# Chapitre 4

# Le langage PROMELA (PROcess Meta LAnguage)

## PROcess MEta Language [Holzmann]

- Spécification et vérification de protocoles
- Abstraction de l'implémentation (niveau conceptuel)
- Modèle de validation (« model checking »)

## SPIN (Simple Promela INterpreter)

⇒ http://spinroot.com/spin/Man/Manual.html

- Validation et vérification de propriétés
- Simulation : échanges de messages

# 1. Mécanismes du langage

## 1.1. Processus, canaux et variables

Processus :

Objets globaux spécifiant le comportement du protocole

Canaux de messages et variables :

Objets globaux ou locaux définissant l'environnement d'exécution des processus

## 1.2. Exécutabilité

- pas de différence entre conditions et instructions
- une instruction est soit exécutable soit bloquée

⇒ **mécanisme de synchronisation entre processus**

Ex : while (a!=b) skip  s'écrit :  (a==b)

# 1.3. Types de données

- Types de base : **bit, bool, byte, short, int**
  - une valeur unique à un instant donné
- Canaux : **chan**
  - peuvent stocker simultanément plusieurs valeurs (messages)

**Table 1 - Data Types**

| Typename | C-equivalent | Macro in limits.h | Typical Range |
|---|---|---|---|
| bit or bool | bit-field | - | 0..1 |
| byte | uchar | CHAR_BIT (width in bits) | 0..255 |
| short | short | SHRT_MIN..SHRT_MAX | -2^15 - 1 .. 2^15 - 1 |
| int | int | INT_MIN..INT_MAX | -2^31 - 1 .. 2^31 - 1 |

- Tableaux : comme en C, rangs de O à N-1
  - byte state[N]
  - state[0] = state[3] + 5 * state[3 * 2/n]

# 1.4. Types de processus

Déclaration d'un type de processus : **proctype**

– définit le comportement d'un processus
– celui-ci sera instancié lors de l'exécution

Ex : proctype A() { byte state ; state = 3 }

– Le séparateur **;** est synonyme du séparateur → (causalité)

Ex :

byte state = 2 ;
proctype A() { (state == 1) → state = 3 }
proctype B() { (state = state - 1 }

Exercice : comparer les processus de type A et B en termes d'exécutabilité

# 1.4.1. Instanciation des processus

- Initialement un seul processus s'exécute : **init**

  Ex :

  ```
  init { skip }

  init { printf ("hello world\n") }

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

  – L'instruction **run** instancie une copie d'un processus de type donné

  – **run** retourne un entier positif (pid du processus instancié) ou 0 en cas d'échec (trop de processus dans le système)

  – **NB** : **pas de notion de « temps » en Promela** (comme dans un système distribué)

  ⇒ toutes les possibilités **d'entrelacement des exécutions** des processus sont analysées (voir exemple)

**NB** : seuls les canaux et les instances des 5 types de base peuvent être passés en paramètre (les tableaux et les types de processus ne peuvent pas être passés en paramètre)

Ex :

```
proctype A (byte state ; short set)
{ (state == 1) → state = set }
init { run A(1, 3) }
```

– L'instruction **run** peut aussi être utilisée dans les types de processus : un processus se termine quand il a achevé son corps d'exécution et que tous les processus qu'il a instanciés se sont terminés.

Exemple : **problème de l'exclusion mutuelle** lors de l'accès aux variables globales par plusieurs processus concurrents :

```
byte state = 1;
proctype A() { (state == 1) -> state = state + 1}
proctype B() { (state == 1) -> state = state - 1}
init { run A() ; run B() }
```

Question : déterminer les différents types de scénarios d'exécution.
Quelle est la valeur finale de la variable state ?

# Spécification de l'algorithme de Dekker en Promela [Holzmann]

```
#define true 1
#define false 0
#define Aturn false
#define Bturn true

bool x, y, t;

proctype A()
{   x = true;
t = Bturn;
(y == false || t == Aturn);
/* critical section */
x = false
}
```

