depuis l'état initial.

1.2.3 Le produit asynchrone avec sémaphores

La construction du produit est intéressante surtout quand les deux processus utilisent un mécanisme de synchronisation, parce que cela permet de construcire l'ensemble des états accessibles et donc de vérifier certaines propriétés comme l'exclusion mutuelle. Etant donnés deux automates qui représentent des processus utilisant des sémaphores, on ajoute au produit des informations sur l'état courant de chaque sémaphore. Nous donnons ci-dessous la définition pour 2 processus et 1 sémaphore, mais cela s'étend facilement à n processus et m sémaphores.

On notera FIFO(p) l'ensemble des FIFO à p places. On notera de la manière suivante les opérations sur les FIFO : si f est une FIFO, $+(f, n)$ dénote la FIFO obtenue en ajoutant l'élément n à f ; $-f$ dénote la FIFO obtenue en supprimant un élément de la FIFO. Ajout et suppression se font en respectant la politique FIFO. On note [] la FIFO vide. On utilise une notation à la C pour les expressions conditionnelles : $(c?t : e) = \texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ e$.

Etant donnés deux automates $A_i = (Q_i, q_o^i, T_i), i \in \{1,2\}$, et un sémaphore $s = (s.c, s.f)$ initialisé à $(c_0, [])$, le produit est $A_s = (Q_1 \times Q_2 \times \mathbb{Z} \times \text{FIFO}(2)), (q_0^1, q_0^2, c_0, []), T)$ où l'ensemble T des transitions du produit est construit à partir de T_1 et T_2 en appliquant uniquement les règles suivantes :

— S'il existe $(q_1, \ell_1, q_1') \in T_1$, et $q_2 \in Q_2$ alors
  — Si ℓ_1 représente une action interne du processus 1, et $1 \notin f$, alors $((q_1, q_2, c, f), \ell_1, (q_1', q_2, c, f)) \in T$
  — Si $\ell_1 = s.P()$, et $1 \notin f$, alors $((q_1, q_2, c, f), \ell_1, (q_1', q_2, c-1, (c-1 < 0? + (f, 1) : f)) \in T$
  — Si $\ell_1 = s.V()$, et $1 \notin f$, alors $((q_1, q_2, c, f), \ell_1, (q_1', q_2, c+1, (c+1 \leq 0? - f : f)) \in T$

— Et symétriquement.

La page 8 donne un exemple de processus séquentiel, et le produit asynchrone avec sémaphore qui représente le comportement d'un ensemble de deux processus exécutant ce même code. Chaque état global comporte : l'état du premier processus, l'état du deuxième processus, et l'état du sémaphore (compteur et file d'attente). Le sémaphore est initialisé à 1.

1.2.4 Le programme Java de construction et affichage du produit

L'annexe A, page 4, donne un squelette de programme Java pour construire et afficher les pro-

duits. Les transitions sont produites dans un fichier `.dot`, de manière à bénéficier des outils de dessin automatique associés (Cf. le paquet `graphviz` disponible en Linux et Windows). La portion de code commentée "*build first transition*" correspond à la règle de construction d'une transition du produit décrite plus haut (paragraphe 1.2.3). La portion "*build second transition*" correspond au cas symétrique. La page 6 montre comment compacter le code lorsque les deux processus exécutent exactement les mêmes instructions. On pourra se servir de ce schéma également si le code des deux processus diffère très peu (avec des `if` sur l'identité du processus donnée par le paramètre `me`).

2 Exercices

2.1 Utilisation Simple

En utilisant un sémaphore, on reproduit le comportement d'un mutex pour former une exclusion mutuelle autour d'une section critique. On considère deux processus `i=1,2` qui exécutent le code suivant (le sémaphore est initialisé à 1) :

```
Avant_i ();  // portion de code qq
smutex.P();
 C_i(); // code en section critique
 D_i();
smutex.V();
Apres_i() ; // portion de code qq
```

Question 1
— *Dessiner l'automate de chaque processus, avec les règles définies plus haut*
— *Dessiner l'automate du produit asynchrone avec sémaphore*
— *Modifier le programme Java donné en annexe pour représenter l'ensemble des états possibles des deux processus et du sémaphore.*
— *Comment vérifie-t-on que l'exclusion mutuelle est bien respectée ?*

2.2 Mécanisme de synchronisation à découvrir

On considère deux processus dont le code est donné Figure 3, page 3.

