Téléinformatique & Protocoles

TP 3 : Le langage de spécification de protocole *Promela*

et le simulateur-vérificateur *spin*

Les premières sections de ce document décrivent sommairement le langage *"Promela"* et l'outil de simulation et de vérification *"spin"*. Un descriptif beaucoup plus complet (et en anglais) pourra être trouvé sur :

`http://cs.uwp.edu/staff/fossum/Classes/CS477/spin/HTML/Manual.html`

Promela signifie *"Protocol Meta Language"*. Il s'agit d'un langage qui permet de modéliser des systèmes distribués, des processus qui se déroulent en parallèle et qui se communiquent à travers des canaux.

Un modèle de protocole écrit par *Promela* peut être simulé et vérifié par le logiciel *spin* et son interface graphique *xspin*. Ces logiciels permettent d'étudier un protocole de façon exhaustive et de trouver toutes les possibilités d'erreur qu'il renferme, par exemple les deadlocks, les boucles etc.

Ainsi ni *Promela* ni *spin* ni *xspin* ne permettent de réaliser des programmes de communication proprement dits. Mais lorsqu'un protocole de communication a été modélisé sous *Promela* et qu'il a été prouvé qu'il ne contient aucune erreur, alors l'implantation effective du protocole dans un programme s'en déduit de façon assez directe.

1 Le langage *Promela*

1.1 Les types de données

Les types de données sont décrits dans le tableau suivant.

Nom du type	Equivalent C	Gamme de valeurs
bit ou *bool*	champ de bits	$\{0, 1\}$
byte	unsigned char	$[0, 255]$
short	short	$\left[-2^{15}-1, 2^{15}-1\right]$
int	int	$\left[-2^{31}-1, 2^{31}-1\right]$

bit et *bool* sont exactement synonymes et représentent un nombre qui ne peut prendre que les valeurs 0 et 1.

On peut également utiliser les tableaux, qui sont définis comme en C. Par exemple l'instruction :

```
byte state[30];
```

est une déclaration d'un tableau appelé `state` et composé de trente entiers positifs compris entre 0 et 255. L'utilisation de ces tableaux (la lecture de leur valeur et leur affectation) s'effectue également comme en C.

Enfin le type *channel* n'existe pas dans le langage C. En *Promela*, ce type représente un canal de transmission. Il sera présenté un peu plus loin.

1.2 Exécutabilité

En *Promela* toutes les instructions sont soit *exécutables*, soit *bloquantes*. Certaines instructions sont toujours exécutables, d'autres sont toujours bloquantes, mais la plupart peuvent être l'un ou l'autre en fonction des conditions. En particulier, une instruction bloquante peut être rendue exécutable (et vice-versa) par certains événements.

Par exemple en C la séquence :

```
while(a!=b); /* attend que a soit egal a b */
a = a+1;     /* ensuite augemente a de 1   */
```

bloque l'exécution du programme si a est différent de b. Le programme ne peut reprendre son cours que lorsque a sera devenu égal à b.

En *Promela*, cette séquence s'écrit simplement :

```
(a==b);  a=a+1;
```

Le séparateur "`->`" est exactement équivalent à "`;`". Il permet simplement, dans certains cas, de rendre les protocoles plus lisibles, notamment pour insister sur un enchaînement causal. Par exemple la séquence précédente s'écrit aussi :

```
(a==b) -> a=a+1;
```

ce qui peut être interprété comme "dès que a sera égal à b, a sera incrémenté de 1.

Les affectations, comme `a=2;` ou encore `a=a+1;` etc sont toujours exécutables.

L'instruction `skip` est une instruction toujours exécutable, mais qui ne fait rien et ne rend aucune valeur.

Les instructions `1;` ou encore `n;` (où n représente un nombre non-nul) sont également des instructions (qui ne servent pas à grand chose, certes) mais qui sont toujours exécutables.

`0;` est une autre instruction, tout aussi utile, mais qui n'est jamais exécutable.

