Théorie et pratique de la concurrence – Master 1 II
TP 1 : Spin et Promela

Spin est un outil pour la vérification et la simulation des systèmes concurrents finis (`http://spinroot.com`).

Pour être étudié, un système concurrent est d'abord décrit en Promela (*Protocol Meta Language*, voir `http://spinroot.com/spin/Man/promela.html`), le langage de modélisation de Spin. Promela est un langage impératif qui se ressemble au langage C agrémenté de quelques instructions pour la concurrence.

Le modèle Promela peut ensuite être analysé avec Spin par :

- *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 (par exemple, exclusion mutuelle) spécifiés en LTL (*Linear Temporal Logic*).

Pour une introduction rapide à Spin et son interface graphique XSpin, voir `http://spinroot.com/spin/Man/GettingStarted.html`

Promela

Un programme Promela est une liste de déclarations de *types*, *constantes*, *variables* (globales) et *processus*, suivie d'un point d'entrée du programme (`init`).

Déclaration de types : Les types de base de Promela sont : `bit` ou `bool`, `byte`, `short` et `int` ; ces types n'ont pas besoin d'être déclarés. Le seul type que l'utilisateur peut être déclaré est le type énumérée `mtype` pour lequel on peut spécifier ses valeurs (constantes symboliques).

Exemple : `mtype = { ack, nak, err };`

Déclaration de constantes : Promela permet les macro-définition à la C. Il est donc possible de déclarer des constantes comme en C avec `#define`, mais aussi des macros plus compliquées.

Déclaration de variables globales : Les variables scalaires ou tableau de Promela sont déclarées comme en C : on indique le type, le nom de la variable et optionnellement sa valeur initiale.

```
bool b1 = false, b2 = false;
bit k = 0;
bit porteouverte[3];
```

Une classe spéciale de variables sont les *canaux*, qui représentent des queues FIFO d'une longueur constante et dont les éléments (appelés messages) sont des tuples (typés) de valeurs scalaires ou vecteur de Promela. Les canaux sont déclarés en utilisant 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 }
```

où les paramètres formels sont déclarés comme en C mais séparés par des point-virgules.

Un processus peut être lancé de deux manière. La *manière implicite*, qui peut s'appliquer que si la liste de paramètres formels est vide, consiste à préfixer `proctype` par le mot clé `active`. Il est même possible de lancer un nombre fixe de copies du processus en utilisant la manière implicite. Par exemple :

```
active proctype ascenseur () { /* un processus 'ascenseur' lancé */
...
}
active [3] proctype porte () { /* trois processus 'porte' lancés */
...
}
```

La *manière explicite*, consiste à utiliser l'instruction `run` dans le point d'entrée du modèle (voir `init`) ou dans un autre processus :

```
run nom ( paramètres_actuels )
```

Le corps des processus est un liste de déclaration de variables (scalaires, tableau ou canal) suivie d'une liste d'*instructions* séparés par point-virgule. *Attention, le caractère "`;`" est séparateur d'instructions et non fin d'instruction comme en C !*

Instructions : Promela s'inspire peu de la syntaxe du C pour les instructions. Les points communs avec le C sont : l'affectation "`=`" et le test d'égalité de valeurs "`==`", l'utilisation des accolades pour délimiter les blocs d'instructions, la syntaxe des opérations sur les bits et les entiers (`++`, `--`, etc.), l'utilisation des étiquettes sur les lignes et l'instruction `goto`, l'utilisation des expressions comme instructions.

Dans ce dernier cas, une expression peut être utilisée comme une instruction si elle ne fait pas des effet de bord ; alors elle est exécutable quand sa valeur devient vraie (par le changement des valeurs des variables partagées). Par exemple, dans le code suivant :

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

l'exécution reste bloquée à la ligne 2 tant que `a` et `b` ont des valeurs différentes (la valeur d'une de ces variables peut être changée par des processus parallèles).

L'instruction vide est `skip`.

L'affectation, l'instruction vide, l'expression et le lancement d'un processus sont des instructions atomiques(indivisibles, exécutées sans entrelacement avec les autres processus) en Promela. Les autres instructions ne le sont pas. Pour rendre atomique un bloc d'instructions, on doit utiliser l'instruction "`atomic`". Par exemple :