Question 2
— *On considère que les actions* `Avant_1()`, `Apres_1()`, `Avant_2()`, `Apres_2()` *sont atomiques. Dessiner les automates correspondant aux deux processus de la figure 3.*
— *Modifier le programme Java de manière à coder ces deux automates, l'exécuter pour obtenir le produit asynchrone, observer ce produit.*
— *En déduire quel mécanisme de synchronisation on obtient en utilisant deux sémaphores comme sur la figure 3.*

```
sem A = 0;
sem B = 0;

proc_1() {
   Avant_1();
   A.V();
   B.P();
   Apres_1();
}

proc_2() {
   Avant_2();
   B.V();
   A.P();
   Apres_2();
}
```

FIGURE 3 – Effet de synchronisation à découvrir entre deux processus

2.3 Interblocages

Question 3
— *Ecrire deux processus, les plus simples possibles, qui utilisent deux sémaphores et entrent en interblocage.*
— *Dessiner les automates, les coder dans le programme Java, générer le graphe.*
— *Comment se manifeste l'interblocage dans ce graphe ?*

2.4 Divers problèmes de synchronisation (et leurs modèles)

2.4.1 Rendez-vous à trois

...

2.4.2 Barrière de synchronisation

N processus arrivent les uns après les autres à une barrière. Ils s'attendent les uns les autres jusqu'à ce que les N soient arrivés. Ils peuvent alors poursuivre.

Question 4 *En utilisant des sémaphores, faire une barrière qui bloque tous les processus jusqu'à ce que N soient arrivés. On ne demande pas que le code soit "réutilisable".*

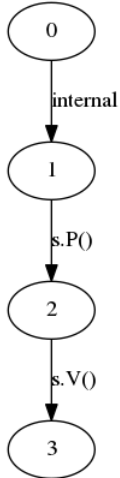
A Exemple de processus, et produit asynchrone

