Un Exemple de Langage Parallèle Asynchrone : PROMELA

Laure Petrucci

3 mars 1999

Table des matières

1 Introduction

PROMELA est un langage de spécification de systèmes parallèles asynchrones. Il permet de décrire des systèmes concurrents, en particulier des protocoles de communication. Ce langage autorise la création dynamique de processus. Les communications peuvent s'effectuer de différentes manières : partage de variables globales, envoi et réception de messages via des canaux de communication. Ces derniers peuvent être définis pour réaliser des communications tant synchrones (rendez-vous) qu'asynchrones (bufferisées).

Les interactions entre processus concurrents peuvent créer de nombreux problèmes tels que des interblocages, des modifications inattendues de valeurs de variables... La vérification de la correction d'un système décrit en PROMELA peut être réalisée grâce à l'outil de simulation et d'analyse *spin* (et son interface graphique *xspin*).

Spin simule l'exécution du système en faisant des choix aléatoires d'ordre d'exécution des différents processus, ou il engendre un programme C qui effectue une validation exhaustive par construction du graphe d'états. Lors des simulations et des validations, *spin* vérifie l'absence de blocage, cherche les réceptions de messages non spécifiées ainsi que les parties de code qui ne peuvent être atteintes—ni par conséquent exécutées. L'outil de validation peut également être utilisé pour vérifier des invariants du système et trouver des cycles d'exécution ne conduisant à aucune progression.

Le système de vérification de *spin* est basé sur le paradigme de la vérification à la volée. Des techniques de compression de la représentation en mémoire du graphe d'états peuvent être utilisées, qui permettent de construire des graphes de taille très importante.

2 Le langage PROMELA

Un programme PROMELA est constitué de *processus*, de *canaux* de communication servant à transmettre des messages, et de variables. Les canaux et variables peuvent être déclarés soit globalement, soit localement, à l'intérieur d'un processus. Les processus spécifient le comportement du système alors que les canaux et variables définissent l'environnement d'exécution.

En PROMELA, il n'y a aucune différence entre les conditions et les instructions. L'exécution d'une *instruction* ne peut avoir lieu que si celle-ci est exécutable. Une instruction est soit *exécutable* soit *bloquée*. Une *condition* ne peut être passée que si elle est satisfaite. Si ce n'est pas le cas, elle est bloquée jusqu'à ce qu'elle devienne vraie.

Les instructions sont *séparées* par un `;` . La dernière instruction, ne devant être séparée d'aucune autre, n'est pas suivie d'un `;` . Un autre séparateur existe : `->`. Tous deux sont équivalents, mais la flêche peut contribuer à la lisibilité du programme en indiquant une causalité.

2.1 Les types de données

Les données utilisés en PROMELA peuvent être d'un type de base ou d'un type plus complexe (i.e. structure, tableau...), défini à partir des types de base.

2.1.1 Les types de base

Les caractéristiques des types de bases sont décrites dans le tableau ci-dessous. La déclaration d'une variable se fait de la manière suivante :

```
byte mavar1;
int mavar2 = 4;
```

nom	*taille*		*domaine*
bit	1	unsigned	0,1
bool	1	unsigned	0,1
byte	8	unsigned	0..255
short	16	signed	$-2^{15}..2^{15}-1$
int	32	signed	$-2^{31}..2^{31}-1$
unsigned	x	unsigned	$0..2^{x}-1$

Le type `unsigned` ayant une taille spécifiée dans le programme, la déclaration d'une variable `mavar` de ce type et de longueur 3 a la forme :

```
unsigned mavar : 3;
```

2.1.2 Les tableaux

Les variables peuvent être déclarées comme des tableaux. Leur utilisation est similaire à celle des langages de programmation classiques.

Exemple 1

```
byte mavariable[4];
mavariable[0] = mavariable[1] + 1
```

La variable `mavariable` *est déclarée comme un tableau de 4* `byte`*. Les indices du tableau commencent à 0.*

2.1.3 Les structures

L'utilisateur peut définir ses propres structures nommées, à partir des types de base. Le mot réservé `typedef` permet d'introduire un nouveau nom pour désigner une structure, dont chaque champ a un type déjà défini. Ce nouveau type peut alors, à son tour, être utilisé pour déclarer et instancier de nouveaux objets.

Exemple 2

```
typedef Field{
    short f;
    byte g
    };
typedef Msg {
    byte a[3];
    int fld1;
    Field fld2;
    bit b
    };
Msg monmessage;

monmessage.fld2.f = 3
```

Deux types structurés sont définis. Le premier, `Field`*, est utilisé dans la définition du second,* `Msg`*. Ce dernier type est celui de la variable* `monmessage`*. La valeur* `3` *est affectée à l'un des champs.*

2.1.4 Les canaux de communication

Les canaux de communication permettent aux processus d'échanger des messages. Un canal peut contenir plusieurs messages, et chaque message peut être composé d'une ou plusieurs informations. Dans ce cas, la communication entre processus est asynchrone. Un canal peut également être déclaré comme ayant une capacité nulle. La communication se fait alors par rendez-vous.

Exemple 3

```
chan q1 = [5] of {int};
chan q2 = [2] of {int, chan};
chan q3 = [0] of {int};
```

Le canal `q1` *peut contenir au maximum 5 messages de type entier, et le canal* `q2` *2 messages composés chacun d'un entier et d'un canal. Le canal* `q0` *n'a pas de capacité de stockage des messages.*

Le nom d'un canal peut être transmis par des canaux de communication, ou passé en paramètre d'une instanciation de processus.

2.1.5 Les définitions de types de messages

Les types de messages peuvent être définis par des noms. Ils sont alors utilisés de manière symbolique au lieu de numérique.

Exemple 4

```
mtype = {ack, nak, err};
chan q = [4] of {mtype, bit};
```

Il y a trois types de messages : `ack`, `nak` *et* `err`. *Le canal de communication* `q` *peut contenir au maximum 4 messages composés d'un type et d'une valeur booléenne.*

Ces types peuvent être utilisés dans la déclaration de variables.

Exemple 5

```
mtype = {plein, moitie, vide};
mtype verre = plein;
```

2.1.6 La déclaration de processus

Le comportement d'un processus est spécifié dans sa déclaration. Celle-ci est effectuée en utilisant le mot réservé `proctype`.

Exemple 6

```
proctype A()
{byte state;

 state = 3
}
```

Un processus `A` *est déclaré, avec une variable locale* `state`. *Il affecte la valeur 3 à cette variable.*

2.1.7 Les constantes et macros

Les constantes peuvent être définies comme en C.

Exemple 7

```
#define vrai 1
#define faux 0
```

permet de définir deux constantes `vrai` *et* `faux`.

Les constantes `true` et `false` sont prédéfinies.

Il est possible, comme en C, de définir des macros, des drapeaux et d'inclure des fichiers. La syntaxe est la même qu'en C, à savoir :

```
#define nom chaine_de_caracteres
#define nom(arg1,...,argn) chaine_de_caracteres
#ifdef nom
#ifndef nom
#if codition
#else
#endif
#undef nom
#include "nomfichier"
```

2.2 Les instructions

Comme les langages structurés classiques, PROMELA dispose de différents types d'instructions : opérations permettant de définir des conditions et effectuer des calculs, ... ainsi que de fonctions spécifiques pour instancier les processus et utiliser les canaux de communication.