```
proctype B()
{   y = true;
t = Aturn;
(x == false || t == Bturn);
/* critical section */
y = false
}

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

Exercice: montrer que cet algorithme assure l'exclusion mutuelle (TD-TP)

# 1.4.2. Séquences atomiques

Séquence d'instructions qui sont exécutées de façon **indivisible**, i.e. **non entrelaçable** avec d'autres processus

Exercice : reprendre l'exercice précédent avec des séquences atomiques. Quelle est la valeur finale de la variable *state* ?

```
byte state = 1;
proctype A() { atomic { (state == 1) → state = state + 1} }
proctype B() { atomic { (state == 1) → state = state - 1} }
init { run A(); run B() }
```

Les séquences atomiques sont un outil important de **réduction de la complexité** dans un modèle de validation, car elles diminuent le nombre d'entrelacements possibles

# 1.5. Canaux de messages

Les canaux modélisent le transfert de messages entre les processus.

Ex : chan a, b ; chan c[3] /* tableau de canaux */

- On peut préciser le nombre maximum de messages que le canal peut contenir

Ex : chan c[3] = [4] of {byte}

- On peut déclarer des messages à plusieurs champs de types différents.

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

Instructions **d'envoi** et de **réception** de messages sur un canal :

**qname ! expr**

⇒ envoie la valeur de l'expression *expr* vers le canal *qname*. Le message est rajouté à la queue de la file du canal.

**qname ? msg**

⇒ reçoit un message du canal *qname* et le stocke dans la variable msg.

• Les canaux transmettent les messages dans l'ordre **FIFO**.

• Pour les messages à plusieurs champs, la notation est la suivante :

qname ! expr1, expr2, expr3

qname ? var1, var2, var3

- Le premier champ d'un message est souvent utilisé pour préciser le type du message (ex : ack, nak, data, etc.) :

  **qname ! expr1(expr2,expr3)**

  **qname ? var1(var2,var3)**

- L'opérateur **len(qname)** retourne le nombre de messages présents dans le canal qname.

Remarque : les opérations d'envoi et de réception sur la canal ne peuvent pas être évaluées sans effets de bord potentiels :

qname ? ack(var)

Si on veut tester la présence d'un message *ack(var)* sur le canal, l'instruction précédente va retirer le message du canal.

⇒ On peut éviter l'effet de bord en utilisant l'instruction suivante :

**qname? [ack(var)]**

→ retourne 1 si l'instruction qname?ack(var) est exécutable, i.e s'il y a un message *ack(var)* sur le canal et retourne 0 sinon.

→ le message *ack(var)* n'est pas retiré du canal

Remarque : dans les séquence non-atomiques suivantes :

(len(qname) < MAX) → qname ! msgtype

qname?[msgtype] → qname ? msgtype

⇒ La deuxième instruction n'est pas forcément exécutable lorsque la première a été exécutée. En effet, si le canal est partagé par plusieurs processus, un autre processus peut écrire ou lire un message dans le canal, avant que le processus exécute la deuxième instruction.

Exercice : dans le programme suivant, quelle est la valeur affichée par le processus de type B ? **[Holzmann]**

```
proctype A(chan q1)
{       chan q2;
        q1?q2;
        q2!123
}

proctype B(chan qforb)
{       int x;
        qforb?x;
        printf("x = %d\n", x)
}

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

# 1.6. Communication par rendez-vous

- Communication **asynchrone** entre processus :

chan qname = [N] of {byte}

- Communication par rendez-vous ou **synchrone** :

chan port = [0] of {byte}

⇒ Le message ne peut pas être stocké dans le canal et doit être livré instantanément

Remarque : la communication par rendez-vous est **binaire** ou **point à point.**

Exercice : comparer les exécutions des processus de type A et B, suivant que la taille du canal *name* soit **0, 1 ou 2**

**-> voir sur écran partagé**

