

Modélisation et analyse qualitative de systèmes

Modélisation d'application en PROMELA
et utilisation de l'outil SPIN pour son analyse

<http://spinroot.com/spin/whatispin.html>

spin et iSpin

- `spin` est un outil permettant la simulation et la vérification d'algorithmes répartis
- `iSpin` est son interface graphique
- Les algorithmes sont décrits dans le langage `promela`
- `promela` permet la représentation de
 - Processus concurrents
 - Mémoire partagée
 - Communication par message
 - Synchronisation
- La simulation et la vérification de systèmes concrets (écrits en C) ont été réalisées avec `spin`.

promela

- Éléments constitutifs
 - Des variables (globales et/ou locales)
 - Des processus concurrents strictement asynchrones
 - Des canaux de communication (globaux et/ou locaux) de type FIFO borné
 - Variables
 - Types de base
 - `bit`, `bool`, `byte`, `short`, `int`
 - Tableaux statiques
 - `int i; short s = 0;`
 - `byte tab[10]; /* indices de 0 à 9 */`
- Les tableaux ne peuvent pas être transmis via les canaux

promela : définition des processus

```
proctype <nom_proctype>(<paramètres formels>) {  
    instructions  
}
```

- *définit* le comportement type des processus `nom_proctype`
- un processus peut avoir des paramètres (types de base ou canaux)
- le processus particulier `init` (s'il existe) est lancé au démarrage de l'application
- Un processus devient *actif*
 - Lorsqu'il est instancié par l'instruction `run` (souvent dans `init`)
`run <nom_proctype>(<paramètres effectifs>)`
 - s'il est déclaré *active*, lors du démarrage de l'application
`active proctype <nom_proctype>() {`
 ...
}

promela : instructions des processus

- L'exécution d'une instruction est atomique
- Instructions :
 - condition (instruction bloquante si la condition n'est pas satisfaite)
 - affectation (toujours exécutable)

les expressions d'affectation et les conditions booléennes utilisent la syntaxe du C

- émission sur un canal de communication !
(instruction bloquante si le canal est saturé, ou perte du message émis)
- réception sur un canal de communication ?
(instruction bloquante si le canal est vide)
- Deux séparateurs d'instructions possibles : ; et ->
- Structures de contrôle conditionnelle (if) et répétitive (do)

Premier programme en promela

```
byte state = 1; | variable globale

proctype A() { | variable locale
    byte tmp;
    (state==1) -> tmp = state; tmp = tmp+1; state = tmp
} | condition | affectation

proctype B() {
    byte tmp;
    (state==1) -> tmp = state; tmp = tmp-1; state = tmp
}

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

Exécution du programme

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

A est lancé avant B. A commence à s'exécuter avant le lancement de B. **Ou non.**

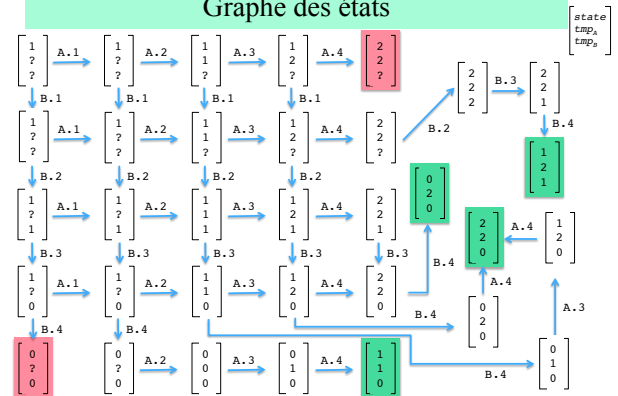
```
proctype A() {
    byte tmp;
    (state==1) -> tmp = state; tmp = tmp+1; state = tmp
}
```

```
proctype B() {
    byte tmp;
    (state==1) -> tmp = state; tmp = tmp-1; state = tmp
}
```

L'exécution d'un processus n'est pas atomique
La variable state peut prendre la valeur 0, 1 ou 2 à la fin de l'exécution

Un des processus peut se trouver définitivement bloqué

Graphe des états



Instructions atomiques (test and set)

```
byte state = 1;

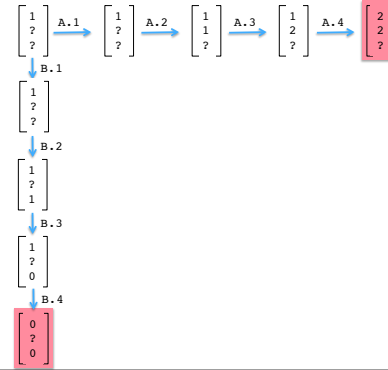
proctype A() {
    byte tmp;
    atomic {
        (state == 1) -> tmp = state; tmp = tmp + 1; state = tmp
    }
}