2.2.1 Les expressions

Les expressions peuvent être construites en utilisant les opérateurs suivants :

```
+ - * / %
> >= < <= == != !
&& ||
& | ~ >> <<
++ --
```

Ces opérateurs sont pour la plupart binaires. La négation ! et le moins `-` peuvent être soit binaires, soit unaires, suivant le contexte. Les incrémentation `++` et décrémentation `--` sont des suffixes unaires.

2.2.2 Les instanciations de processus

La définition d'un processus à l'aide de la primitive `proctype` spécifie le comportement d'un processus, mais ne lance pas son exécution. Au départ, un seul processus est exécuté : `init`. Ce processus initial peut initialiser des variables globales et instancier des processus. Ceci se fait par l'instruction `run`.

Exemple 8

```
byte state = 2;

proctype A()
{(state == 1) -> state = 3
}

proctype B()
{state = state - 1
}

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

Une variable globale `state` *est déclarée et initialisée à 2. Deux processus* `A` *et* `B` *sont définis, qui opèrent sur cette variable globale. Ils sont instanciés par le processus* `init`*. Le processus* `A` *attend que la condition soit satisfaite tandis que le processus* `B` *peut s'exécuter immédiatement.*

La primitive `run` peut également passer des paramètres au nouveau processus à créer.

Exemple 9

```
proctype A(byte state; short foo)
{(state == 1)->state = foo
}

init
{run A(1,3)
}
```

Un processus `A` *est initialisé, avec les paramètres 1 et 3.*

L'instruction `run` peut être utilisée dans n'importe quel processus pour en créer un nouveau. Un processus qui s'exécute disparaît lorsqu'il termine, c'est-à-dire lorsqu'il atteint la fin du code à exécuter, mais jamais avant les processus qu'il a lui-même créés. L'accès par plusieurs processus à une variable globale peut conduire à des résultats surprenants.

Exemple 10

```
byte state = 1;

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

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

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

Lorsque les deux processus ne testent pas simultanément la variable `state`*, l'un d'eux s'exécute, et l'autre reste bloqué. Lorsque les deux tests sont effectués en même temps, les deux processus se terminent normalement. La valeur de* `state` *est quelconque parmi 0, 1 et 2.*

Des solutions cohérentes à ce problème peuvent aisément être spécifiées en PROMELA, comme l'*algorithme de Peterson.*

Exemple 11

```
#define Aturn false
#define Bturn true

bool x,y,t;

proctype A()
{x = true;
 t = Bturn;
 (y == false || t == Aturn);
 printf("Debut section Critique de A\n");
 printf("Fin section Critique de A\n");
 x = false
}

proctype B()
{y = true;
 t = Aturn;
 (x == false || t == Bturn);
 printf("Debut section critique de B\n");
 printf("Fin section critique de B\n");
 y = false;
}

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

Un processus peut être déclaré de manière à s'activer au début de l'exécution. Pour cela le mot-clef `active` doit précéder `proctype`. Plusieurs instanciations d'un même processus peuvent également être activées dans l'état initial du système. Ces processus ne sont pas activés avec des arguments. Toutefois, on peut préciser des paramètres formels pour des instanciations ultérieures à partir d'autres processus. Les arguments des processus instanciés par `active` sont alors initialisés à 0.

Exemple 12

```
active proctype A() { ... }
active [4] proctype B() { ... }
```

Un processus de type `A` *et quatre de type* `B` *sont instanciés lors de l'activation du système.*

2.2.3 Les séquences atomiques

Une manière d'éviter les problèmes d'accès concurrents à une variable globale consiste à regouper, à l'aide de la primitive `atomic`, plusieurs instructions en une séquence qui sera exécutée comme une seule instruction, sans entrelacement avec les autres processus.

Exemple 13

```
byte state = 1;

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

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

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

La variable `state` *prend la valeur 0 ou 2, suivant le processus (*`A` *ou* `B`*) qui s'exécute. L'autre processus reste bloqué.*

Si un processus en train d'exécuter une séquence atomique se bloque, le contrôle est passé, de manière non déterministe à un autre processus. Si l'instruction bloquante devient par la suite exécutable, le contrôle pourra—ce n'est pas systématique ni immédiat—retourner au processus bloqué, et la suite de la séquence atomique sera exécutée. Lorsqu'un envoi de rendez-vous a lieu au sein d'une séquence atomique, le contrôle est passé au processus récepteur du rendez-vous.

Une autre primitive, `d_step`, similaire à `atomic`, permet d'exécuter en une seule séquence de longs calculs locaux déterministes. Le nombre d'états du graphe est alors réduit, et les vérifications sont plus rapides.

Exemple 14

```
d_step {
    tab[0] = 0;
    tab[1] = 1;
    tab[2] = 2;
    }
```

Les instructions *d_step* et *atomic* peuvent être combinées. L'utilisation de *d_step* peut être plus large que celle présentée dans ce document, mais elle doit être effectuée avec moultes précautions.

2.2.4 L'affichage de messages

L'instruction `printf` permet d'afficher des messages, de manière similaire à l'affichage en C. Le format d'affichage permet de prendre en compte des variables, et éventuellement des couleurs.

Exemple 15

```
show var = 3;                                        /* 1 */
printf("La valeur de la variable var est %d\n",var);/* 2 */
printf("MSC: J'affiche dans la fenetre MSC\n");      /* 3 */
printf("MSC: ~R rouge\n");                           /* 4 */
printf("MSC: ~G vert\n");                            /* 5 */
printf("MSC: ~B bleu\n");                            /* 6 */
```

L'instruction 2 affiche `La valeur de la variable var est 3`. *L'affichage numéro 3 imprime* `J'affiche dans la fenetre MSC` *à l'intrieur de la fenêtre montrant les envois de messages. Les options d'affichage des lignes 4, 5 et 6 permettent de changer ègalement la couleur du message, respectivement en rouge, vert et bleu. La déclaration de variable précédée de* `show` *permet d'afficher les différentes valeurs que prend la variable* `var` *lors de la simulation, dans la fenêtre MSC.*

L'affichage de messages dans la fenêtre MSC permet aussi d'insérer des points d'arrêt pour la simulation. Ceci est fait en utilisant le mot `BREAK` dans le message à afficher.

2.2.5 Les structures de contrôle

Ce sont des primitives permettant d'effectuer des sélections, branchements, boucles, ...

Les sélections La structure de sélection `if ... fi` contient une liste de séquences d'exécution précédées par ::. Une seule des séquences de la liste sera exécutée. Si plusieurs de ces séquences sont exécutables, l'une d'elles sera choisie de manière non déterministe. Si aucune n'est exécutable, le processus reste bloqué jusqu'à ce que l'une d'elles puisse être sélectionnée.

