Vérification de programmes

Plan

1 Un exemple de système : le dîner des philosophes **1**
- 1.1 Contexte . . . 1
- 1.2 Modélisation . . . 2
- 1.3 Solution . . . 2

2 Principe du model checking **2**
- 2.1 Définition du model checking . . . 2
- 2.2 Modélisation du système . . . 2
- 2.3 Modélisation formelle . . . 3
- 2.4 Produit d'automates . . . 3
- 2.5 Application avec deux philosophes . . . 4

3 Mise en oeuvre avec Spin **4**
- 3.1 Modélisation à l'aide de Promela . . . 5
- 3.2 Solution correcte . . . 6

4 Expression de propriétés **7**
- 4.1 Expression des propriétés . . . 7
- 4.2 Automates de Büchi . . . 7
- 4.3 Model Checking LTL . . . 8
- 4.4 Vérification du modèle . . . 9
- 4.5 Que faire quand une erreur se produit . . . 9
- 4.6 Résultats . . . 10

5 Preuve de l'algorithme 2 **11**
- 5.1 Preuve mathématique . . . 11
- 5.2 Vers une autre approche . . . 11
- 5.3 Exemple en C++ . . . 11

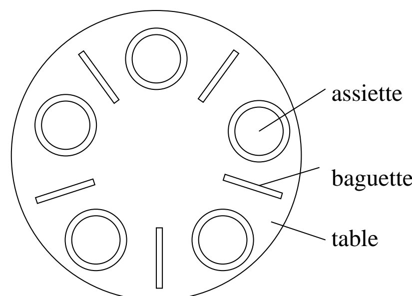
6 Exercices **13**

Ce cours vise à donner une vision partielle de certaines méthodes formelles permettant de s'assurer de la correction d'un système, d'un protocole représenté par un programme.

1 Un exemple de système : le dîner des philosophes

1.1 Contexte

Un certain nombre de philosophes, disons N sont assis à une table circulaire dans un restaurant chinois. Ci-dessous figure une image pour $N = 5$.



Il n'y a que N baguettes, une entre chaque philosophe. Pour manger un philosophe a besoin de deux baguettes : il doit prendre une baguette, puis l'autre, il peut alors manger, puis il s'arrête et repose ses baguettes. Chaque philosophe peut manger plusieurs fois.

1.2 Modélisation

On modélise le problème des philosophes généralement par N programmes s'exécutant simultanément. Les ressources partagées entre les programmes sont les baguettes. Les programmes sont souvent modélisés par des threads (instances d'exécutions).

L'algorithme d'un philosophe pourrait être par exemple :

Algorithme 1. *Etape 1 - Prendre la baguette droite si possible, sinon attendre.*
Etape 2 - Prendre la baguette gauche si possible, sinon attendre.
Etape 3 - Manger.
Etape 4 - Reposer les baguettes.
Etape 5 - Attendre un temps variable avant de repasser à l'étape 1.

Ce programme à l'air relativement simple. Cependant si N programmes sont exécutés simultanément cela peut conduire à un cas de blocage (deadlock en anglais) : chaque philosophe prend sa baguette droite et se retrouve alors bloqué dans l'étape 2 à jamais puisque aucune baguette gauche n'est alors disponible.

Nous remarquons qu'un assemblage de programmes simples peut être relativement complexe, il y a en effet peut-être d'autres cas de blocages que nous n'avons pas vus.

1.3 Solution

Avant de voir si un programme comporte des blocages donnons tout d'abord la solution au problème des philosophes.

Une solution fréquente consiste à numéroter les baguettes de 1 à N et d'utiliser l'algorithme suivant :

Algorithme 2. *Etape 1 - Prendre la baguette droite ou gauche avec le plus petit indice, si elle est disponible.*
Etape 2 - Prendre la baguette droite ou gauche avec le plus grand indice, si elle est disponible.
Etape 3 - Manger.
Etape 4 - Reposer les baguettes
Etape 5 - Attendre un temps variable avant de repasser à l'étape 1.

Cet algorithme est mathématiquement prouvé dans la section 5.