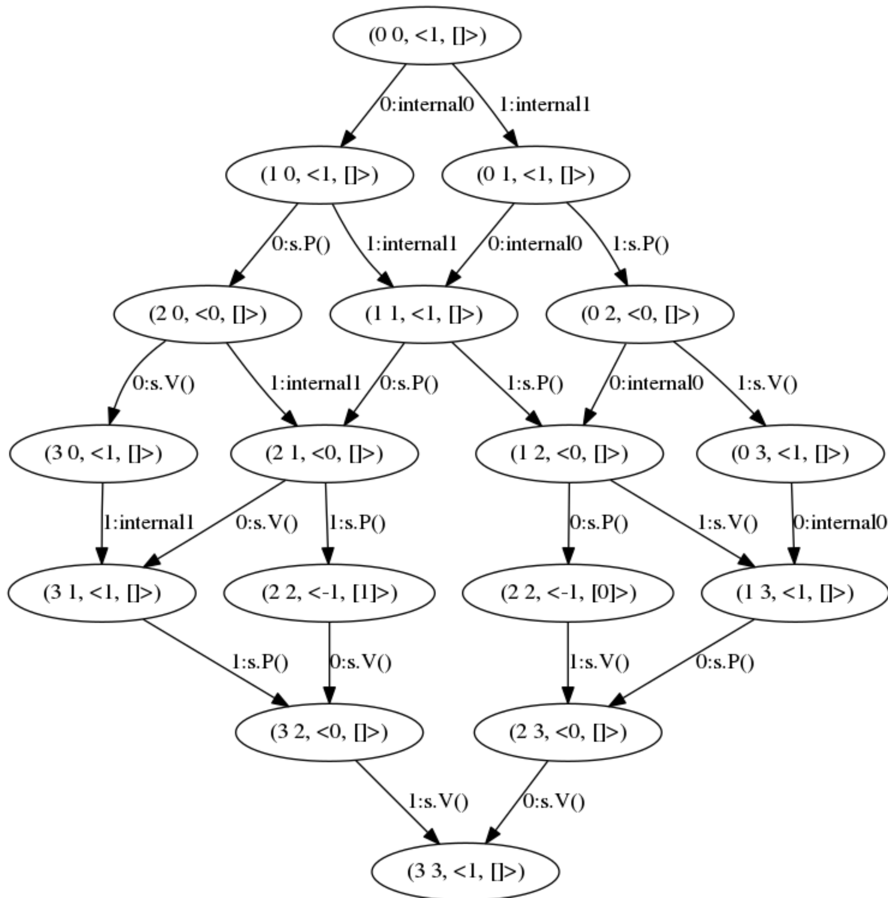
```
1  import java.util.* ;
2
3  // "functional" FIFO, no in-place modifications.
4  class FIFO {
5      Queue queue ;
6      FIFO () { queue  = new LinkedList();}
7      FIFO (Queue q) { queue  = q ;}
8      public FIFO add (int i) {
9          Queue mod =  new LinkedList();
10         mod.addAll (queue); mod.add (i);
11         return new FIFO(mod) ;
12     }
13     public boolean contains (int i) { return queue.contains(i);}
14
15     public boolean equals (Object other) {
16         if (!(other instanceof FIFO)) return false ;
17         LinkedList oq = (LinkedList)((FIFO)other).queue ;
18     if (oq.size()!=this.queue.size()) return false;
19         for (int i = 0 ; i < this.queue.size() ; i++) {
20             if (((LinkedList)queue).get(i) != oq.get(i)) return false ;
21         }
22         return true ;
23     }
24
25     public FIFO remove () {
26         Queue mod =  new LinkedList();
27         mod.addAll (queue); mod.remove ();
28         return new FIFO(mod) ;
29     }
30     public String toString () { return queue.toString();}
31     public int hashCode() { return queue.hashCode() ;}
32 }
33
34 class SemaphoreState {
35     int count ;     // the counter
36     FIFO blocked ; // the fifo of blocked processes
37     SemaphoreState (int c, FIFO f) { count =c ; blocked = f;}
38     SemaphoreState (int c) { count = c; blocked = new FIFO();}
39
40     public SemaphoreState P(int process) {
41         return new SemaphoreState(count-1, (count-1<0? blocked.add(process): blocked));}
42     public SemaphoreState V() {
43         return new SemaphoreState(count+1, (count+1<=0?blocked.remove():blocked));}
44     public boolean isBlocked (int process) { return blocked.contains(process);}
45
46     public boolean equals (Object other){
47         return
48             other instanceof SemaphoreState &&
49             count   == ((SemaphoreState)other).count &&
50             blocked.equals (((SemaphoreState)other).blocked) ;
51         }
52     public int hashCode() { return count+blocked.hashCode() ; }
53     public String  toString() { return "<" + count + ", " + blocked + ">" ; }
54 }
55
56 class state {
57     SemaphoreState sem ; // state of the semaphore
58     int s0, s1;           // states in the two processes
59     state (int i0, int i1, SemaphoreState s) { s0 = i0 ; s1 = i1 ; sem = s ;}
60     public String toString () {return "\"(" + s0 + " " + s1 + ", " + sem +")\"";}
61     public int hashCode () {return s0+s1+sem.hashCode();}
62     public boolean equals (Object other){
63         return
64             other instanceof state &&
65             s0 ==  ((state)other).s0 &&
66             s1 ==  ((state)other).s1 &&
67             sem.equals (((state)other).sem);
68     }
```

```
69  }
70  class ProductWithSem {
71    public static void main (String args[]) {
72
73      Set<state> all      = new HashSet<state> ();
74      Set<state> explored = new HashSet<state> ();
75
76      // initial state (s0, s1,  sem)
77      all.add (new state (0, 0, new SemaphoreState(1)));
78
79      System.out.println ("digraph g {");
80      while (! all.isEmpty()) {
81          state s = (state) (all.toArray())[0];
82          all.remove (s) ;
83          if (!explored.contains (s)) {
84
85          explored.add (s) ;
86
87          // build the first transition (first one moves)
88          // we show the three cases of the definition:
89          // -- internal action from 0 to 1 (no change on the semaphore)
90          // -- action s.P() from 1 to 2