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

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

Une instruction bloquante dans une
séquence atomique « casse » l'atomicité

Graphe des états



Exclusion mutuelle

```
#define Aturn    false
#define Bturn    true
bool x, y, t;
```

```
active proctype A() {
    do
        :: x = true;
           t = Bturn;
           (y == false || t == Aturn);
           printf("A enters in CS\n");
           /* critical section */
           printf("A leaves CS\n");
           x = false
    od
}

option
active proctype B() {
    do
        :: y = true;
           t = Aturn;
           (x == false || t == Bturn);
           printf("B enters in CS\n");
           /* critical section */
           printf("B leaves CS\n");
           y = false
    od
}
```

option

Canaux de communication (FIFO)

```
mtype = {T1, ACK}
chan can_AB = [1] of {mtype, int};
```

```
active proctype A() {
    int x;
    mtype t;
    can_AB ? t(x);
    printf("x = %d\n", x);
    can_AB ! ACK, 123
}
```

```
active proctype B() {
    int y;
    mtype t;
    can_AB ! T1(123);
    can_AB ? t(y);
    printf("y = %d\n", y)
}
```

déclaration d'un canal :
capacité et type de données
véhiculées

lecture sur le canal :
les champs sont stockés dans
t et x respectivement

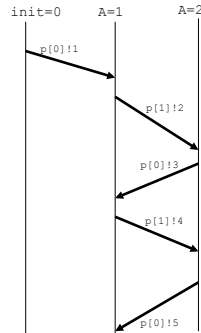
envoi sur le canal
du message (T1, 123)

Message Sequence Charts

```
proctype A(chan in,out) {
  byte x;
  do
    :: in ? x -> out ! x+1
  od
}

init{
  chan p [2] = [1] of {byte};

  atomic {
    p[0]!1;
    run A(p[0],p[1]);
    run A(p[1],p[0]);
  }
}
```



Canaux de communication (suite)

- Opérateur `len`
`len(<nom de canal>)`
 Rend le nombre de messages présents dans le canal
- Instruction de condition sur un canal
`c?[123]`
 Condition satisfaite si la prochaine valeur lue sur le canal `c` est égale à 123
- Lecture sans consommation sur un canal
`c?<x>`
 La donnée en tête du canal `c` est placée dans `x` mais n'est pas retirée de `c`
- Attention à la non-atOMICité
`c?[123] -> c?123`
 La deuxième instruction peut ne pas être exécutable alors que la condition est satisfaite

Communications synchrones

```
#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()
  }
}
```

Canal de synchronisation (capacité nulle).
Une écriture ne peut être réalisée que simultanément à une lecture

Bien qu'étant une instruction d'écriture, cette instruction ne sera jamais franchissable

Condition et indéterminisme

```
#define a 1
#define b 2

chan ch = [1] of { byte };

proctype A(){
  ch!a
}

proctype B(){
  ch!b
}

proctype C(){
  byte x;
  if
  :: ch?a -> x=a
  :: ch?b -> x=b
  fi;
  printf("%d",x);
}
```

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

L'entrée `::` exécutée est choisie de façon indéterministe parmi les instructions « franchissables »

Le programme affiche soit 1 soit 2

La dernière entrée peut être conditionnée par le mot clé `else`

Canaux non FIFO ou non fiables

- Pas de lecture aléatoire d'un canal, mais on peut récupérer un message qui n'est pas en tête de liste
`c??[123]`
 Condition satisfaite si le canal c contient un message 123

- On peut utiliser l'indéterminisme pour construire des canaux non fiables

```
if
::   ch?var1,var2
::   ch?_,_
fi;
```

Répétition et indéterminisme

```
byte count1 = 1, count2 = 1;

proctype counter1() {
  do
    :: count1 = count1 + 1
    :: count1 = count1 - 1
    :: (count1 == 0) -> break;
  od
}

proctype counter2() {
  do
    :: (count2 != 0) ->
      if
        :: count2 = count2 + 1
        :: count2 = count2 - 1
      fi
    :: (count2 == 0) -> break
  od
}
```

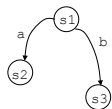
Même principe que pour la structure conditionnelle