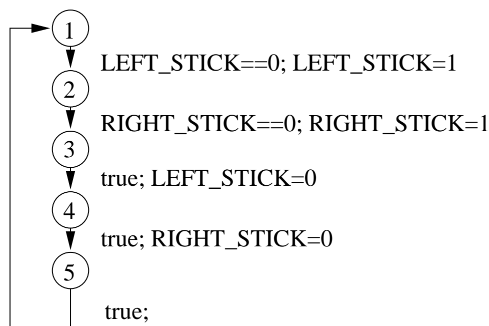
2 Principe du model checking

2.1 Définition du model checking

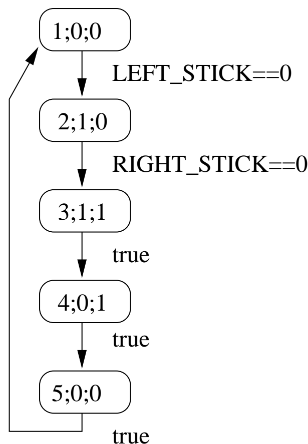
L'idée du model checking est d'avoir un modèle représentant un programme ou un système. Tous les états possible de ce modèle sont alors explorés pour vérifier une propriété. La difficulté est de trouver les algorithmes et le formalisme exprimant les propriétés permettant de vérifier le plus efficacement possible un système donné.

2.2 Modélisation du système

On peut abstraire les programmes sous forme d'automates. Les automates possèdent un état, ici par exemple les étapes 1 à 5. Puis il y a des transitions entre ces états. Les transitions sont constituées d'une garde (la condition qui permet de déclencher la transition) et une liste d'actions. Ainsi l'algorithme 1 est présenté ci-dessous :



La garde est séparée de la liste d'action par un point virgule. Il est à noter que l'on peut supprimer les actions en les incluant dans les états. Ainsi un état peut être représenté sous la forme d'un triplet. Par exemple l'état initial sera $1;0;0$ pour dire que le processus est dans l'étape 1 et que les variables `LEFT_STICK` et `RIGHT_STICK` sont à zéro. La première transition sans actions conduit alors à l'état $2;1;0$. La figure ci-dessous montre l'automate équivalent où les transitions ne portent plus que la condition de garde.



2.3 Modélisation formelle

Un tel automate est nommé structure de Kripke. On peut définir cette structure par l'énoncé suivant :

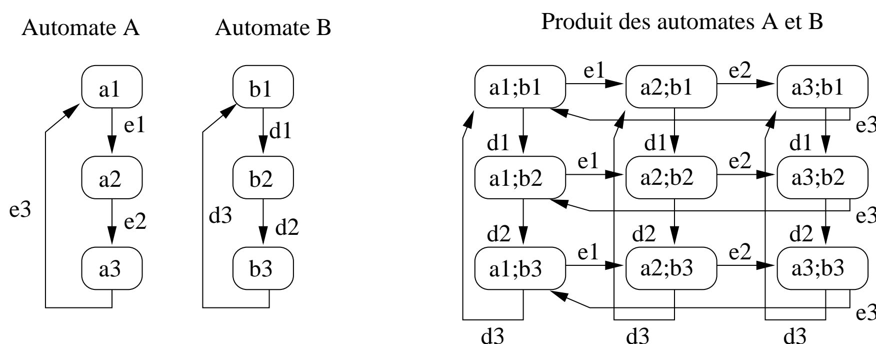
Définition 1. *On suppose donné un ensemble fini P de propriétés élémentaires. Soit un ensemble d'états Q, un ensemble d'étiquettes E, un ensemble de transitions $T \subseteq Q \times E \times Q$, un état q_0 de Q nommé état initial, et l l'application qui à tout état de Q associe l'ensemble fini des propriétés élémentaires vérifiées dans cet état. On nomme alors structure de Kripke le quintuplet (Q, E, T, q_0, l).*

Ainsi pour modéliser l'automate précédent on peut prendre :

- on se donne une propriété nommée `LEFT_STICK` et une autre nommée `RIGHT_STICK`. Elles seront vraies quand la baguette droite ou gauche est utilisée, ainsi $P = \{$`LEFT_STICK`, `RIGHT_STICK`$\}$.
- prenons $Q = \{q_0, q_1, q_2, q_3, q_4\}$.
- prenons $E = \{e1, e2, e3\}$ avec $e1 = true$, $e2 = \neg$`LEFT_STICK`, $e3 = \neg$`RIGHT_STICK`.
- l'ensemble T est défini par : $(q_0, e2, q_1)$, $(q_1, e3, q_2)$, $(q_2, e1, q_3)$, $(q_3, e1, q_4)$, $(q_4, e1, q_0)$.
- l'application l est définie par : $l(q_0) = \{\}$, $l(q_1) = \{$`LEFT_STICK`$\}$, $l(q_2) = \{$`LEFT_STICK`, `RIGHT_STICK`$\}$, $l(q_3) = \{$`RIGHT_STICK`$\}$, $l(q_4) = \{\}$,