Exemple 16

```
#define a 1
#define b 2

byte x = 0;

proctype A()
{x = a
}

proctype B()
{x = b
}

proctype C()
{if
 :: (x == a) -> printf("x vaut 1\n");
 :: (x == b) -> printf("x vaut 2\n");
 fi;
}

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

Le processus `C` *teste la valeur de la variable* `x`. *Celle-ci a été modifiée par un des deux processus* `A` *et* `B`.

Exemple 17

```
byte nb = 5;

proctype compte()
{if
 :: nb = nb + 1
 :: nb = nb - 1
 fi;
 printf("nouvel nb = %d\n",nb)
}

init
{printf("ancien nb = %d\n",nb);
 run compte()
}
```

La nouvelle valeur de la variable `nb` *sera indifféremment 4 ou 6.*

Une primitive `else` utilisée au début d'une séquence d'instructions permet d'exécuter celle-ci si et seulement si aucune autre séquence peut l'être.

Les expressions conditionnelles Elles sont analogues au (`expr1?expr2:expr3`) du C. Leur syntaxe est : `(expr1 -> expr2 : expr3)`, où les parenthèses sont obligatoires.

Les boucles Il est possible d'écrire des boucles `do ... od` à l'intérieur d'une spécification PROMELA. Ces boucles sont infinies. On peut en sortir en utilisant l'instruction `break`. La boucle a une structure similaire à la sélection avec des listes d'instructions à exécuter.

Exemple 18

```
byte nb = 127;

proctype compte()
{do
 :: (nb!=0 && nb < 255) ->
      /* l'operation ne peut pas depasser les limites
         du type byte */
      if
      :: nb = nb + 1
      :: nb = nb - 1
      fi;
      printf("nouvel nb = %d\n",nb)
 :: (nb == 255) -> printf("Cas nb = 255\n"); break
 :: (nb == 0) -> printf("Cas nb = 0\n"); break
 od;
 printf("Fin de compte\n")
}

init
{printf("ancien nb = %d\n",nb);
 run compte()
}
```

Le processus `compte` *incrémente ou décrémente la valeur de la variable* `nb`*, de type* `byte`*. L'opération effectuée est choisie de manière non déterministe. Lorsque la valeur atteint les limites autorisées pour le type* `byte`*, le processus sort de sa boucle et termine.*

Les branchements Un autre moyen de sortir d'une boucle consiste à utiliser un branchement inconditionnel `goto` qui permet d'aller à une étiquette.

Exemple 19

```
proctype pgcd(int x,y)
{printf("Le pgcd de %d et %d est ",x,y);
 do
 :: (x > y) -> x = x - y
 :: (x < y) -> y = y - x
 :: (x == y) -> goto fini
 od;
fini :
 printf("%d\n",x)
}

init
{run pgcd(24,36)
}
```

Une étiquette ne peut apparaître qu'avant une instruction. Si aucune instruction ne doit être exécutée à partir de cet endroit, on peut utiliser `skip` qui est une instruction toujours exécutable et sans effet.

L'instruction `unless` Cette instruction s'écrit sous la forme `B unless C` où `B` et `C` sont des morceaux de programmes. S'ils sont constitués de plusieurs instructions, il faut les regrouper : {`B1; B2; B3`} `unless` {`C1; C2`}. L'exécution de l'instruction commence par celle du fragment `B1`. Avant chaque étape dans `B`, l'exécutabilité du fragment `C1` est testée. L'exécution des instructions dans `B` continue tant que `C1` ne peut être exécuté. Dès que `C1` devient exécutable, le contrôle passe à `C` et la fin de `B` est abandonnée. Si, durant l'exécution de `B`, `C1` ne devient pas exécutable, l'exécution de `C` n'a pas lieu lorsque `B` termine.

Échappements L'instruction `timeout` modélise une condition particulière qui permet à un processus d'annuler son attente d'une condition qui pourra ne jamais devenir vraie. Cette instruction fournit une échappatoire à un état bloquant. La condition `timeout` devient vraie seulement quand aucune autre instruction du système distribué n'est exécutable.

2.2.6 L'utilisation des canaux de communication

Les opérations principales effectuées sur un canal de communication sont l'envoi et la réception de messages, avec les opérateurs ! et ?.

L'envoi et la réception de messages La commande `nomcanal!expr` envoie la valeur de l'expression `expr` sur le canal `nomcanal`. Cette valeur est mise en queue de la file de `nomcanal`. L'instruction `nomcanal?msg` reçoit un message, le retire de la file associée à `nomcanal` et le stocke dans la variable `msg`. La gestion de la file associée à un canal de communication s'effectue selon une politique *FIFO*. Si un message doit contenir plusieurs valeurs, elles sont transmises dans une liste.

```
nomcanal!expr1,expr2,expr3
nomcanal?var1,var2,var3
```

On peut également utiliser une autre notation, permettant de spécifier le type du message :

```
nomcanal!expr1(expr2,expr3)
nomcanal?var1(var2,var3)
```

L'envoi d'un message dans un canal ne peut avoir lieu que si celui-ci n'est pas encore plein. De même, la réception d'un message ne peut être effectuée que si le canal n'est pas vide. Lorsque ces opérations ne peuvent être réalisées, elles sont bloquantes.

Exemple 20

```
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
}
```

Les deux processus `A` *et* `B` *ont pour argument un canal de communication, qui leur est fourni par le processus* `init`*. La valeur imprimée est* `123`*.*

La communication par *rendez-vous* Un canal de communication de taille 0 définit un point de rendez-vous. Le canal peut ainsi transporter des messages, mais pas les stocker. Les rendez-vous entre processus sont des opérations synchrones.

Exemple 21

```
#define msgtype 33
chan name = [0] of {byte,byte};

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

proctype B()
{byte state;

 name?msgtype(state);
 printf("state = %d\n",state);
}

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

Les deux processus `A` *et* `B` *sont instanciés au sein d'une séquence atomique. Ils communiquent par un rendez-vous où* `A` *transmet la valeur 124 à* `B`*, qui la stocke dans la variable locale* `state`*. La seconde instruction de* `A` *est un rendez-vous qui ne peut avoir lieu car il n'y a pas d'instruction de réception correspondante dans* `B`*. Si le canal* `name` *avait eu une capacité non nulle, la seconde instruction du processus* `A` *aurait pu être exécutée. Lors de la terminaison de l'exécution, le canal* `name` *aurait contenu le message 121.*

Réceptions de messages particulières Il est également possible d'attendre la réception d'un message spécifique, éventuellement en le recherchant dans la file du canal. L'executabilité d'une réception peut être vérifiée. Les différents types de réceptions sont résumés dans le tableau ci-dessous.

	instruction	*signification*
1	q?var,const	réception FIFO : les constantes doivent correspondre, le message est retiré du canal
2	q??var,const	premier trouvé : comme 1, en prenant le premier message pour lequel les constantes correspondent
3	q?[var,const]	test de l'exécutabilité d'une réception de type 1
4	q??[var,const]	test de l'exécutabilité d'une réception de type 2
5	q?<var,const>	comme 1, sans consommation du message
6	q??<var,const>	comme 2, sans consommation du message

Autres primitives de manipulation de canaux Plusieurs fonctions permettent de travailler sur la longueur de la file associée à un canal. Elles effectuent des tests, mais n'ajoutent ni ne retirent de message.

La fonction `len(nomcanal)` renvoie le nombre de messages présents dans la file associée à `nomcanal`.

Les primitives `full(nomcanal)` et `empty(nomcanal)` indiquent si la file associée à `nomcanal` est pleine ou vide.

Les opérations `nfull(nomcanal)` et `nempty(nomcanal)` indiquent si la file associée à `nomcanal` est non pleine ou non vide.

L'entrée standard comme canal de communication Un canal de communication prédéfini `STDIN` permet de lire un caractère sur l'entrée standard.

Exemple 22

```
chan STDIN;
short c;

do
:: STDIN?c ->
      if
      :: c == -1 -> break      /* EOF */
      :: else -> printf("%c",c)
      fi
od
```

Cette séquence d'instructions permet de lire des caractères, de les écrire sur la sortie standard, jusqu'à ce que l'on lise le caractère de fin de de fichier.