- Le premier compteur peut passer par la valeur 0 sans pour autant s'arrêter
- Le second compteur s'arrête systématiquement dès qu'il atteint la valeur 0

```
init {
  atomic {
    run counter1();
    run counter2();
  }
}
```

Sauts, instructions d'échappement et nulle

- Les instructions peuvent être munies d'une étiquette.
- `goto label` permet de se brancher inconditionnellement à l'instruction `label`.
- L'exécution de l'instruction `skip` n'affecte que la valeur du compteur ordinal du processus qui la contient.



```
proctype A() {
  s1: if
    :: x ? a -> goto s2
    :: x ? b -> goto s3
    :: else -> skip
  fi;
  s2: ...;
  goto s4;
  s3: ...
}
```

Déclaration et appels de sous-programme

- Pas de fonction ou de procédures, mais des « inline » : portion de code recopiée lors de l'appel (macro). Les paramètres ne sont pas typés.
- Les inline sont déclarés au même niveau que les processus. Un inline peut faire appel à un autre inline mais il ne peut y avoir de dépendance cyclique, ni d'appel récursif.

```
inline ma_procedure (x,y) {
  code promela
}
```

- Un inline ne renvoie pas de résultat à l'appelant
- Un inline n'a pas de variable locale : la portée d'une variable est soit le processus, soit le programme. Mais on peut déclarer une variable dans un inline.

Expansion des « inline »

- Possibilité de visualiser le code produit après expansion des inline :

```
spin -I source.pml
```

- Exemple :

```

#define N 2      /* Number of processes */
mtype = {msg1, msg2};
chan emission[N] = [3] of { mtype };
inline Broadcast (msg) {
    ...
}

active [2] proctype node() {
    mtype msg_recu;
    Broadcast (msg1);
    Broadcast (msg2);
    emission[_pid] ? msg_recu;
    emission[_pid] ? msg_recu;
}

```

Exemple

```

inline Broadcast (msg) {
    byte i;
    i=1;
    do
        :: i < N -> atomic {
            emission[ (_pid+i)%N]!msg;
        }
        i++;
        :: i == N -> break
    od;
}

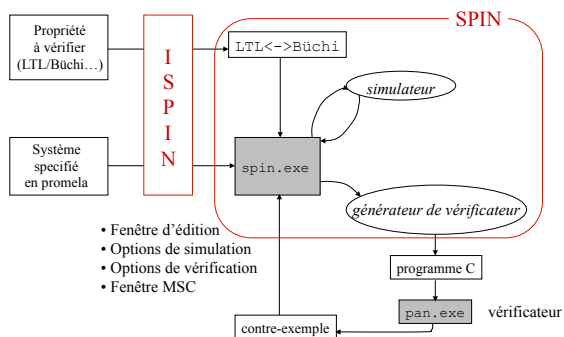
prototype node()
{
    {
        i = 1;
        do
            ..
            ((i<2));
            atomic {
                emission[(_pid+i%2)]!msg1;
            };
            i = (i+1);

            ::
            ((i==2));
            goto :b0;

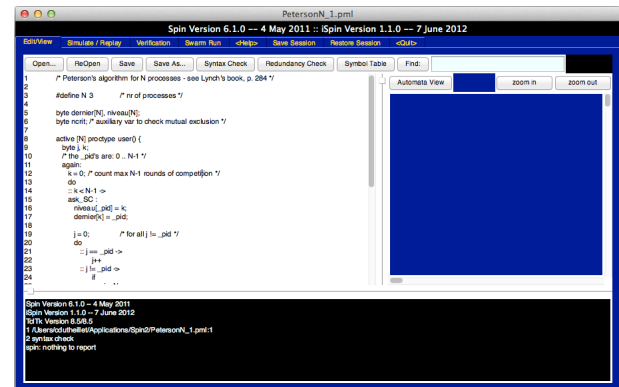
        od;
        :b0:
    };
    {
        i = 1;
        do
            ...