```
atomic {
     (state==1); state = state+1
}
```

permet d'obtenir toujours une valeur égale à 2 pour `state`.

La plus simple instruction de sélection est l'instruction "`if`", qui s'approche plus au niveau de la syntaxe du `switch` de C, avec une sémantique différente. En effet, contrairement au `switch`, le test des différents cas est fait de façon non-déterministe (aléatoire) et si aucune branche ne peut être sélectionnée, l'exécution est bloquée en attente de l'activation d'une sélection. La syntaxe générale est :

```
if
:: sequence_d_instructions_1
:: ...
:: sequence_d_instructions_N
fi
```

Par exemple, le code suivant incrémente *ou* décrémente la valeur de `count` une fois :

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

Pour obtenir une sélection en fonction d'une expression booléenne, la première instruction de la séquence doit être une expression booléenne, par exemple :

```
if
:: (count % 2 == 1) -> count = count + 1 /* syntaxe dérivée */
:: (count % 2 == 0); count = count - 1
fi
```

En effet, une instruction C "`if (cond) stat`" peut être codé également par "`cond -> stat`" en Promela.

L'instruction d'itération "`do`" a une syntaxe similaire au "`if`", sauf qu'après l'exécution de l'instruction sélectionnée le contrôle passe au début de l'instruction `do` jusqu'à l'exécution d'une instruction "`break`". Par exemple, le code suivante aura une exécution infinie :

```
proctype ascenseur () {
    show byte etage = 1; /* show: faire apparaître la valeur à la simulation */
    do
    :: (etage != 3) -> etage++
    :: (etage != 1) -> etage--
    od
}
```

Sur un canal, on peut envoyer (opération "`!`") ou recevoir (opération "`?`") de 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.

D'autres instructions sont à découvrir sur `http://spinroot.com/spin/Man/promela.html` et elles seront introduite au fur et à mesure de besoins.

Point d'entrée du modèle : L'exécution du système commence par la section `init` qui est exécutée en parallèle avec les éventuels processus actifs. Le corps de la section "`init`" est le même que celui des processus : une liste de déclarations de variables locales et une liste d'instructions. Par exemple :

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

Spin et XSpin

Nous utiliserons l'interface graphique de Spin, appelée XSpin. L'appel est fait en ligne de commande :

```
> xspin &
```

XSpin comporte un (mauvais) éditeur de Promela, ainsi qu'une interface graphique aisée par le menu `Run` avec les multiples fonctions de l'outil Spin.

Pour la simulation : `Run` → `Set Simulation Parameters...`.

Pour la vérification : `Run` → `LTL Property Manager...`.

Propriétés LTL : Les opérateurs LTL disponibles sont ceux listés dans l'interface : les opérateurs booléens (l'implication est notée →), opérateur globalement (notation `[]`), l'opérateur dans le futur (notation `<>`) et l'opérateur *until* (notation `U`). *Attention, l'opérateur next n'est pas à utiliser pour avoir des bonnes performances !*

Les opérandes doivent être des propositions atomiques. Pour définir des propositions atomiques, utilisez des macros C dans la section *Symbols Definition*. Vous pouvez ici faire référence aux variables globales du modèle, aux variables locales des processus (en préfixant par le nom du processus, par exemple `porte[2].etage`), ou à une étiquette dans le corps d'un processus (par exemple `porte[2]@l1`).

Vous pouvez/devez sauvegarder les formules écrites dans des fichiers à extension `.ltl` (c'est pas le même fichier que la spécification Promela, dont l'extension consacrée est `.pml`).

Exercice 1: Soit l'algorithme suivant pour l'exclusion mutuelle :

```
int turn;

P1:
Loop forever:
p1: section NC
p2: await turn==1
p3: section C
p4: turn:=2

P2:
Loop forever:
q1: section NC
q2: await turn==2
q3: section C
q4: turn:=1
```

- Codez cet algorithme en Promela en utilisant des étiquettes pour les sections non-critiques et critiques.
- Simulez votre modèle.
- Ecrivez en LTL la propriété "*au plus un processus se trouve dans la section critique*".
- Vérifiez cette propriété sur votre modèle.

Exercice 2:

Codez l'algorithme de Dekker vu en cours et démontrez l'absence de famine (regarder ce qui se passe lorsque vous cochez ou pas la case *with weak fainess* lors de la vérification de la formule LTL).