La primitive `eval` La fonction `eval` permet d'utiliser, lors d'une réception de message, la valeur d'une variable comme constante, et ce pour forcer une réception particulière.

Exemple 23

```
x = 3;
q?eval(x)
```

La valeur à recevoir sur le canal `q` *est* `3`.

2.2.7 Variables et fonctions associées aux processus

La variable locale prédéfinie `_pid` contient le numéro d'instanciation du processus.

Exemple 24

```
active [3] proctype A()
{printf("Je suis le processus numero %d\n",_pid)
}
```

3 processus sont instanciés, qui affichent chacun leur numéro (0, 1 et 2).

La variable globale prédéfinie `_last` contient le numéro d'instanciation du processus ayant effectué la dernière étape dans la séquence d'exécution courante. Sa valeur initiale est 0. L'utilisation de cette variable accroît le nombre d'états accessibles.

La fonction prédéfinie `enabled(pid)` renvoie vrai si le processus dont le numéro d'instanciation est `pid` peut exécuter une opération dans son état courant. L'utilisation de cette fonction est restreint aux clauses `never` (voir section 2.5.7) pour des modèles ne contenant pas de rendez-vous.

Exemple 25

```
never {
    accept :
        do
        :: _last != 1 && enabled(1)
        od
    }
```

Cette clause indique que le processus 1 ne peut rester activable sans jamais s'exécuter.

La fonction prédéfinie `pc_value(pid)` renvoie le numéro de l'état courant du processus dont le numéro d'instanciation est `pid`. L'utilisation de cette fonction est également restreint aux clauses `never`.

L'instruction `np_`, qui ne peut être utilisée qu'au sein d'une clause `never`, est une variable prédéfinie qui vaut `true` si le système global est dans un état de progression, `false` sinon.

2.3 Déclaration et appel de procédures

Une procédure peut être déclarée `inline`, en dehors des déclarations de processus.

Exemple 26

```
inline mafonction(x,y){
     ... code Promela ...
}
...
mafonction(4,a);
...
```

Les premières lignes montrent une déclaration de la fonction mafonction. *L'appel utilise les paramètres 4 et a.*

La substitution du texte de la procédure est effectuée par le parseur, avec les paramètres adéquats. Les procédures ainsi déclarées peuvent être récursives, c'est-à-dire qu'une déclaration de procédure peut contenir un appel de procédure, mais la suite d'appels ne peut être cyclique.

La portée des variables est locale au processus qui l'utilise. Par conséquent, toute variable déclarée dans une procédure est locale au processus qui effectue l'appel de procédure.

2.4 Primitives et concepts de spécification supplémentaires

La modélisation de la récursion Il est possible, en PROMELA, de modéliser des procédures, même récursives. Les valeurs de retour peuvent être transmises via une variable globale ou un message.

Exemple 27

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

 if
 :: (n <= 1) -> p!1
 :: (n >= 2) ->
        run fact(n-1,fils);
        fils?res;
        p!n*res
 fi
}
```

```
init
{chan fils = [1] of {int};
 int res;

 run fact(7,fils);
 fils?res;
 printf("7! = %d\n",res)
}
```

Le processus `fact(n,p)` *calcule récursivement la factorielle de* `n`*. Il communique le résultat à son processus père par un message dans le canal* `p`*.*

Priorités d'exécution Le lancement d'un processus peut être accompagné d'une priorité d'exécution spécifique. Lors d'une simulation aléatoire, un processus de forte priorité aura plus de chances de s'exécuter qu'un processus de faible priorité. La priorité est un entier, 1 représente la plus faible.

Exemple 28

```
active proctype proc1() priority 1
...
run proc2() priority 2
...
```

Les processus de type proc1 *auront une priorité 1, alors que ceux de type* proc2 *auront une priorité 2.*

Les priorités sont utilisées uniquement pour la simulation aléatoire. Elles n'ont en particulier aucun effet sur la vérification du système.

Conditions d'exécution d'un processus On peut préciser des conditions pour qu'une instruction d'un processus soit exécutable, grâce à la clause `provided`.

Exemple 29

```
mtype = { free, busy, idle, waiting, running };

show mtype h_state = idle;
show mtype l_state = idle;
show mtype mutex = free;

active proctype high() /* can run at any time */
{
end: do
     :: h_state = waiting;
        atomic { mutex == free -> mutex = busy };
        h_state = running;
        /* critical section - consume data */
        atomic { h_state = idle; mutex = free }
      od
}

active proctype low() provided (h_state == idle) /* scheduling rule */
{
end: do
     :: l_state = waiting;
        atomic { mutex == free -> mutex = busy};
        l_state = running;
        /* critical section - produce data */
        atomic { l_state = idle; mutex = free }
      od
}
```

Les instructions du processus `low` *ne peuvent être exécutées que si la valeur de la variable* `h_state` *est* `idle`*.*

La variable cachée _ La variable non nommée _ peut être écrite mais non lue. Elle peut par exemple servir à perdre un message dans un canal en effectuant l'instruction `q?_`. Cette variable n'est pas contenue dans l'état du système.

2.5 Instructions aidant à la validation d'un système

Les instructions que nous allons décrire dans cette section facilitent la validation d'un système. Il s'agit d'assertions et de caractéristiques particulières d'états, qui doivent être satisfaites pour que le comportement du système soit correct.

2.5.1 Assertions

Les instructions `assert(condition booléenne)` sont toujours exécutables. Si la condition spécifiée est valide, cette instruction n'a aucun effet. Par contre, si ce n'est pas le cas, l'instruction signalera une erreur lors de validations avec *spin*.

Exemple 30

```
active proctype verifinv() {assert(invariant)}
```

Ce processus permet de vérifier que la condition `invariant` *est vraie pour tout état atteignable du système.*

2.5.2 États terminaux

Pour vérifier le bon fonctionnement du système, il est intéressant de spécifier les états de terminaison normale des processus. Par exemple, pour vérifier l'absence d'états de blocage, l'outil d'analyse doit pouvoir distinguer un état dans lequel la terminaison est normale, d'un état de terminaison anormale. Un état de terminaison normale peut être par exemple un état dans lequel le processus a atteint la fin du code à exécuter et où tous les canaux sont vides. Mais tous les processus ne sont pas forcés d'atteindre la fin du corps de leur spécification. Ils peuvent par exemple attendre une réception éventuelle.

On utilise des étiquettes d'états terminaux pour indiquer à l'analyseur que ces états terminaux sont eux aussi valides. Il peut y avoir plusieurs états terminaux possibles pour un même processus. Les étiquettes doivent être uniques au sein d'un processus. Tout nom d'étiquette commençant par `end` représente un état terminal possible.

2.5.3 États de progression

Dans le même esprit, les étiquettes d'états de progression indiquent des états qui doivent être exécutés pour que le processus progresse. N'importe quel cycle infini de l'exécution du processus ne passant pas par au moins un de ces états de progression signale une famine potentielle. Les noms des étiquettes d'états de progression doivent commencer par `progress`.

2.5.4 États d'acceptation

Les états d'acceptation permettent de spécifier les états devant être indéfiniment visités pour qu'une propriété soit vraie. Une étiquette d'état acceptant doit commencer par `accept`.

2.5.5 Assertions sur les envois et réceptions de messages

Les assertions de réception et d'envoi exclusifs :