```

SPIN et iSpin



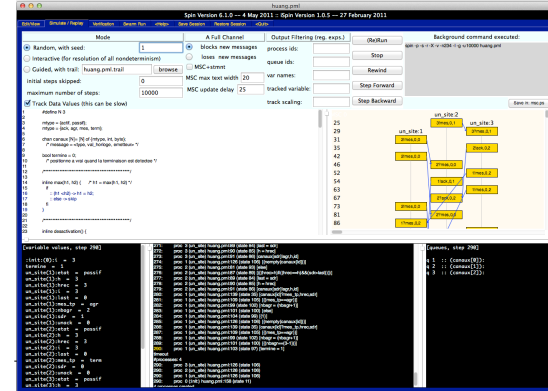
iSpin



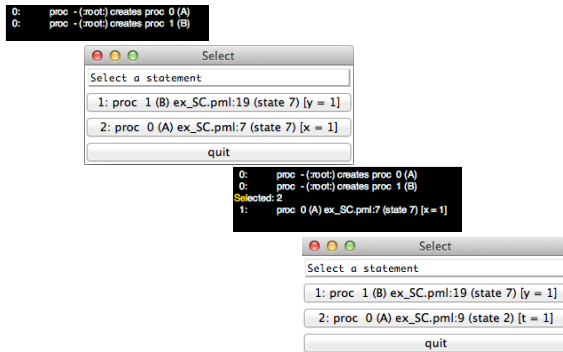
Simulation (Simulate / Replay) avec ispin

- Type de simulation
 - aléatoire
 - interactive
 - guidée
- Paramétrage de l'exécution
 - gestion des canaux (blocage ou perte lors d'écriture dans canal plein)
 - borne sur la longueur maximale de la séquence
- Affichages
 - fenêtres d'affichage des variables et des canaux
 - fenêtre MSC
 - déroulement de l'exécution
- Ne prouve pas que le programme est correct. Peut prouver qu'il est faux.

Paramétrage d'une simulation



Simulation interactive



Simulation aléatoire

- Exécute le nombre d'étapes déterminé par la borne max.



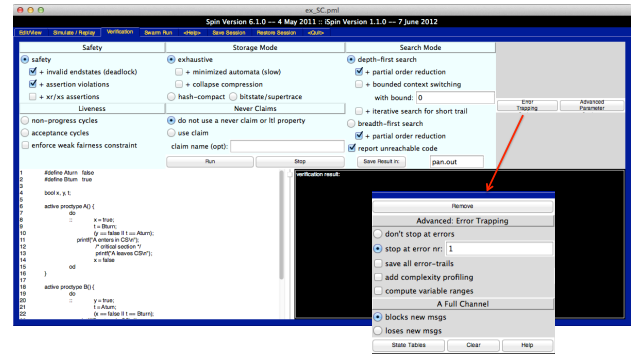
Si la borne est trop élevée, l'exécution est longue !

- On peut ensuite « rembobiner » (rewind) et revenir à une étape donnée
 - en cliquant sur la ligne correspondante dans le panneau d'exécution
 - en cliquant sur la boîte correspondante dans la fenêtre MSC
- A partir de ce point d'exécution, on peut ensuite avancer (step forward) ou reculer (step backward) pas à pas.

Vérification avec iSpin

- Permet de garantir qu'*aucune* exécution ne produira des situations indésirables :
 - Blocage
 - Plusieurs processus en section critique
 - Famine
- Classification des propriétés :
 - Basiques : terminaison, assertions
 - Sûreté : rien de mauvais ne se produira
 - Vivacité : Quelque chose de bien finira par arriver

Paramétrage de la vérification



Exemple d'assertion

- `assert (condition);`
 - N'est jamais bloquant
 - Renvoie une erreur si `condition` est fausse
 - La valeur de condition n'est évaluée qu'au moment de l'exécution de l'instruction → définir l'instruction dans un processus séparé si on veut l'évaluer systématiquement.
- `xr c;`
 - Le processus qui inclut cette assertion affirme qu'il a un accès exclusif au canal `c`. Toute lecture par un autre processus sera considérée comme une erreur.

Exemple

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

```
active [2] proctype P() {
    demande[_pid] = 1;
    do
        :: (tour != _pid) ->
            (Idemande[1- _pid]);
        tour = _pid;
        :: (tour == _pid) ->
            break;
    od;
    nb++;
    skip; /* SC */
    assert(nb == 1);
    nb--;
    demande[_pid] = 0;
}
```


Propriétés LTL (une introduction)

- LTL (*Linear Time Logic*) est une logique temporelle
- Ne prend pas en compte un temps quantifié, mais l'évolution du système
 - permet de définir des propriétés sur une séquence d'exécution
- Une formule est composée de propositions atomiques, opérateurs logiques, opérateurs temporels
- Des algorithmes efficaces permettent de valider une propriété LTL sur l'ensemble des exécutions du système

Exemples de propriétés

- Si p est vraie dans l'état courant, alors plus tard, q deviendra vraie

• → • → • **p** → • → • → • → • **q** → • → ...
- Lorsque p devient vraie, elle le reste jusqu'à ce que q le devienne

• → • → • **p** → • **p** → • **p** → • **p** → • **q** → • → ...
- Lorsque p devient vraie, elle le reste indéfiniment

• → • → • **p** → • **p** → • **p** → • **p** → • **p** → • **p** → ...
- ...

