Promela - résumé

M1
TP SVL – 2011-2012

Spin est un outil pour analyser la consistance logique des systèmes concurrents, spécifiquement des protocoles de communication échangeant des données.

Les spécifications Spin sont écrites en Promela (Process Meta Language). Une spécification décrit un ensemble de processus qui communiquent. La communication entre processus peut-être :

– synchrone (ex : rendez-vous)
– asynchrone (par tamponnage)

On donne ici les principaux aspects de Promela.

1 Types de données

`bit` ou `bool`
`byte` (0 à 255, non signé)
`short` : -2^{15-1} .. 2^{15-1}
`int` : -2^{31-1} .. 2^{31-1}
Un type énuméré appelé `mtype` : Ex :

```
mtype = { ack, nak, err, next, accept }
mtype maVar = arr;
```

NB : le choix du type est important : pour réduire l'espace des états on choisira le type qui convient possédant le moins de valeurs possibles (ex `byte` plutôt que `int`).
Tableaux :

```
byte state[N]
state[0] = state[3] + 5 * state[3*2/n]
```

Structures :

```
typedef <nomStructure> {
   <typeChamp1> <nomChamp1>;
   <typeChamp2> <nomChamp2>;
...
}
```

Par exemple :

```
typedef Field {
  short a;
  byte b;
}
```

puis on déclare : `Field maStruct;` et on affecte/accède : `maStruct.a = 1;`

Pour afficher le contenu d'une variable on utilise un affichage à la C :

```
printf("x = %d\n", x)
```

2 Processus

Un processus représente un automate à nombre fini d'états. Une spécification Promela est souvent constituée de plusieurs processus qui communiquent. Le corps d'un processus est une suite d'instructions de style impératif, avec en plus la possibilité d'écrire et lire dans des canaux de communication.

2.1 Exécutabilité d'une instruction

En Promela il n'y a pas de différence entre condition et instruction : on peut utiliser une condition booléenne comme une instruction qui peut être :

– bloquante si la condition est fausse ;
– exécutable si la condition est vraie.

L'exécution d'une instruction est donc conditionnée par son exécutabilité, ce qui est la brique de base pour la synchronisation entre processus. Une instruction peut être soit bloquée, soit exécutable. Par ex, les 2 instructions qui suivent sont équivalentes :

```
while (a != b)
      skip /* en C, wait for a==b */

(a == b) /* en Promela, wait for a==b */
```

Une condition booléenne est bloquante tant qu'elle est fausse. Une affectation et une déclaration sont toujours exécutables.

2.2 Déclaration

Le comportement d'un processus est défini dans une déclaration `proctype`.

```
proctype A() {
  byte state; // variable locale
  state = 3 // affectation
}
```

Le `;` sert de séparateur, et pas de terminateur. Une autre syntaxe possible (et totalement équivalente) est le `->`, moyen informel d'indiquer une relation de cause à effet entre 2 instructions.

```
mtype = {tourA,tourB };
mtype state = tourA; // globale

proctype A() {
  state == tourA -> printf("A"); state := tourB
}

proctype B() {
     state == tourB -> printf("B");
}
```

C'est forcément A qui s'exécutera en premier, B étant bloqué par le test `state == tourB`.

2.3 Instanciation

La déclaration ne lance pas l'exécution d'un processus, qui doit se faire explicitement. Initialement, le seul processus à s'exécuter est un processus particuler appelé `init`. La plus petite spécification Promela est `init { skip }`.

On peut aussi exécuter un processus en indiquant son nom par l'instruction run :

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

`run` peut toujours s'exécuter, sauf si trop de processus sont déjà en train de tourner.

Les processus lancés s'exécutent en parallèle, avec une *sémantique d'entrelacement non déterministe* : si 2 processus peuvent s'exécuter, l'un des 2 sera choisi de manière non déterministe. Si dans un même processus plusieurs instructions sont exécutables, l'une d'elles sera choisie de manière non déterministe.