```
xr q1;
xs q2;
```

indiquent que le processus qui les exécute est *le seul* à pouvoir recevoir des messages sur le canal `q1` et envoyer des messages sur le canal `q2`.

2.5.6 Traces d'événements

Une déclaration de `trace` permet de spécifier un comportement correct du système.

Exemple 31

```
trace{
 do
 :: q1!a; q2?b
 od
}
```

La trace d'événements déclarée spécifie que les opérations d'envoi sur le canal `q1` *alternent avec les opérations de réception sur le canal* `q2`*. De plus tous les envois sur* `q1` *sont des messages de valeur* `a` *et toutes les réceptions sur* `q2` *sont des messages de valeur* `b`*.*

De manière similaire, la primitive `notrace` décrit les séquences d'exécution invalides.

2.5.7 Formules de logique temporelle

Une formule de logique temporelle peut être spécifiée à l'aide de la clause `never`. Ces propriétés sont définies sur des séquences d'exécution complêtes. Par conséquent, même si le préfixe d'une séquence ne présente pas d'intérêt, il *faut* le modéliser par une séquence de propositions de toute évidence vraie.

Exemple 32

```
never {do :: skip :: break od -> P -> !Q}
```

Cette clause permet de modéliser la propriété : "tout état dans lequel `P` *est vraie est suivi d'un état dans lequel* `Q` *est vraie".*

Exemple 33

```
never {
    do
    :: skip
    :: condition1 -> break
    od;
    accept : do :: condition1 od
    }
```

Cette clause exprime que la `condition1` *ne peut jamais rester vraie indéfiniment. Elle peut être initialement fausse, devenir vraie, puis le rester.*

Une clause `never` permet de suivre les exécutions au sein du système. Pour ce faire, le vérificateur calcule le produit synchrone de l'automate de la clause et de celui du système.

Nous verrons dans la section 4 que l'on peut écrire assez facilement des formules de logique temporelle à l'aide de *xspin*.

3 Utilisation de *spin*

Le travail qui conduisit à l'écriture de l'analyseur *spin* commença au début des années 80, aux laboratoires Bell. Plusieurs systèmes de ce type furent écrits, tous basés sur le paradigme de la *vérification à la volée*. D'autres analyseurs furent construits par ailleurs, qui effectuent de la vérification traditionnelle de modèles (*model checker*).

Un analyseur traditionnel fonctionne avec une procédure de vérification à deux passes. Dans la première phase, le comportement du système étudié est exploré et une représentation abstraite (*graphe d'états*) générée. Dans un second temps, cette représentation abstraite est utilisée pour prouver ou infirmer des propriétés.

Un analyseur à la volée utilise une procédure de vérification à une passe. Il essaie de stocker en mémoire aussi peu d'informations que nécessaire à la vérification elle-même, et d'effectuer cette analyse pendant l'exploration du comportement du système étudié.

spin s'appuie sur une technique de réduction utilisant des ordres partiels. Il peut effectuer des simulations aléatoires de l'exécution du système, ou générer un programme C qui effectue

une validation rapide et exhaustive de l'espace d'états du système. L'analyseur peut par exemple regarder si des invariants du système spécifiés par l'utilisateur peuvent ne pas être satisfaits lors d'une exécution du protocole. Il permet en outre la vérification de formules de logique temporelle linéaire (LTL).

3.1 Options de *spin*

Si *spin* est invoqué sans option, il effectue une simulation aléatoire.

`-nN` attribue N comme racine pour la validation.

`-p` montre les changements d'états des processus PROMELA à chaque étape.

`-g` indique la valeur courante des variables globales à chaque étape.

`-l` indique la valeur des variables locales après un changement d'état de leur processus.

`-r` montre toutes les réceptions de messages. Le processus qui effectue la réception est indiqué, avec son nom et son numéro, ainsi que le numéro de la ligne du code source, le message et le canal.

`-s` montre tous les envois de messages.

`-a` génère un analyseur spécifique au protocole. Le résultat est un ensemble de fichiers en C qui peuvent être compilés pour produire l'analyseur.

`-m` peut être utilisé pour changer la sémantique des opérations d'envoi de messages. Par défaut, un envoi de message ne peut être effectué que si le canal n'est pas plein. Avec cette option, les envois de messages peuvent toujours être exécutés. Si le canal est plein, le message est perdu.

`-t` est une option de trace. Si l'analyseur trouve une assertion non satisfaite, un état de blocage ou une réception non spécifiée, il écrit une trace de l'erreur dans un fichier `pan.trail`. Cette trace peut être examinée en détail en invoquant *spin* avec l'option `-t`.

`-D` Deux options d'analyse du flot de données sont disponibles.

La première indique les opérations d'écriture effectuées qui ne pourront être suivies d'une lecture. La valeur écrite est par conséquent sans intérêt. En évitant ces écritures, on réduit le nombre d'états accessibles pendant la vérification, diminuant ainsi le temps et l'espace utilisés.

La seconde option indique les dernières opérations de lecture effectuées sur les différents objets. Toute opération décriture faite ensuite ne sera pas suivie d'une lecture.

```
$ spin -D nomfich                    # premiere option
$ spin -D -D nomfich                 # seconde option
```

`-d` Il est possible d'afficher la liste des symboles identifiés par *spin* lors de l'analyse du source PROMELA.

```
$ spin -d nomfich
```

Pour chaque objet sont décrits son type, son nom et son nombre d'éléments (tableau), sa valeur initiale ou sa taille (canal), son champ d'application (local, global), et ses sous-types (canal).

`-N fichierclause` Lit la clause `never` dans le fichier *fichierclause*

3.2 Options de compilation de l'analyseur

Suivant les options de compilations choisies, l'analyseur travaillera de différentes manières : méthode de stockage des graphes d'états, niveau de détail des rapports fournis à l'utilisateur, ...

3.2.1 Réductions

Il est possible d'utiliser une méthode de réduction basée sur les ordres partiels préservant les propriétés de sûreté et de vivacité lors de la vérification. Une autre réduction consiste à représenter chaque état par un seul bit. Ces deux réductions peuvent être combinées.

```
$ spin -a nomfich                    # produire l'analyseur
$ cc -DNOREDUCE -o complet pan.c     # recherche exhaustive
$ cc -DNOREDUCE -DBITSTATE bit pan.c # en mode bit
$ cc -o redcomplet pan.c             # ordre partiel
$ cc -DBITSTATE -o rbit pan.c        # bit + ordre partiel
```

Le mode de compilation par défaut utilise la réduction par ordres partiels. Pour que cette réduction préserve les propriétés de vivacité, les clauses `never` doivent être fermées par bégaiement. Cela signifie que la propriété exprimée par la clause `never` ne doit pas dépendre du nombre de pas de programme pendant lesquels une propriété reste vraie ou fausse. Les clauses `never` générées par traduction automatique d'une formule LTL (voir section 4.2) satisfont cette propriété.

3.2.2 Complexité

L'option de compilation `-DPEG` permet de collecter des informations supplémentaires pendant la vérification. L'analyseur compte le nombre d'occurrences de chaque instruction effectuées pendant la vérification. Cela permet de repérer les instructions qui contribuent le plus à la taille du graphe d'accessibilité.

3.2.3 Mode de vérification bavard

Deux modes sont disponibles pour étudier plus en détail ce que fait l'analyseur à chaque étape.