91          // -- action s.V() from 2 to 3
92          state ss0 = null;
93          String str0 = "";
94          if (!s.sem.isBlocked(0)){
95              switch (s.s0) {
96              case 0:
97                  ss0 = new state (1,   s.s1, s.sem) ;
98                  str0 += "internal0" ; break ;
99              case 1:
100                 ss0 = new state (2,   s.s1, s.sem.P(0));
101                 str0 += "s.P()" ; break;
102             case 2:
103                 ss0 = new state (3,   s.s1, s.sem.V());
104                 str0 += "s.V()" ; break;
105             }
106         }
107         if (ss0 != null) {
108             System.out.println (s + "->" + ss0 + "[label=\"0:"+ str0+ "\"]");
109             all.add (ss0) ;
110
111         }
112
113         // build the second transition (similar)
114         state ss1 = null;
115         String str1="";
116         if (!s.sem.isBlocked(1)) {
117             switch (s.s1) {
118             case 0:
119                 ss1 = new state (s.s0, 1,   s.sem) ;
120                 str1 += "internal1" ; break ;
121             case 1:
122                 ss1 = new state (s.s0, 2,   s.sem.P(1));
123                 str1 += "s.P()" ; break;
124             case 2:
125                 ss1 = new state (s.s0, 3,   s.sem.V());
126                 str1 += "s.V()" ; break;
127             }
128         }
129         if (ss1 != null) {
130             System.out.println (s + "->" + ss1 + "[label=\"1:"+ str1+ "\"]");
131             all.add (ss1) ;
132         }
133         }
134     }
135     System.out.println ("#Nb states = " + explored.size());
136     System.out.println ("}");
137   }
138 }
```

```
1  import java.util.* ;
2
3  // "functional" FIFO, no in-place modifications.
4  class FIFO {
5      Queue queue ;
6      FIFO () { queue  = new LinkedList();}
7      FIFO (Queue q) { queue  = q ;}
8      public FIFO add (int i) {
9          Queue mod =  new LinkedList();
10         mod.addAll (queue); mod.add (i);
11         return new FIFO(mod) ;
12     }
13     public boolean contains (int i) { return queue.contains(i);}
14
15     public boolean equals (Object other) {
16         if (!(other instanceof FIFO)) return false ;
17         LinkedList oq = (LinkedList)((FIFO)other).queue ;
18     if (oq.size()!=this.queue.size()) return false;
19         for (int i = 0 ; i < this.queue.size() ; i++) {
20             if (((LinkedList)queue).get(i) != oq.get(i)) return false ;
21         }
22         return true ;
23     }
24     public FIFO remove () {
25         Queue mod =  new LinkedList();
26         mod.addAll (queue); mod.remove ();
27         return new FIFO(mod) ;
28     }
29     public String toString () { return queue.toString();}
30     public int hashCode() { return queue.hashCode() ;}
31 }
32
33 class SemaphoreState {
34     int count ;     // the counter
35     FIFO blocked ; // the fifo of blocked processes
36     SemaphoreState (int c, FIFO f) { count =c ; blocked = f;}
37     SemaphoreState (int c) { count = c; blocked = new FIFO();}
38
39     public SemaphoreState P(int process) {
40         return new SemaphoreState(count-1, (count-1<0? blocked.add(process): blocked));}
41     public SemaphoreState V() {
42         return new SemaphoreState(count+1, (count+1<=0?blocked.remove():blocked));}
43     public boolean isBlocked (int process) { return blocked.contains(process);}
44
45     public boolean equals (Object other){
46         return
47             other instanceof SemaphoreState &&
48             count   == ((SemaphoreState)other).count &&