1.3 Les processus

1.3.1 Les types de processus

Le comportement de tous les processus est défini par une déclaration `proctype`. Par exemple la déclaration suivante définit un type de processus appelé `A` avec une seule variable locale `state`.

```
proctype A()
{
    byte state;
    state = 3;
}
```

La syntaxe est proche de celle des procédures dans les langages comme *pascal* ou *C*. Pourtant comme nous le verrons ces définitions ont un rôle très différent de celui des procédures.

1.3.2 Instanciation de processus

A l'image du langage *C* et de la fonction principale `main`, tous les protocoles spécifiés par *Promela* commencent par l'exécution d'un processus principal appelé `init`. En général, ce processus principal a pour rôle de lancer des processus dont le comportement a été défini par les déclarations `proctype`. Par exemple :

```
proctype A()
{
    byte state;
    state = 3;
}

init
{
```

```
    run A();
}
```

lance l'exécution d'un processus de type `A()`. Mais `init` peut lancer autant de processus de type différent, ou de même type, éventuellement en leur donnant des arguments différents. Par exemple :

```
int b=3;

proctype A(int a)
{
    byte x;
    x = a+b;
}

proctype B(int m)
{
    byte y=0;
    (m==3) -> y=(y + b)%8;
}

init
{
    run A(2); run B(3); run B(5);
}
```

Dans cette spécification, `b` est une variable globale, alors que `x` et `y` sont des variables locales à chacun des processus.

Le processus `init` lance trois processus, l'un de type `A()` et les deux autres de type `B()`. Ces processus ne sont pas exécutés séquentiellement, l'un après l'autre, mais en parallèle. Le premier processus, de type `A`, affecte la somme `a+b` (c'est à dire 3+2=5) à la variable locale `x` et ensuite se termine.

Le second processus, de type et le troisième processus, sont tous les deux de type `B`, mais sont lancés avec des paramètres différents. Chacun compare le paramètre d'entrée avec 3. S'il y a égalité, alors la valeur de la variable `y` est augmenté de `b` modulo 8, et s'il n'y a pas égalité, alors le processus est terminé. Le second processus est lancé avec un paramètre d'entrée égal à 3. Donc dans ce processus, la valeur de `y` est effectivement augmenté, et le processus se termine. Par contre le troisième processus est lancé avec un paramètre d'entrée de 5. Donc la condition `(m==3)` n'est jamais exécutable. Donc le troisième processus reste bloqué sur cette instruction, sans pouvoir être débloqué.

Le fait de lancer un processus de type `A()`, c'est à dire le fait d'écrire `run A()` constitue une *instanciation* de ce type de processus. Autrement dit, une déclaration `proctype` ne fait que définir le comportement d'un type de processus. Il ne définit pas le processus lui-même.

1.3.3 Processus concurrents

Tous les processus lancés par des instanciations `run` s'exécutent non pas de manière séquentielle (`A` puis `B` puis ...) mais de manière concurrente. Par exemple, lançons trois processus `A`, `B` et `C` contenant chacun trois instructions $instrA1$, $instrA2$, $instrA3$, $instrB1$, etc.

Lorsque les processus s'exécutent de manière séquentielle, alors l'ordre d'exécution des instructions est celui donné dans le tableau suivant.

processus `A`	processus `B`	processus `C`
instrA1		
instrA2		
instrA3		
	instrB1	
	instrB2	
	instrB3	
		instrC1
		instrC2
		instrC3

Par contre, lorsque les processus s'exécutent de façon concurrente, on ne connaît pas a priori l'ordre dans lequel les instructions seront executées dans les différents processus. L'ordre peut très bien être l'ordre séquentiel, mais les instructions pourront aussi être entrelacés plus ou moins régulièrement. Voici un exemple.

processus `A`	processus `B`	processus `C`
instrA1		
		instrC1
instrA2		
	instrB1	
	instrB2	
	instrB3	
		instrC2
		instrC3
instrA3		

