Modélisation et Spécification – Master 2 LC
TD 3 : Spécification en LTL et l'outil Spin

Mihaela Sighireanu
(www.liafa.jussieu.fr/~sighirea/cours/modspec/)

Le but de ce TD est d'utiliser comme formalismes :
- pour la modélisation, les automates décrits en Promela (le langage de l'outil Spin), et
- pour la spécification, la logique LTL.

L'outil Spin permettra de vérifier que les modèles écrits en Promela satisfont des propriétés LTL.

Normalement, vous connaissez déjà Spin et Promela (cours "Concurrence" en M1 II). Toutefois, une présentation rapide est faite à la fin de ce document.

1 Exercices

Exercice 1 : *Ascenseur*

Les exemples donnés dans la section ?? permettent de modéliser un ascenseur pour trois étages.

1. Complétez la modélisation du processus "`porte`".
2. Simulez la modélisation obtenue.
3. Ecrivez en LTL la propriété "p_1 : *si la porte du première étage est ouverte, elle sera fermée l'état suivant*".
4. Ecrivez en LTL la propriété "p_2 : *inévitablement, une des trois portes sera ouverte*".
5. Vérifiez les deux propriétés en utilisant Spin.
6. Pourriez-vous spécifier (en LTL) d'autres propriétés de votre modèle ?

Exercice 2 : *Carrefour*

Un carrefour à deux voies orthogonales (EstWest et NordSud) est contrôlé par deux paires de feux tricolores (rouge, jaune, vert).

1. Donner une modélisation en Promela d'un seul feu qui fonctionne selon la législation française (les états rouge, jaune et vert sont exclusifs).

2. Exprimer en LTL la propriété : "Les feux s'allument toujours dans l'ordre vert, jaune, rouge et puis vert, etc. avec un seul feu allumé à la fois."

   (*Ind.* Utiliser comme prédicats v pour feu vert allumé, j pour le feux jaune, etc.)
3. Utiliser SPIN pour vérifier cette propriété sur votre modèle.
4. En utilisant les canaux de communication synchrones, modéliser un ensemble de deux feux qui contrôlent le carrefour.
5. Exprimer en LTL la propriété qui décrit le comportement correct du carrefour.
6. Vérifier cette propriété avec SPIN.

<u>Exercice</u> 3 : *Traduction simultanée Français/LTL*

Exprimer dans un Français ordinaire, les propriétés LTL suivantes :

1. $(\texttt{<>}r \rightarrow (p\ U\ q))$
2. $\texttt{[]}(q \rightarrow \texttt{[]}\neg p)$

Exprimer en LTL les phrases suivantes. Si la phrase est ambiguë, ditez comment vous l'avez eclaircie et pourquoi.

1. "p est vraie"
2. "p devient vraie avant r"
3. "Aucune autre commande de café n'est acceptée entre l'acquittement de la somme due et l'enlèvement du gobelet"
4. "p a lieu au plus une fois"
5. "p a lieu au plus deux fois"
6. "Au plus deux erreurs sur le mot de passe sont acceptées entre le login et la connexion"

2 Introduction rapide à Spin et Promela

Pour plus de détails, consulter les manuels de SPIN *et* PROMELA *disponible à* `www.spinroot.com`*.*

SPIN est un outil pour la vérification et la simulation des systèmes concurrents, en particulier les protocoles de communication.

Pour être étudié, un système est d'abord décrit en PROMELA (*Protocol Meta Language*), le langage de modélisation de SPIN. Promela est un langage impératif qui se ressemble au langage C agrémenté de quelques primitives de synchronisation. Pour communiquer, les processus peuvent utiliser des canaux de communication *fifo*, prendre rendez-vous ou utiliser des variables partagées.

SPIN comporte essentiellement deux modes :

- *Simulation* : le modèle est exécuté pas à pas, ce qui permet de se familiariser avec son comportement.
- *Vérification* : les états du modèle sont explorés exhaustivement pour vérifier que le modèle satisfait des propriétés exprimées, par exemple, en LTL.