2.4 Produit d'automates

Nous allons maintenant décrire le procédé modélisant l'exécution concurrente de processus :



On considère deux automates A et B. Le système composé de l'exécution parallèle de A et B possède des états de la forme $a_i;b_j$. Les transitions possibles résultent soit d'une exécution de A soit d'une exécution de B.

On voit ainsi que l'on peut définir formellement le produit de deux structure de Kripke :

Définition 2. *Etant donné deux structures de Kripke* (Q,E,T,q_0,l) *et* (Q',E',T',q'_0,l')*, on définit leur produit par* $(Q\times Q',E\cup E',T'',(q_0,q'_0),l'')$ *où :*

- T'' *est définit par :* $((q_1,q'_1),e,(q_2,q'_2))\in T''$ *si et seulement si* $(q_1,e,q_2)\in T$ *et* $q_2=q'_2$ *ou* $(q'_1,e,q'_2)\in T'$ *et* $q_1=q'_1$.
- l'' *est définit par* $l''(q_1,q'_1)=l(q_1)\cup l(q_2)$.

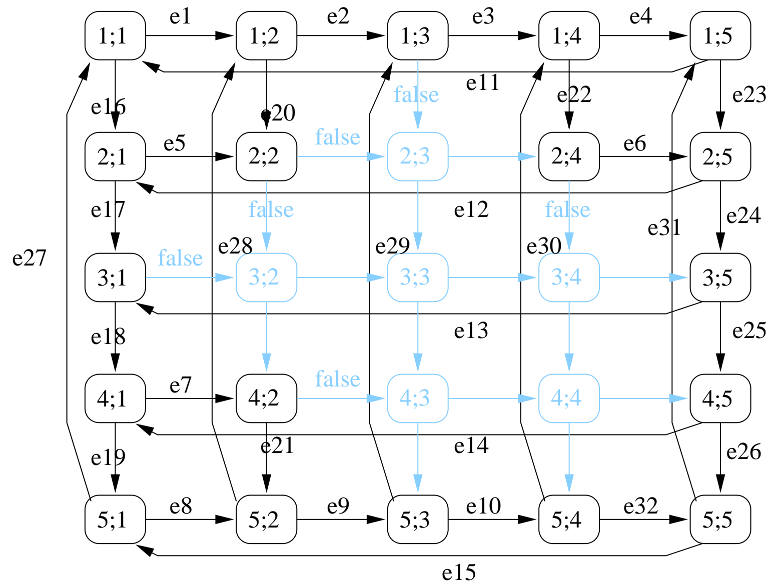
On en déduit alors la propriété évidente ci-dessous :

Propriété 1. *Le produit de deux structures de Kripke est une structure de Kripke.*

Ce résultat est important. Il nous dit que fondamentalement un programme (une structure de Kripke) est de la même nature qu'un programme multi threadé (un produit de structures de Kripke).

2.5 Application avec deux philosophes

On peut ainsi faire le produit des automates de la section 2.2 comportant des étiquettes comme illustré ci-dessous :



Le contenu des étiquettes n'est pas explicité. Il s'agit de l'étiquette correspondant à l'automate qui s'exécute : ainsi par exemple l'étiquette $e1$ correspondant à l'exécution du philosophe numéro 0 sera `LEFT_STICK[0]==0; LEFT_STICK[0]=1`. En revanche l'étiquette $e16$ correspondant à l'exécution du philosophe numéro 1 sera `LEFT_STICK[1]==0; LEFT_STICK[1]=1`.

On remarque qu'aucune transition ne sort de l'état $2;2$. Nous aurions cependant du rajouter entre autre la transition correspondant à l'exécution du philosophe 0 avec l'étiquette `RIGHT_STICK[0]==0; RIGHT_STICK[0]=1`. Or juste avant on a effectué l'action : `LEFT_STICK[1]=1`, or `RIGHT_STICK[0]` et `LEFT_STICK[1]` étant similaires la garde sera toujours fausse. Il n'y a donc pas de transitions sortant de l'état $2;2$. Cet état est bloquant, le terme anglais consacré est deadlock.