Il en résulte que, de la même manière que pour les systèmes distribués, si on n'étudie pas explicitement la synchronisation des processus, on pourra observer des résultats aléatoires. En voici un exemple.

```
byte state = 1;

proctype A()
{ (state==1) -> state = state+1; }

proctype B()
{ (state==1) -> state = state-1; }

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

`state` est une variable globale.

- Si `A` commence et termine avant `B`, alors la valeur de `state` ne sera plus de 1 et `B` restera éternellement bloqué à l'instruction `(state==1)` qui ne deviendra jamais exécutable. La valeur finale de `state` sera donc de 2.
- Si c'est `B` qui commence et termine avant `A`, alors c'est ce dernier qui restera bloqué et la valeur de `state` sera de 0.
- Enfin si les deux processus exécutent l'instruction `(state==1)` avant de changer la valeur de `state`, alors aucun des deux processus ne restera bloqué et la valeur de la variable `state` sera de 1.

Ainsi on peut voir que malgré une syntaxe voisine, les déclarations `proctype` ne définissent pas des procédures ou des fonctions comme en *C* ou en *Pascal*.

1.3.4 Séquences atomiques

Ce problème de *tester-affecter* est bien connu. Un des moyens de contrôler ce type de problème en *Promela* est d'utiliser les *séquences atomiques*. En enfermant un ensemble d'instructions dans un bloc préfixé `atomic`, on peut s'assurer que ces instructions seront exécutées comme un tout indivisible, sans que d'autres instructions venant d'autre processus s'intercalent entre eux. Par exemple dans l'exemple précédent, on pourra écrire :

```
byte state = 1;

proctype A()
{ atomic
  { (state==1) -> state = state+1; }
}

proctype B()
{ atomic
  { (state==1) -> state = state-1; }
}

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

Ainsi on peut s'assurer que la valeur de `state` ne peut être que 0 ou 2 et l'un des processus restera bloqué. Attention ! Les séquences atomiques ne résolvent pas tous les problèmes. En effet dans une séquence atomique, toutes les instructions doivent être exécutables, sauf éventuellement la première. Dans le cas, contraire, si une instruction dans la séquence n'est pas exécutable, alors il s'agira d'un *deadlock*, car aucun autre processus ne pourra rendre cette instruction exécutable et débloquer la situation.

1.4 Transmission de messages

1.4.1 La définition des canaux

Les *canaux* (*"channels"*) modélisent le transfert de données entre processus. On définit les canaux par des déclarations de ce type :

```
chan qname = [16] of {short}
```

Ceci définit un canal qui peut contenir jusqu'à 16 messages de type `short`. Les canaux sont des types de données comme les autres, dans le sens où on peut les passer en paramètre à des processus.

Chaque message peut avoir plus d'un champ. Par exemple, dans le canal suivant :

```
chan qname = [16] of {byte, int, chan, byte}
```

contient jusqu'à 16 message, chacun consistant en deux valeurs de 8 bits, une valeur de 32 bits et un nom de canal.

1.4.2 L'envoi et la réception de messages

L'instruction

```
qname!expr;
```

envoie la valeur de l'expression `expr` sur le canal que nous venons de créer. Plus précisément, elle met `expr` à la fin du canal.

```
qname?msg;
```

reçoit le message. Elle le retire du début du canal et l'enregistre dans la variable `msg`. Les canaux transfèrent les messages dans l'ordre *FIFO*.

Si plus d'une seule valeur doit être transférée dans un seul message, alors il faudra écrire :

```
qname!expr1,expr2,expr3;
qname?var1,var2,var3;
```

L'opération d'émission n'est exécutable que lorsque le canal en question n'est pas plein. De même, l'opération de réception n'est exécutable que lorsque le canal en question n'est pas vide.