`run` peut aussi servir à passer des valeurs (pas de type tableau) pour des processus paramétrés.

```
proctype A(byte init) {  state = init; state = state - 1 }
init {  run A(3) }
```

On peut aussi créer autant d'instances de A que souhaité.

3 Instructions de contrôle

3.1 Condition

```
if
:: garde1 -> option1
:: garde2 -> option2
...
:: else -> optionElse
fi
```

Si plusieurs gardes sont exécutables, alors l'une des branches est choisie de manière non déterministe. La branche `else` (optionnelle) est exécutable seulement quand aucune autre branche n'est exécutable. La branche else n'est pas atomique : le processus peut perdre la main après avoir passé le `else` mais avant d'avoir exécuté l'`optionElse`. L'option ne pourra être exécutée que si la garde est exécutable, mais l'exécution de l'option et de la garde n'est pas atomique. Le if termine quand l'une de ses branches a terminé.

On peut utiliser des branches mutuellement exclusives, ou non :

```
if                          if
  :: (a != b) -> option1      :: printf("1");
  :: (a == b) -> option2      :: printf("2");
fi                          fi
```

3.2 Répétition

Même syntaxe avec `do... od`. Même sémantique, mais en répétant tant que `break` n'a pas été exécuté.

```
byte count;
proctype counter() {
do
  :: count = count + 1
  :: count = count - 1
  :: (count == 0) -> break
od
}
```

Dans le cas d'une exécution avec `init {run counter() }`, le processus termine avec `count` valant 0, au bout d'un nombre non déterministe d'incrémentation et de décrémentations.
Dans le cas d'une exécution avec `init {run counter(); run counter();} count` peut valoir 0 sans pour autant faire terminer la boucle, les branche "décrémentation" et "incrémentation" de l'autre processus étant toujours exécutables. À la sortie de la boucle le compteur ne vaudra donc pas forcément 0.

3.3 assert

`assert(booleanCondition)`
Toujours exécutable, indique une erreur si la condition est fausse.

3.4 atomic

La primitive `atomic` englobe une suite d'instructions qui deviennient atomiques, cad qu'elles seront exécutées d'un bout à l'autre sans qu'un autre processus puisse prendre la main. Le `atomic` peut servir à implanter un "test and set".
Avec `:: atomic { (count == 0) -> break; }` , le processus `counter` ci-dessus ne peut être interrompu entre le moment où il teste `count` et la sortie de la boucle.

Attention : Dans les exercices qui vous sont demandé, vous ne devez jamais bloquer un processus dans un `atomic`. Spin prévoit que si un processus se trouve bloqué alors qu'il est en `atomic`, l'exécution est transférée à un autre processus. En particulier, lorqu'un rendez-vous est fait sous `atomic`, le contrôle est transféré au récepteur (fortement déconseillé).

4 Échange de messages

4.1 Communication asynchrone

Un canal est déclaré avec le type prédéfini `chan`.
Ex : `chan canal1 = [16] of { short }`
16 est la taille du canal `canal1`, qui permet de stocker des messages de type `short`.
Ex : `chan canal2 = [16] of { byte, int, chan, byte }`
permet de stocker des messages contenant 4 champs, dont un de type canal de communication.

La réception est bloquante quand le canal est vide. L'émission est bloquante quand le canal est plein.

Émission : `canal1!expr`, ou `canal2!expr1,expr2,x,expr4`

Réception :
- `canal1?msg` pour réception d'un message stocké dans une variable `msg` ;
- `canal1?_` pour réception d'une valeur quelconque, non conservée ;
- `qname2?_,expr2,y,_` pour la réception d'un message à 4 champs, le 1er et le 4ème quelconques et non conservés, le 3ème quelconque et stocké dans la variable `y`, le 2nd devant être égal à la valeur de `expr2` pour que la réception soit exécutable.

4.2 Communication par rendez-vous

Même syntaxe, mais le canal est déclaré de taille 0, et l'émission et la réception sont simultanés.