```
#define msgtype 33

chan name = [0] of { byte, byte };

proctype A()
{       name!msgtype(124);
        name!msgtype(121)
}

proctype B()
{       byte state;
        name?msgtype(state)
}

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

# 2. Structures de contrôle

3 structures de contrôle :

- **Sélection**
- **Répétition**
- **Saut inconditionnel**

## 2.1. Sélection

```
if /* ici sémantique du « if » classique : 1 seule alternative
      exécutable */
:: (a != b)  → option1
:: (a == b) → option2
fi
```

# Exemple de sélection [Holzmann]

```
#define a 1
#define b 2

chan ch = [1] of { byte };

proctype A()
{       ch!a
}

proctype B()
{       ch!b
}

proctype C()
{       if
        :: ch?a
        :: ch?b
        fi
}

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

# Exemple de sélection

**/* ici sémantique différente du « if » classique :**
**les 2 alternatives sont exécutables ⇒ choix aléatoire */**

```
byte count;
proctype counter ()
{ if
  :: count = count + 1
  :: count = count – 1
 fi
}
```

Remarque : les 2 branches du « if » sont exécutables

# 2.2. Répétition

Pour sortir de la boucle on utilise l'instruction **break**

Exemple :

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

Exercice : le processus dans l'exemple se termine-t-il nécessairement ? Sinon proposer une modification pour forcer la terminaison quand le compteur atteint 0.

# Exemple du sémaphore de Dijkstra [Holzmann] (TD-TP)

```
#define p 0
#define v 1

chan sema = [0] of {bit};

proctype dijkstra()
{


 do
 :: sema!p ->
    sema?v;
 od
}

proctype user()
{
 do
 :: sema?p -> /* critical section */
    sema!v; /* non critical section */
 od
}


init{atomic {run dijkstra();
              run user(); run user();run user()} }
```

# 2.3. Saut inconditionnel

Le saut inconditionnel permet aussi de sortir d'une boucle : **goto**

Exemple : algorithme d'Euclide de recherche du PGCD de deux entiers positifs

```
proctype Euclide (int x, y)
{ do
  :: (x > y) → x = x- y
  :: (x < y) → y = y -x
  :: (x == y) → goto done
  od ;
done : printf ("pgcd : %d\n", x)
}
```

# 3. Procédures et récursion

Les procédures récursives peuvent se modéliser par des processus.

Exemple de factorielle **[Holzmann]**:

**-> vérifier avec ispin : “test_fact.pml”**

```
proctype fact(int n; chan p)
{       chan child = [1] of { int };
        int result;

        if
        :: (n <= 1) -> p!1
        :: (n >= 2) ->
                run fact(n-1, child);
                child?result;
                p!n*result
        fi
}
init
{       chan child = [1] of { int };
        int result;

        run fact(7, child);
        child?result;
        printf("result: %d\n", result)
}
```

# 4. Types de messages

Exemple :

**mtype = { ack, nak, err, next, accept }**

Ceci équivaut à la définition :

```
#define ack 1
#define nak 2
#define  err 3
#define next 4
#define accept 5
```

**Exemple : modélisation du protocole de Lynch [Holzmann] (TP)**

# 5. Critères de correction des protocoles

## Objectifs de PROMELA :

- Modéliser le **comportement des processus** à un haut niveau d'abstraction
  - **modèle de validation** du protocole (« model checking »)
  - définir des **critères de correction**
    - respect des invariants du système
    - pas de « deadlocks »
    - pas de « livelocks » (mauvais cycles)
    - pas de terminaisons incorrectes
- PROMELA repose sur un modèle **d'automates à états finis :**
  - vérifications **automatiques** (ex : absence de « deadlocks »
  - vérifications de **propriétés** formulées par le programmeur

  (ex : invariants) **=> labels, assertions**

# 5.1. Assertions

Un critère de correction peut s'exprimer sous forme d'une **condition booléenne** qui doit être satisfaite dans un état donné :

**assert (condition)** : toujours exécutable et peut être placée n'importe où dans un code PROMELA

- si la condition est vraie : l'instruction « assert » est sans effet
- l'instruction « assert » est violée s'il existe au moins une séquence d'exécution telle que la condition est fausse lorsque l'instruction « assert » devient exécutable.

**Exemple d'exclusion mutuelle (slide 8) :**