Voici un exemple dans lequel le processus `init` lance deux processus `A` et `B` en leur donnant le même canal en paramètre. Le processus `A` envoie l'entier 123 au processus `B` qui l'enregistre dans la variable `x`.

```
proctype A(chan q1)
{  q1!123;  }

proctype B(chan q2)
{
    int x;
    q2?x;
}

init
{
     chan q1 = [1] of {int};
     run A(q1); run B(q1);
}
```

Enfin pour tester si un canal est vide ou non sans lire le message, on peut utiliser une instruction de type :

```
qname?[var];
```

Une telle instruction rend 1 si le canal contient un message, et 0 dans les autres cas.

Attention ! ! ! dans une séquence de type :

```
qname?[var] -> qname?var;
```

la seconde instruction n'est pas nécessairement exécutable. En effet entre l'exécution de la première instruction et celle de la seconde, d'autres instructions appartenant à d'autres processus peuvent être exécutées, et notamment ces instructions peuvent vider le canal `qname`. Il n'est pas non plus possible de recourir aux séquences atomiques pour résoudre ce problème, car dans les séquences atomiques, seul la première instruction peut ne pas être toujours exécutable. Or la lecture dans un canal n'est pas toujours exécutable. Donc dans le cas présent, on ne peut pas utiliser les séquences atomiques.

1.4.3 Synchronisation

Soit un canal de taille nulle :

```
chan qname = [0] of {byte};
```

Avec un tel canal, les messages peuvent passer mais ils ne peuvent pas être stockés. La lecture dans un tel canal est toujours bloquante, jusqu'à ce que, de l'autre côté du canal un, autre processus soit prêt à écrire quelque chose. De même l'écriture est bloquante jusqu'à ce que, de l'autre côté du canal, un autre processus soit prêt à y lire quelque chose.

Ainsi, en utilisant des canaux de taille nulle un processus peut *"donner rendez-vous"* à un autre processus à un point précis de leur évolution respective, de manière à pouvoir partir de ce point ensemble. Autrement dit, les canaux de taille nulle permettent de synchroniser plusieurs processus.

1.5 Flux de contrôle

1.5.1 La sélection ou l'instruction `if`

```
if
:: (a!=b) -> option1;
:: (a==b) -> option2;
fi;
```

Si `a` est égal à `b`, c'est l'option `option2` qui sera exécutée et sinon, c'est l'option `option1` qui sera exécutée. De manière plus générale, dans une instruction `if` on pourra avoir autant d'options qu'on le désire :

```
if
:: instr1 -> option1;
:: instr2 -> option2;
    ...
:: instrN -> optionN;
fi;
```

Au plus une option peut être choisie. Si parmi les premières instructions `instr1`, `instr2` ..., `instrN` :

– aucune instruction n'est exécutable, alors l'instruction `if` dans son ensemble ne sera pas exécutable.
– une seule `instrI` est exécutable, alors toute les instructions contenues dans `optionI` seront exécutés, mais les autres options ne le seront pas.
– plusieurs voire toutes les instructions sont exécutables, alors une des lignes est choisie au hasard et l'option correspondante est exécutée.

1.5.2 La répétition ou l'instruction `do`

```
do
:: count = count + 1;
:: count = count - 1;
:: (count == 0) -> break;
od;
```

Ici, aussi longtemps que le processus n'aura pas rencontré un `break` (ou un `goto` comme on le verra) il répétera les actions décrites ci-dessus pour l'instruction `if`. Dans l'exemple ci-dessus, les deux premières instructions sont toujours exécutables (ce sont des affectations) La troisième ne l'est que si `count` est nul. Ainsi, la variable `count` sera aléatoirement incrémenté ou décrémenté au cours des cycles de `do`. La boucle ne pourra s'arrêter que lorsque `count` est nul. Mais même s'il est nul, l'arrêt ne sera pas nécessaire, puisque les deux autres instructions restent toujours exécutables.

1.5.3 La pseudo-instruction `else`

Aussi bien pour l'instruction `if` que pour l'instruction `do`, on peut utiliser une option spéciale `else`. Elle n'est exécutable que lorsque toutes les autres options ne le sont pas. L'exécution de l'instruction `else` elle-même n'a aucun effet.