```
$ spin -a nomfich                    # produire l'analyseur
$ cc -DCHECK -o pan pan.c            # mode concis
$ cc -DVERBOSE -o pan pan.c          # mode bavard
```

L'option `-DSAFETY` permet d'optimiser la mémoire et le temps nécessaires aux vérifications lorsque seules des propriétés de sûreté sont recherchées.

3.3 Options de l'analyseur

L'analyseur, une fois compilé, dispose également de différentes options. Sans option, il ne recherche que les propriétés de sûreté : assertions non satisfaites, états terminaux non valides, réceptions non spécifiées, parties de code non atteignables.

`-?` imprime un récapitulatif des options de l'analyseur. Attention, lorsque l'on travaille sous Unix, cette option doit être passée entre guillemets pour que le shell n'essaie pas d'effectuer une substitution.

`-c`*N* indique le nombre d'erreurs au bout duquel l'analyseur doit s'arrêter. Par défaut, il s'arrête à la première erreur.

`-l` indique à l'analyseur de chercher les boucles de non-progression.

`-a` demande à l'analyseur de chercher les cycles d'acceptation.

`-m`*N* effectue une recherche jusqu'à la profondeur N. La profondeur par défaut est de 10.000 pas.

`-d` imprime les tables d'états et de transitions.

`-n` supprime la liste des états non atteignables.

`-i` si l'analyseur a été compilé avec les options `-DREACH` et `-DNOREDUCE`, cette option permet de rechercher le plus court chemin conduisant à une erreur.

`-I` cette option est similaire à `-i`, mais effectue des approximations. Elle est par conséquent plus rapide, mais ne donne pas forcément le meilleur résultat.

`-q` demande que les canaux soient vides pour qu'un état terminal soit valide.

4 Les fonctionnalités de *xspin*

xspin offre à l'utilisateur une interface graphique facilitant les accès aux différentes commandes de *spin*. Lors d'une session de travail avec *xspin*, une trace de toutes les actions effectuées est indiquée sur la sortie standard : erreurs de syntaxe, événements inattendus, compilations en tâche de fond, vérifications, ... La fenêtre principale peut être utilisée pour éditer le source PROMELA. *xspin* utilise une copie du fichier de spécification dans un fichier nommé `pan.in`.

4.1 La simulation

Le menu `Set Simulation Parameters` fait apparaître une fenêtre présentant diverses options de simulation. Celle-ci peut être aléatoire ou guidée. Une simulation guidée ne peut avoir lieu que si une vérification a été effectuée au préalable, montrant une erreur dans le protocole. La trace menant à l'erreur peut alors être déroulée en utilisant la simulation guidée. La simulation guidée peut être effectuée pas à pas aussi bien en avançant qu'en reculant (attention aux valeurs des variables dans ce cas, elles peuvent être erronnées). La simulation elle-même est lancée au travers du menu `(Re)Run Simulation`.

4.2 Les formules de logique temporelle

Le menu `LTL Property Manager` fait apparaître une fenêtre dans laquelle l'utilisateur peut entrer des fomules LTL, avec la syntaxe classique. En cliquant sur le bouton `Generate`, cette formule est transformée en une clause `never`. Cette clause peut être vérifiée en cliquant sur le bouton `Run Verification`.

4.3 Les automates

À chaque définition de processus (`proctype`) est associé un automate que l'on visualise en utilisant le menu `View Spin Automaton for each Proctype`. Les différents états et transitions y sont présentés.

5 Exemples d'utilisation de *spin*

Nous présentons, dans cette section, deux exemples détaillés de spécification en PROMELA et de sessions de déboggage à l'aide de *spin*.

5.1 Problème d'exclusion mutuelle

Une solution au problème de l'exclusion mutuelle a été publiée par Hyman en pseudo-Algol sous la forme suivante :

```
boolean array b(0,1)
integer k,i
/* processus i, avec i = 0 ou 1 et k = 1-i */
C0:  b(i) := false;
C1:  if k != i then begin
C2:  if !b(1-i) then goto C2
                else k:=i ;  goto C1 end;
     else section critique;
     b(i) := true;
     reste du programme;
     goto C0;
     end
```

Supposons que nous voulons vérifier que cet algorithme garantit réellement l'accès exclusif à la section critique. Nous construisons d'abord une spécification en PROMELA.

```
bool demande[2];
bool tour;

proctype P(bool i)
{demande[i] = 1;
 do
 :: (tour != i) ->
        (!demande[1 - i]);
        tour = i
 :: (tour == i) ->
        break;
 od;
 skip; /* section critique */
 demande[i] = 0
}

init {run P(0); run P(1)}
```

Nous pouvons maintenant générer, compiler et exécuter l'analyseur correspondant à cette spécification.

```
$ spin -a Hyman0
$ cc pan.c
$ a.out
(Spin Version 2.8.1 -- 14 April 1996)
        + Partial Order Reduction

Full statespace search for:
        never-claim                - (none specified)
        assertion violations       +
        non-progress cycles        - (not selected)
        acceptance   cycles        - (not selected)
        invalid endstates          +

State-vector 20 byte, depth reached 19, errors: 0
      56 states, stored
      15 states, matched
      71 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.29682e+06        memory usage (bytes)

unreached in proctype P
        (0 of 12 states)
unreached in proctype :init:
        (0 of 3 states)
```

Il n'y a, a priori, pas de problème avec cette spécification. Toutefois, nous devons exprimer la propriété requise en PROMELA. Pour cela, nous ajoutons un compteur qui contient le nombre de processus en section critique.

```
bool demande[2];
bool tour;
byte nb;
```

```
proctype P(bool i)
{demande[i] = 1;
 do
 :: (tour != i) ->
        (!demande[1 - i]);
        tour = i
 :: (tour == i) ->
        break;
 od;
 skip; /* section critique */
 nb++;
 assert(nb == 1);
 nb--;
 demande[i] = 0
}

init {run P(0); run P(1)}
```

Nous procédons de nouveau à la génération, compilation et exécution de l'analyseur.

```
$ spin -a Hyman1
$ cc pan.c
$ a.out
pan: assertion violated (nb==1) (at depth 15)
pan: wrote Hyman1.trail
(Spin Version 2.8.1 -- 14 April 1996)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never-claim                 - (none specified)
        assertion violations        +
        non-progress cycles         - (not selected)
        acceptance   cycles         - (not selected)
        invalid endstates           +

State-vector 24 byte, depth reached 25, errors: 1
      66 states, stored
      16 states, matched
      82 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 6 (resolved)
(max size 2^18 states)

1.29682e+06        memory usage (bytes)
```

L'analyseur indique que l'assertion peut ne pas être satisfaite. Nous allons donc examiner la trace.