Ainsi on constate que certains états du système produit (ceux en bleu clair) ne sont jamais atteints : il s'agit d'états où les deux philosophes auraient en même temps la même baguette.

3 Mise en oeuvre avec Spin

Dans cette section nous présenterons l'usage du model checker Spin [Spi] [Hol97], sur le problème des philosophes.

3.1 Modélisation à l'aide de Promela

Nous donnons ici la description du système dans le langage Promela, qui est le formalisme de description des systèmes utilisés dans Spin.

Tout d'abord dans le modèle checking nous allons fixer une valeur pour *N* le nombre de philosophes. De manière similaire aux langages C et C++ la constante *N* est définie par :

```
#define N 11
```

Nous définissons les baguettes par un tableau de taille *N* qui va contenir des octets (byte en anglais) donnant l'état de la baguette (occupée ou libre).

```
byte stick[N];
```

Ainsi `stick[0]` vaut 1 si la baguette numérotée 0 est utilisée, et `stick[0]` vaut zéro si la baguette ne l'est pas. Ainsi pour chaque philosophe on peut définir sa baguette droite et sa baguette gauche par :

```
#define LEFT_STICK stick[_pid-1]
#define RIGHT_STICK stick[_pid%N]
```

Chaque philosophe va être modélisé par une instance d'exécution exécutant l'algorithme 1. Une instance d'exécution est nommée proctype en Promela. Voici le code :

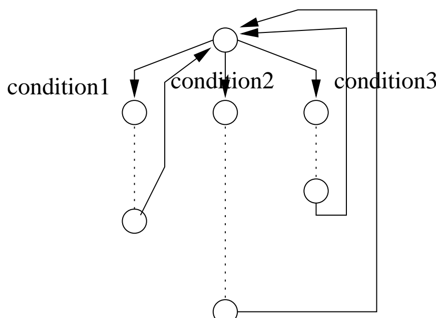
```
proctype Philo () {
 /* algorithme avec un dead lock */
 do
 :: d_step { LEFT_STICK == 0 ->  LEFT_STICK=1 };
    if
    :: d_step { RIGHT_STICK == 0 -> RIGHT_STICK=1};
       LEFT_STICK=0;
       RIGHT_STICK=0;
    fi;
 od;
}
```

Trois types de construction sont utilisés dans l'exemple ci-dessus :

1. boucles do : il s'agit d'une construction de la forme :

   ```
   do
   :: condition1 -> ..... ;
   :: condition2 -> ..... ;
   :: condition3 -> ..... ;
   od;
   ```

   C'est une boucle infinie. Si l'une des instruction après `::` est vraie alors la branche est exécutée. Une fois la branche exécutée on revient sur la boucle do, qui recommence à l'infini sauf si une instruction break est rencontrée dans une branche. Le point important qui diffère des langages de programmation impératifs habituels est le non déterminisme : si deux conditions sont simultanément vraies alors l'une ou l'autre peut être empruntée. La figure ci-dessous illustre l'automate correspondant à un `do` :



2. test if : il s'agit d'une construction de la forme :

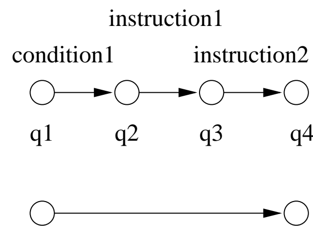
   ```
   if
   :: condition1 -> ..... ;
   :: condition2 -> ..... ;
   :: condition3 -> ..... ;
   if;
   instruction_suivante;
   ```

   le fonctionnement est similaire à celui des boucles do sauf qu'après l'exécution d'une branche c'est l'instruction suivante qui est exécutée.

3. l'instruction d_step (deterministic step) : elle permet de regrouper plusieurs instructions en une seule. Simplement la suite d'instruction doit être déterministe étant donné la condition initiale. Ainsi la séquence :

   ```
   d_step { condition1 -> instruction1; instruction2; ... }
   ```

   permet si la condition à l'entrée du d_step condition1 est vraie d'exécuter la suite d'instructions. La figure ci-dessous montre que pour une séquence d'instructions linéaire l'englober dans une déclaration d_step est équivalent à faire une seule transition. La seule condition étant bien entendu que la condition initiale suffise à déclencher les instructions 1 et 2.