Un programme PROMELA est une liste de déclarations de *variables*, de *canaux* et de *processus*.

Déclaration de variables : On indique le type (un des `bit`, `byte`, `short`, or `int`), le nom de la variable et optionnellement sa valeur initiale.

```
bool b1 = false, b2 = false;
bit k = 0;
```

Les variables de type tableau sont déclarées comme en C, par exemple :

```
bit porteouverte[3];
```

Déclaration de canaux : Il s'agit de cannaux *FIFO*. On l'introduit par le mot clé **chan**, suivi du nom du canal et optionnellement de sa longueur et du type des messages qui circulent. Par exemple :

```
chan Ouvreporte=[0] of {byte, bit},
     Transfert=[2] of {bit, short, chan};
```

`ouvreporte` est un canal *synchrone*, car sa longueur est 0, ce qui correspond à un rendez-vous ; sur ce canal circulent des messages ayant une partie `byte` et une partie `bit`. `Transfert` est un canal *asynchrone*, car il peut stocker (au plus) deux messages.

Déclaration de processus : La forme la plus simple de déclaration de processus est :

```
proctype nom ( paramètres_formels )
{ instructions }
```

Un processus est instancié en utilisant l'instruction `run` :

```
run nom ( paramètres_actuels )
```

Par exemple :

```
proctype porte (byte i) {
 ...
}
proctype ascenseur () {
 ...
}
```

Si le processus n'a pas des paramètres formels, une autre façon de l'instacier est de le déclarer "`active`". Ainsi, on peut déclarer des tableaux de processus "actifs". La position du processus dans le tableau est donnée par la variable pré-définie `_pid`. On peut donc écrire :

```
active [3] proctype porte () {
 ...
}
active proctype ascenseur () {
 ...
}
```

Opérations sur les canaux : Sur un canal on peut envoyer (opération "`!`") ou recevoir (opération "`?`") des messages. Par exemple :

```
ouvreporte!i,0;
ouvreporte?i,1;
ouvreporte?eval(etage),1
```

La fonction "`eval`" force l'égalité des valeurs reçues avec `etage`, la variable `etage` n'est pas changée.

Expressions : Un expression peut être utilisée comme une instruction si elle ne fait pas des effet de bord (opérations "`--`" et "`++`" de C). Dans ce cas, elle est exécutable quand sa valeur devient vraie (par le changement des valeurs des variables partagées). Par exemple :

```
(a == b);
```

est équivalent à :

```
while (a != b) skip;
```

Instruction “`atomic`” : L’exécution d’une séquence d’instructions préfixée par “`atomic`” peut être rendue indivisible, c’est-à-dire sans l’entrelacement des actions des autres processus. Par exemple :

```
atomic { (a == b) -> a = b + 1 }
```

Instruction “`if`” : Un branche de l’instruction “`if`” est exécutable si la première instruction de la branche, appelée aussi sa garde, est vraie. L’instruction “`if`” bloque jusque à ce qu’une branche devient exécutable, dans quel cas la branche est exécutée. Si plusieurs branches sont exécutables, l’une est choisie aléatoirement (*non-déterminisme*).

L’exemple suivant incrémente ou décrémente la valeur de `count` une fois.

```
if
:: count = count + 1
:: count = count - 1
fi
```

Instruction “`do`” : Similaire à l’instruction “`if`”, sauf que l’instruction est exécutée et puis la sélection est répétée jusque à l’exécution d’une instruction “`break`”.

```
proctype ascenseur () {
    show byte etage = 1;
    do
    :: (etage != 3) -> etage++
    :: (etage != 1) -> etage--
    :: ouvreporte!etage,1; ouvreporte?eval(etage),0
    od
}
```

Instruction “`init`” : Si aucun processus n’est déclaré actif, l’exécution du système commence par le processus `init` (c’est le `main` de Promela). Par exemple :

```
init {
    run porte(1); run porte(2); run porte(3);
    run ascenseur()
}
```