```
    $ spin -t -p Hyman1
 1: proc 0 (:init:) line 21 "Hyman1" (state 1) [(run P(0))]
 2: proc 0 (:init:) line 21 "Hyman1" (state 2) [(run P(1))]
 3: proc 2 (P) line  6 "Hyman1" (state 1)  [demande[i] = 1]
 4: proc 2 (P) line  8 "Hyman1" (state 2)  [((tour!=i))]
 5: proc 2 (P) line  9 "Hyman1" (state 3)  [(!(demande[(1-i)]))]
 6: proc 1 (P) line  6 "Hyman1" (state 1)  [demande[i] = 1]
 7: proc 1 (P) line 11 "Hyman1" (state 5)  [((tour==i))]
```

```
 8: proc 1 (P) line 14 "Hyman1" (state 10) [(1)]
 9: proc 2 (P) line 10 "Hyman1" (state 4)  [tour = i]
10: proc 2 (P) line 11 "Hyman1" (state 5)  [((tour==i))]
11: proc 2 (P) line 14 "Hyman1" (state 10) [(1)]
12: proc 2 (P) line 15 "Hyman1" (state 11) [nb = (nb+1)]
13: proc 2 (P) line 16 "Hyman1" (state 12) [assert((nb==1))]
14: proc 1 (P) line 15 "Hyman1" (state 11) [nb = (nb+1)]
spin: line  16 "Hyman1", Error: assertion violated
15: proc 1 (P) line 16 "Hyman1" (state 12) [assert((nb==1))]
spin: trail ends after 15 steps
#processes: 3
                demande[0] = 1
                demande[1] = 1
                tour = 1
                nb = 2
15: proc 2 (P) line 17 "Hyman1" (state 13)
15: proc 1 (P) line 17 "Hyman1" (state 13)
15: proc 0 (:init:) line  21 "Hyman1" (state 3)
3 processes created
```

5.2 Protocole du bit alterné

Il n'est pas toujours possible d'exprimer la validité du système par un invariant global. Ceci est illustré par l'exemple du protocole du bit alterné, avec perte et distorsion de messages, et étendu à l'aide d'accusés de réception négatifs. Écrivons d'abord une spécification du protocole en PROMELA.

```
#define MAX 5
mtype = {mesg, ack, nak, err};

proctype emetteur(chan in, out)
{byte o,s,r;

 o = MAX - 1;
 do
 :: o = (o + 1) % MAX;               /* message suivant */
encore: if
        :: out!mesg(o,s)             /* envoi */
        :: out!err(0,0)              /* endommage */
        :: skip                      /* ou perdu */
        fi;
        if
        :: timeout -> goto encore
        :: in?err(0,0) -> goto encore
        :: in?nak(r,0) -> goto encore
        :: in?ack(r,0) ->
                if
                :: (r == s) -> goto progress
                :: (r != s) -> goto encore
                fi
        fi;
progress : s = 1 - s      /* numero de sequence suivant */
 od
}
```

```
proctype recepteur(chan in, out)
{byte i;        /* entree */
 byte s;        /* numero de sequence */
 byte es;       /* numero de sequence attendu */
 byte ei;       /* entree attendue */

 do
 :: in?mesg(i,s) ->
        if
        :: (s == es) ->
                assert(i == ei);
progress :        es = 1 - es;
                ei = (ei + 1) % MAX;
                if
                :: out!ack(s,0)      /* envoyer */
                :: out!err(0,0)      /* endommager */
                :: skip              /* perdre */
                fi
        :: (s != es) ->
                if
                :: out!nak(s,0)      /* envoyer */
                :: out!err(0,0)      /* endommager */
                :: skip              /* perdre */
                fi
        fi
 :: in?err(i,s) ->
        out!nak(s,0)
 od
}

init
{chan s_r = [1] of {mtype, byte, byte};
 chan r_s = [1] of {mtype, byte, byte};

 atomic {
     run emetteur(r_s,s_r);
     run recepteur(s_r,r_s)
     }
}
```

Nous effectuons maintenant la génération, compilation et exécution de l'analyseur.

```
$ spin -a bitalt0
$ cc pan.c
$ a.out
(Spin Version 2.8.1 -- 14 April 1996)
        + Partial Order Reduction

Full statespace search for:
        never-claim                - (none specified)
        assertion violations       +
        non-progress cycles        - (not selected)
        acceptance   cycles        - (not selected)
        invalid endstates          +

State-vector 52 byte, depth reached 131, errors: 0
     463 states, stored
     167 states, matched
```

```
     630 transitions (= stored+matched)
       1 atomic steps
hash conflicts: 45 (resolved)
(max size 2^18 states)

1.3173e+06        memory usage (bytes)

unreached in proctype emetteur
        line 28, state 27, "-end-"
        (1 of 27 states)
unreached in proctype recepteur
        line 58, state 24, "-end-"
        (1 of 24 states)
unreached in proctype :init:
        (0 of 4 states)
```

Le résultat obtenu montre que toutes les données qui sont reçues le sont dans le bon ordre, et sans suppression. Par contre, nous n'avons pas vérifié que toutes arrivent. Il se peut qu'émetteur et récepteur bouclent en échangeant des messages d'erreur, sans jamais progresser. Pour montrer l'absence de cycles d'exécution infinis ne passant pas par des états de progression, nous examinons les boucles.

```
$ a.out -l
pan: non-progress cycle (at depth 131)
pan: wrote bitalt0.trail
(Spin Version 2.8.1 -- 14 April 1996)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never-claim                - (none specified)
        assertion violations       +
        non-progress cycles        + (fairness disabled)
        acceptance   cycles        - (not selected)
        invalid endstates          +

State-vector 52 byte, depth reached 135, errors: 1
     132 states, stored (139 visited)
       2 states, matched
     141 transitions (= visited+matched)
       1 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.30501e+06        memory usage (bytes)
```

Il y a des cycles de non-progression. Le premier rencontré peut être tracé. Un canal peut perdre ou abimer un message indéfiniment. Pour voir combien il y a de cycles de non-progression, nous pouvons utiliser l'option `-c`.

```
$ a.out -l -c0
pan: non-progress cycle (at depth 131)
(Spin Version 2.8.1 -- 14 April 1996)
        + Partial Order Reduction

Full statespace search for:
        never-claim                - (none specified)
        assertion violations       +
```

```
        non-progress cycles         + (fairness disabled)
        acceptance   cycles         - (not selected)
        invalid endstates           +

State-vector 52 byte, depth reached 136, errors: 108
     463 states, stored (901 visited)
     638 states, matched
    1539 transitions (= visited+matched)
       2 atomic steps
hash conflicts: 365 (resolved)
(max size 2^18 states)

1.32139e+06        memory usage (bytes)

unreached in proctype emetteur
        line 28, state 27, "-end-"
        (1 of 27 states)
unreached in proctype recepteur
        line 58, state 24, "-end-"
        (1 of 24 states)
unreached in proctype :init:
        (0 of 4 states)
```

Il y a 108 cycles sans progression. On pourrait les examiner un par un, mais ce serait fastidieux. Par conséquent nous allons restreindre ce nombre de cycles à ceux qui ne sont pas des boucles infinies de pertes de messages (cas normal). Pour cela, nous ajoutons des étiquettes `progress` devant chaque perte.

```
#define MAX 5
mtype = {mesg, ack, nak, err};

proctype emetteur(chan in, out)
{byte o,s,r;

 o = MAX - 1;
 do
 :: o = (o + 1) % MAX;              /* message suivant */
encore: if
        :: out!mesg(o,s);           /* envoyer */
        :: skip;
progress1 : out!err(0,0)            /* endommager */
        :: skip                     /* perdre */
        fi;
        if
        :: timeout -> goto encore
        :: in?err(0,0) -> goto encore
        :: in?nak(r,0) -> goto encore
        :: in?ack(r,0) ->
                if
                :: (r == s) -> goto progress
                :: (r != s) -> goto encore
                fi
        fi;
progress : s = 1 - s      /* numero de sequence suivant */
 od
}
```

```
proctype recepteur(chan in, out)
{byte i;        /* entree */
 byte s;        /* numero de sequence */
 byte es;       /* numero de sequence attendu */
 byte ei;       /* entree attendue */

 do
 :: in?mesg(i,s) ->
        if
        :: (s == es) ->
                assert(i == ei);
progress :        es = 1 - es;
                ei = (ei + 1) % MAX;
                if
                :: out!ack(s,0)      /* envoyer */
                :: skip;
progress2 :           out!err(0,0)   /* endommager */
                :: skip              /* perdre */
                fi
        :: (s != es) ->
                if
                :: out!nak(s,0)      /* envoyer */
                :: skip;
progress3 :           out!err(0,0)   /* endommager */
                :: skip              /* perdre */
                fi
        fi
 :: in?err(i,s) ->
        out!nak(s,0)
 od
}

init
{chan s_r = [1] of {mtype, byte, byte};
 chan r_s = [1] of {mtype, byte, byte};

 atomic {
     run emetteur(r_s,s_r);
     run recepteur(s_r,r_s)
     }
}
```

Nous essayons de retrouver un cycle de non-progression dans ce nouveau modèle.