Il nous faut en plus définir un processus initial qui va lancer les *N* autres:

```
init {
 atomic {
   int i=0;
   do
   :: i < N-1  -> run Philo(); i=i+1
   :: i == N-1 -> run Philo();   break
   od;
  }
  /* attendre sans fin */
  do
  :: 0;
  od;
}
```

3.2 Solution correcte

Voici la solution correcte où c'est la baguette de plus petit indice qui est prise en premier. On remarquera l'usage d'une variable nommée _pid (pour process identifier) qui donne le numéro correspondant à l'instance d'exécution courante.

```
proctype Philo () {
 /**
  * Solution correcte où la baguette de plus petite priorité est prise
  * en premier.
  */
 do
 :: d_step { _pid!=N && stick[_pid-1] == 0 ->   stick[_pid-1]=1 };
    if
```

```
    :: d_step { stick[_pid] == 0 ->  stick[_pid]=1};
       stick[_pid-1]=0;
       stick[_pid]=0;
    fi;
 :: d_step { _pid==N && stick[0] == 0 ->   stick[0]=1 };
    if
    :: d_step { stick[N-1] == 0 ->  stick[N-1]=1};
       stick[N-1]=0;
       stick[0]=0;
    fi;
 od;
}
```

4 Expression de propriétés

Maintenant que nous avons réalisé une modélisation du système à l'aide de Promela, il faut vérifier qu'elle correspond bien à nos attentes. Nous devons pour cela vérifier certaines propriétés comme par exemple : si un philosophe est en train de manger alors à un moment dans le futur il va poser ses baguettes.

4.1 Expression des propriétés

Il faut tout d'abord un langage permettant d'exprimer les propriétés que l'on souhaite vérifier, et pour lequel nous disposons d'un algorithme efficace.

Nous allons décrire le formalisme de la logique temporelle linéaire (LTL). Nous allons raisonner sur les chemins d'exécution. Un chemin d'exécution est une suite d'états d'une structure de Kripke représentant une exécution véritable. En voici une définition formelle :

Définition 3. *On appelle chemin d'exécution σ pour une structure de Kripke (Q,E,T,q_0,l) une suite potentiellement infinie $(\sigma(0),\sigma(1),...)$ d'états dans Q telle que :*

- $\sigma(0) = q_0$,
- *pour tout $i \in \mathbb{N}$ il existe $e \in E$ tel que $(\sigma(i),e,\sigma(i+1)) \in T$, et $\sigma(i)$ satisfait e, c'est à dire $e \in l(\sigma(i))$.*

Si $\sigma(i)$ vérifie la propriété P on note $\sigma,i \models P$, c'est à dire $P \in l(\sigma(i))$.
On va alors définir les opérateurs X, F, U et G pour exprimer les aspects temporels:

- on définit X (état suivant) par $\sigma,i \models \mathsf{X}P$, si et seulement si $\sigma,i+1 \models P$
- on définit U (until) par $\sigma,i \models P\mathsf{U}Q$, si et seulement si il existe j plus grand ou égal à i tel que pour tout k dans $[i,j[$ $\sigma,k \models P$ et $\sigma,j \models Q$

On peut alors définir deux autres opérateurs à partir du U (until) :

- on définit F (un moment dans le futur) par $\sigma,i \models \mathsf{F}P$, si et seulement si il existe j plus grand ou égal à i tel que $\sigma,j \models P$. $\sigma,i \models \mathsf{F}P$ est en fait équivalent à $\sigma,0 \models true\mathsf{U}P$.
- on définit G (toujours) par $\sigma,i \models \mathsf{G}P$, si et seulement si pour tout j plus grand ou égal à i on a $\sigma,j \models P$. $\sigma,i \models \mathsf{G}P$ est en fait équivalent à $\sigma,i \models \neg(\mathsf{F}\neg P) = \neg(true\mathsf{U}\neg P)$.

On peut composer les opérateurs. Ainsi pour exprimer qu'une exécution σ vérifie une infinité de fois une propriété P on peut écrire :

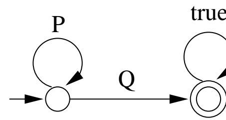
$$\sigma,0 \models \mathsf{GF}P$$

4.2 Automates de Büchi

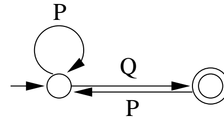
Un résultat important qui justifie l'usage de la logique précédente est présenté dans cette section : les formules de la logique LTL peuvent se coder sous forme d'automates de Büchi.