Formule LTL

- Propositions atomiques
 - Caractérisent un état ou un ensemble d'états, indépendamment de l'évolution du système
 - Expressions booléennes (\Rightarrow , $<$, $>$, $\&\&$, \parallel , ...) portant sur
 - Les variables
 - Les canaux
 - Les processus (opérateur @, variable _pid)
- Opérateurs logiques :
 - $\&\&$, \parallel , $!$, \Rightarrow (*implies*)
- Opérateurs temporels :
 - \parallel (*always*), \Diamond (*eventually*), U (*until*)
 - NB : eventually = dans le futur

Propriétés temporelles classiques

spin ne fournit plus de « template » pour les propriétés 🤔

Construire une propriété est difficile. Heureusement, beaucoup de propriétés relèvent de schémas classiques :

- **Invariance** : lorsque p devient vraie, elle le reste ensuite tout au long de l'exécution
- **Réponse** : chaque fois que p devient vraie, on passe ensuite par un état où q est vraie
- lorsque p devient vraie, elle le reste jusqu'à ce que q devienne vraie

Exemples

- s atteint l'état « terminé », et lorsqu'il l'atteint, il ne le quitte plus :

• → • → • → • → • → • → • → ...
eventually (always terminé)

- Si s atteint l'état terminé, il ne le quitte plus :

• → • → • → • → • → • → • → • → ...
 ou • → • → • → • → • → • → • → • → ...
eventually (terminé implies always terminé)

- Chaque fois que s demande la section critique, il reste demandeur jusqu'à l'obtenir (et il l'obtient)

• → • → • → • → • → • → • → • → • → • → • → • → • → ...
always (demande implies (demande until servi))

Peterson pour 2 processus

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

active [2] proctype user() {
    pid moi, lui;

    moi = _pid;
    lui = 1 - _pid;

do
    :: demande[moi] = true;
    ask_SC :
        tour = lui;
        /* attente SC */
        (demande[lui] == false || tour == moi);
    in_SC :
        nb++;
        assert(nb == 1);
        nb--;
        demande[moi] = false;
od
}
```

Absence de famine

- Tout processus qui demande la section critique finira par l'obtenir (propriété de *réponse*).
- Quel que soit l'état dans lequel se trouve l'application, si cet état correspond à une demande de SC, alors il sera suivi dans le futur d'un état où le processus demandeur est en SC.

• → • → • → • → • → • → • → • → • → • → • → • → • → ...

- Application à Peterson :

```
#define p (user[0]@ask_SC)
#define q (user[0]@in_SC)
always (p implies eventually q)
```

Vérification d'une formule LTL

- La formule doit être ajoutée au programme

– Ajout des macros :

```
#define p (user[0]@ask_SC)
#define q (user[0]@in_SC)
```

– Ajout de la formule :

```
ltl abs_famine { [] (p -> <> q) }
```

ou :

```
ltl abs_famine {
    always (p implies eventually q) }
```

- On coche ensuite l'option « use claim » lors du paramétrage de la vérification

Prise en compte de l'équité

- Spin examine **toutes** les séquences d'exécution possibles.
- Certaines ne correspondent pas à la réalité de l'environnement → introduction de la notion d'équité.
- **Équité faible (weak fairness)** : lorsque la prochaine action d'un processus est exécutable de manière permanente, elle finira par être exécutée.
- Équité forte : lorsque la prochaine action d'un processus est exécutable infiniment souvent, elle finira par être exécutée.

Exécution répartie et équité

- Non prise en compte de l'équité faible \Rightarrow un processus qui n'est pas en panne ne s'exécute jamais
- Ajouter cette hypothèse à la vérification est conforme au contexte
- Une propriété fausse dans un contexte non équitable peut devenir vraie dans un contexte équitable
- L'équité ne s'applique qu'entre des processus différents
 - une action qui est toujours en compétition localement avec une autre action peut ne jamais être choisie

Conclusion

- Spin permet la vérification de propriétés :
 - exclusion mutuelle
 - absence de famine
 - ...
- La vérification explore tous les comportements possibles d'une application (\neq simulation et \neq exécution)
 - gourmande en mémoire : attention au choix de représentation !
- Faire les bonnes hypothèses sur le contexte peut éviter la construction de séquences non significatives
 - le contexte est-il équitable ?