49             blocked.equals (((SemaphoreState)other).blocked) ;
50     }
51     public int hashCode() { return count+blocked.hashCode() ; }
52     public String  toString() { return "<" + count + ", " + blocked + ">" ; }
53 }
54
55 class state {
56     int  cp[] = new int[2]; // control states in the two or more processes
57     SemaphoreState sem ; // state of the semaphore
58     state (int cpi[], SemaphoreState s) { cp[0] = cpi[0] ; cp[1] = cpi[1] ; sem = s ;}
59     public String toString () {return "\"(" +  cp[0]+" " + cp[1] + ", " + sem +")\"";}
60     public int hashCode () {return cp[0]+cp[1]+sem.hashCode();}
61     public boolean equals (Object other){
62         return
63             other instanceof state &&
64             cp[0]  ==  ((state)other).cp[0] &&
65             cp[1]  ==  ((state)other).cp[1] &&
66             sem.equals (((state)other).sem);
67     }
```

```
68  }
69
70  class ProductWithSemG {
71      public static void buildAndPrint (int me, state s, Set<state> all){
72          // we show the three cases of the definition:
73          // -- internal action from 0 to 1 (no change on the semaphore)
74          // -- action s.P() from 1 to 2
75          // -- action s.V() from 2 to 3
76      state  ss  = null;
77      String str = "";
78      int[] newcp = new int[2];
79
80      for (int k=0;k<2;k++) newcp[k] = s.cp[k];
81      if (!s.sem.isBlocked(me)){
82              switch (s.cp[me]) {
83              case 0:
84                  newcp[me] =1;
85                  ss = new state (newcp, s.sem) ;
86                  str += "internal"+me ; break ;
87              case 1:
88                  newcp[me] =2;
89                  ss = new state (newcp, s.sem.P(me));
90                  newcp[me] =1;
91                  str += "s.P()" ; break;
92              case 2:
93                  newcp[me] =3;
94                  ss = new state (newcp, s.sem.V());
95                  str += "s.V()" ; break;
96              }
97      }
98      if (ss != null) {
99          System.out.println (s + "->" + ss + "[label=\"" + me + ":"+ str+ "\"]");
100         all.add (ss) ;
101     }
102     }
103
104
105     public static void main (String args[]) {
106
107     Set<state> all      = new HashSet<state> ();
108     Set<state> explored = new HashSet<state> ();
109     int initcp[] = {0, 0};
110
111     // initial state ([0.0], sem init a 1 )
112     all.add (new state (initcp, new SemaphoreState(1)));
113
114     System.out.println ("digraph␣g␣{");
115     while (! all.isEmpty()) {
116         state s = (state) (all.toArray())[0];
117         all.remove (s) ;
118         if (!explored.contains (s)) {
119
120             explored.add (s) ;
121             buildAndPrint (0, s, all);
122             buildAndPrint (1, s, all);
123         }
124     }
125     System.out.println ("#Nb␣states␣=␣" + explored.size());
126     System.out.println ("}");
127     }
128 }
```





Légende des états : $(q_1 q_2, < s.c, s.f >)$. Sémaphore initialisé à 1.
Légende des transitions : $n : \ell$, où n est le numéro du processus qui bouge, et ℓ l'étiquette de sa transition.

Noter en particulier qu'il est possible d'atteindre l'état 22 avec deux valeurs différentes du sémaphore, selon que le premier ou le deuxième processus a exécuté l'appel `s.P()` le premier. Noter aussi l'effet du sémaphore : quand, pour un état global G, le processus 1 est dans la file d'attente du sémaphore, il n'y a pas de transition par le processus 1 issue de G. Même chose pour le processus 2.