```
$ spin -a bitalt1
$ cc pan.c
$ a.out -l
pan: non-progress cycle (at depth 132)
pan: wrote bitalt1.trail
(Spin Version 2.8.1 -- 14 April 1996)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never-claim                - (none specified)
        assertion violations       +
        non-progress cycles        + (fairness disabled)
        acceptance   cycles        - (not selected)
```

```
        invalid endstates           +

State-vector 52 byte, depth reached 136, errors: 1
     133 states, stored (140 visited)
       2 states, matched
     142 transitions (= visited+matched)
       1 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.30501e+06        memory usage (bytes)
```

Examinons la trace obtenue. Elle révèle une grave erreur dans la spécification du protocole.

```
    $ spin -t -r -s bitalt1
  5: proc  1 (emetteur) line 12 Send mesg,0,0 -> queue 1 (out)
  6: proc  2 (recepteur) line 38 Recv mesg,0,0 <- queue 1 (in)
 11: proc  2 (recepteur) line 45 Send ack,0,0 -> queue 2 (out)
 12: proc  1 (emetteur) line 21 Recv ack,0,0 <- queue 2 (in)
 16: proc  1 (emetteur) line 12 Send mesg,1,1 -> queue 1 (out)
 17: proc  2 (recepteur) line 38 Recv mesg,1,1 <- queue 1 (in)
 22: proc  2 (recepteur) line 45 Send ack,1,0 -> queue 2 (out)
 23: proc  1 (emetteur) line 21 Recv ack,1,0 <- queue 2 (in)
 27: proc  1 (emetteur) line 12 Send mesg,2,0 -> queue 1 (out)
 28: proc  2 (recepteur) line 38 Recv mesg,2,0 <- queue 1 (in)
 33: proc  2 (recepteur) line 45 Send ack,0,0 -> queue 2 (out)
 34: proc  1 (emetteur) line 21 Recv ack,0,0 <- queue 2 (in)
 38: proc  1 (emetteur) line 12 Send mesg,3,1 -> queue 1 (out)
 39: proc  2 (recepteur) line 38 Recv mesg,3,1 <- queue 1 (in)
 44: proc  2 (recepteur) line 45 Send ack,1,0 -> queue 2 (out)
 45: proc  1 (emetteur) line 21 Recv ack,1,0 <- queue 2 (in)
 49: proc  1 (emetteur) line 12 Send mesg,4,0 -> queue 1 (out)
 50: proc  2 (recepteur) line 38 Recv mesg,4,0 <- queue 1 (in)
 55: proc  2 (recepteur) line 45 Send ack,0,0 -> queue 2 (out)
 56: proc  1 (emetteur) line 21 Recv ack,0,0 <- queue 2 (in)
 60: proc  1 (emetteur) line 12 Send mesg,0,1 -> queue 1 (out)
 61: proc  2 (recepteur) line 38 Recv mesg,0,1 <- queue 1 (in)
 66: proc  2 (recepteur) line 45 Send ack,1,0 -> queue 2 (out)
 67: proc  1 (emetteur) line 21 Recv ack,1,0 <- queue 2 (in)
 71: proc  1 (emetteur) line 12 Send mesg,1,0 -> queue 1 (out)
 72: proc  2 (recepteur) line 38 Recv mesg,1,0 <- queue 1 (in)
 77: proc  2 (recepteur) line 45 Send ack,0,0 -> queue 2 (out)
 78: proc  1 (emetteur) line 21 Recv ack,0,0 <- queue 2 (in)
 82: proc  1 (emetteur) line 12 Send mesg,2,1 -> queue 1 (out)
 83: proc  2 (recepteur) line 38 Recv mesg,2,1 <- queue 1 (in)
 88: proc  2 (recepteur) line 45 Send ack,1,0 -> queue 2 (out)
 89: proc  1 (emetteur) line 21 Recv ack,1,0 <- queue 2 (in)
 93: proc  1 (emetteur) line 12 Send mesg,3,0 -> queue 1 (out)
 94: proc  2 (recepteur) line 38 Recv mesg,3,0 <- queue 1 (in)
 99: proc  2 (recepteur) line 45 Send ack,0,0 -> queue 2 (out)
100: proc  1 (emetteur) line 21 Recv ack,0,0 <- queue 2 (in)
104: proc  1 (emetteur) line 12 Send mesg,4,1 -> queue 1 (out)
105: proc  2 (recepteur) line 38 Recv mesg,4,1 <- queue 1 (in)
110: proc  2 (recepteur) line 45 Send ack,1,0 -> queue 2 (out)
111: proc  1 (emetteur) line 21 Recv ack,1,0 <- queue 2 (in)
115: proc  1 (emetteur) line 12 Send mesg,0,0 -> queue 1 (out)
116: proc  2 (recepteur) line 38 Recv mesg,0,0 <- queue 1 (in)
122: proc  2 (recepteur) line 47 Send err,0,0 -> queue 2 (out)
```

```
123: proc  1 (emetteur) line 19 Recv err,0,0 <- queue 2 (in)
124: proc  1 (emetteur) line 12 Send mesg,0,0 -> queue 1 (out)
125: proc  2 (recepteur) line 38 Recv mesg,0,0 <- queue 1 (in)
127: proc  2 (recepteur) line 52 Send nak,0,0 -> queue 2 (out)
128: proc  1 (emetteur) line 20 Recv nak,0,0 <- queue 2 (in)
129: proc  1 (emetteur) line 12 Send mesg,0,0 -> queue 1 (out)
130: proc  2 (recepteur) line 38 Recv mesg,0,0 <- queue 1 (in)
<<<<<START OF CYCLE>>>>>
132: proc  2 (recepteur) line 52 Send nak,0,0 -> queue 2 (out)
133: proc  1 (emetteur) line 20 Recv nak,0,0 <- queue 2 (in)
134: proc  1 (emetteur) line 12 Send mesg,0,0 -> queue 1 (out)
135: proc  2 (recepteur) line 38 Recv mesg,0,0 <- queue 1 (in)
137: proc  2 (recepteur) line 52 Send nak,0,0 -> queue 2 (out)
spin: trail ends after 137 steps
#processes: 3
137: proc  2 (recepteur) line 37 (state 23)
137: proc  1 (emetteur) line 17 (state 22)
137: proc  0 (:init:) line 71 (state 4)
3 processes created
```

Lorsqu'un accusé de réception est endommagé et transformé en message d'erreur, émetteur et récepteur se retrouvent coincés dans un cycle infini où l'émetteur envoie le dernier message pour lequel il n'a pas reçu d'accusé, et le récepteur rejette ce message en envoyant un accusé négatif.