1.5.4 Le saut conditionnel ou l'instruction `goto`

C'est un autre moyen de sortir d'une boucle. Voici un exemple :

```
proctype PGCD(int x,y)
   {
        do
        :: (x>y) -> x = x-y
        :: (x<y) -> x = y-x
        :: (x==y) -> goto done
        od;
   done:
        skip
}
```

1.5.5 La temporisation

Dans *Promela*, il n'y a pas de notion de temps. On ne considère qu'une succession d'états. Donc il n'est pas possible de réaliser un véritable temporisateur. Mais il n'est pas non plus *souhaitable* d'avoir recours à un tel temporisateur, car il serait très difficile de *prouver* la validité d'un protocole avec un tel élément.

Dans *Promela*, on a recours à une instruction spéciale appelée `timeout` qui n'est exécutable que lorsqu'on est arrivé dans un état où tous les processus du système distribué sont bloqués. Par exemple :

```
do
:: qname?var;
:: timeout -> break;
od
```

Dans la séquence ci-dessus, si rien ne peut être lu dans le canal `qname` et si tous les autres processus sont bloqués, alors `timeout` devient exécutable et permet de sortir de la boucle.

2 Vérification

La section précédente décrivait comment on peut spécifier un protocole avec le langage *Promela*. Maintenant voyons comment il est possible de savoir si un protocole est juste ou non.

2.1 Un protocole *"correct"*

En réalité, un protocole, d'un point de vue intrinsèque, n'est ni correct, ni incorrect. Il a un certain comportement et c'est à l'utilisateur de spécifier ce qu'il entend par un comportement correct.

Un protocole qui s'arrête et n'évolue plus est soit terminé, soit bloqué (*deadlock*). Mais c'est à l'utilisateur de spécifier au vérificateur que cet état n'est pas l'état dans lequel le protocole devrait s'arrêter. Il en va de même pour le bouclage et autres phénomènes. Dans la suite, nous allons décrire comment un utilisateur peut spécifier le comportement que son protocole *devrait* avoir.

2.2 Assertions

L'instruction `assert` est toujours exécutable. Elle prend en argument une condition booléenne, par exemple `assert(state==1)`. S'il existe un état possible du protocole dans lequel, lorsque `assert` est exécutée, la variable `state` ne vaut pas 1, alors le vérificateur lance une erreur, et présente le scénario qui conduit à cette situation.

2.3 Les deadlocks et les labels `end`

Lorsqu'un processus se termine simplement en arrivant à sa dernière accolade, alors on considèrera par défaut qu'elle s'est terminé de manière correcte.

Mais dans certains cas, la fin du processus ne correspond pas à cette dernière accolade. Dans ce cas, le vérificateur supposera, a priori, qu'il s'agit d'une fin incorrecte, lancera un message d'erreur et montrera un scénario menant à cet état d'interblocage. Pour spécifier qu'un tel état n'est pas un interblocage, mais un état de fin correct, vous devrez mettre un label `end :` devant l'instruction à laquelle le processus s'arrête.

2.4 Les labels `progress`

Pour spécifier au vérificateur qu'un dérooulement correct du protocole doit toujours repasser par un point précis, on devra mettre un label `progress` devant ce point.

Ainsi, pour le vérificateur, toute boucle infinie ne contenant pas cette instruction sera considérée comme une erreur. Le vérificateur montrera un scénario menant à cette situation.

2.5 Les boucles et les labels `accept`

Un bouclage peut parfaitement être un comportement correct. Mais pour spécifier qu'un bouclage n'est *pas* un comportement correct, on pourra utiliser un label `accept`.

Ainsi, pour le vérificateur, toute boucle infinie contenant cette instruction sera considérée comme une erreur de protocole. Le vérificateur montrera un scénario menant à cette situation de bouclage.

3 Manips

Dans un premier temps, il s'agira de se familiariser avec *spin* et *xspin*. Ensuite, il s'agira de modéliser en *Promela*, les protocoles 1, 2 et 3 et 4, vus en TD.

Les instructions et les manips seront définies plus précisément dans un autre document.