```
byte state = 1;
proctype A() { (state == 1) -> state = state + 1}
proctype B() { (state == 1) -> state = state - 1}
init { run A() ; run B() }
```

**Si on veut exprimer que quand un processus de type A() (resp. B()) finit son exécution la valeur de la variable « state » est 2 (resp. 0) :**

```
byte state = 1;
proctype A() { (state == 1) -> state = state + 1 ;
   assert(state == 2)}
proctype B() { (state == 1) -> state = state - 1 ;
   assert(state == 0)}
init { run A() ; run B() }
```

**Remarque : ces assertions sont fausses**

**-> vérifier avec ispin : « test_assert.pml »)**

# 5.2. Invariants du système

Une application plus générale du « **assert** » est la formalisation des **invariants du système** (conditions booléennes qui restent vraies dans tous les états atteignables du système) :

**proctype monitor() { assert (invariant) }**

- ce processus s'exécute indépendamment des autres
- avec **l'entrelacement**, l'instruction « assert » est **exécutable** à tous les états du système
- l'invariant peut être vérifié à tout moment

# Exemple du sémaphore de Dijkstra (slide 18) :

```
byte count ; chan sema = [0] of {bit} ;
proctype user()
{ skip ;
  sema?p;
  count = count + 1;
  skip;   /* section critique */
  count = count - 1;
  sema!v; /* sortie de la section critique */
  skip;
}
proctype monitor() {assert(count == 0 || count == 1) }

init
{atomic {run dijkstra() ; run monitor();
         run user(); run user(); run user()}
}
```

# 5.3. Deadlocks (interblocages)

- Les séquences d'exécution :
  - ➔ soit se terminent
  - ➔ soit rebouclent sur un état précédent de la séquence

- **Tous les processus ne se terminent pas forcément :**

Ex : processus serveur (sémaphore)

- Tous les cycles infinis ne sont pas des « **deadlocks** » :

=> distinguer un état final « correct » d'un état final « incorrect »

- Un état final **incorrect** peut être un « deadlock » ou un état d'erreur du à une spécification incomplète du protocole

  (ex : réception d'un message non attendu)

- Un état final **correct** est donc tel que :

  tous les processus instanciés se sont terminés ou ont atteint

  un état étant marqué comme état final valide

- Un modèle de validation est donc correct s'il ne contient pas de séquence d'exécution avec un état final invalide

  **=> PROMELA permet d'indiquer qu'un état final doit être considéré comme correct, même s'il est dans une boucle :**

  **le label « end » marque un état comme étant un état final valide**

## Exemple

-> vérifier avec ispin : « test_Dijkstra_label_SC.pml »

```
proctype dijkstra()
{
end : do
      :: sema!p → sema?v
      od
}
```

➔ Le processus de type dijkstra est considéré dans un état final valide si cet état est étiqueté par « end »

Remarque : on peut utiliser plusieurs labels de type **« end-state »** dans un même processus : préfixe commun « end » (end0, end_world, etc.)

# « Non-progress » cycles

Pour définir l'absence de cycles infinis qui ne progressent pas, il faut pouvoir définir les états qui dénotent un progrès

- **le label « progress »** définit un état qui doit être atteint pour que le protocole progresse (ex : incrémentation d'un n° de séquence, réception d'une donnée, etc.)

**Exemple du sémaphore :**

```
proctype dijkstra()
{
end:        do
         :: sema!p →
progress:   sema?v
         od
}
```

Remarque : on peut utiliser plusieurs labels de type « progress » dans un même processus : préfixe commun «progress» (progress0, progress_is_slow, etc.).