Définition 4. *Un automate de Büchi est un couple (K,F) où K est une structure de Kripke, et F un ensemble d'états, dits états terminaux de K. Une exécution d'un automate de Büchi est valide si elle passe par une infinité d'éléments de F.*

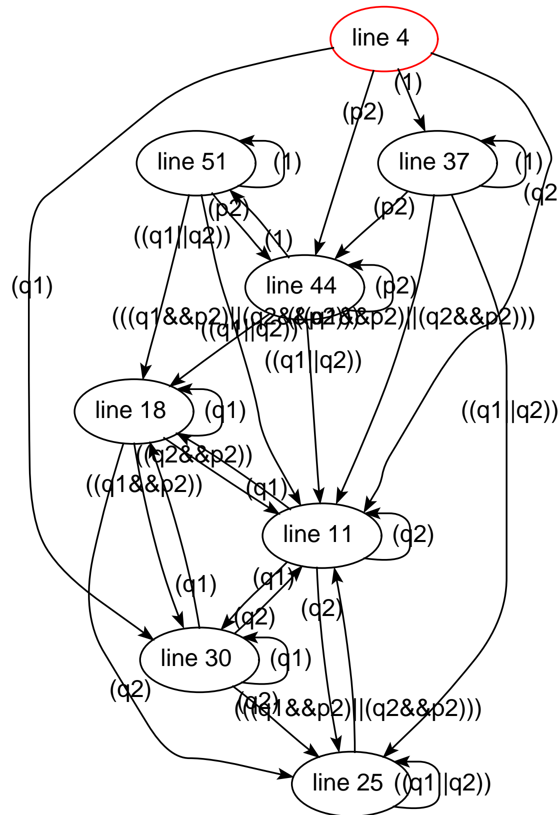
Ainsi la figure ci-dessous représente l'automate de Büchi pour $P\mathsf{U}Q$. L'état initial possède une flèche entrante, et l'état terminal possède un double cercle.



Voici l'automate représentant $\mathsf{G}(P\mathsf{U}Q)$:



Voici l'automate représentant $\mathsf{G}(\mathsf{F}(p1\mathsf{U}p2)\mathsf{U}\mathsf{G}(q1\mathsf{U}q2))$:



Voici une propriété qui n'est pas démontrée dans ce cours :

Propriété 2. *Toute formule LTL peut être représentée par un automate de Büchi.*

4.3 Model Checking LTL

Nous décrivons ici, sans le détailler comment vérifier qu'une structure de Kripke vérifie une formule LTL.

Imaginons maintenant que nous faisions le produit d'un automate de Büchi et de notre système. Si nous trouvons un état acceptant pour le produit c'est que l'on a trouvé une exécution du système d'origine dans laquelle $P\mathsf{U}Q$ est vraie.

En pratique les model checkers vérifient plutôt qu'il n'y a pas d'exécution possible, c'est à dire pas de chemins passant une infinité de fois par les états acceptants. Ainsi si on veut vérifier que le système satisfait $P \mathrm{U} Q$, on prend le l'automate de Büchi pour $\neg(P \mathrm{U} Q)$, on en fait produit synchronisé et on vérifie qu'aucun chemin ne passe une infinité de fois par les états acceptants.

Un programme générant l'automate de Büchi d'une formule LTL est fourni avec Spin. Une version plus performante [GO01] est réalisée par le programme LTL2BA à l'url suivante :

`http://www.liafa.jussieu.fr/~oddoux/`

Le model checking LTL comme décrit précédemment est dû essentiellement à Lichtenstein, Pnueli, Vardi et Wolper [VW86].

4.4 Vérification du modèle

Voici une application de la section précédente.

Il faut tout d'abord vérifier que le modèle que nous avons fait correspond bien à la réalité.

Par exemple quand un philosophe mange il doit tenir sa baguette droite et sa baguette gauche. C'est à dire la formule suivante doit être vraie : $p_1 \Rightarrow p_2$ où :

```
#define p1 (philoState[1]==2)
#define p2 (len(stick[1])==1 && len(stick[2])==1)
```

Pour vérifier la formule il suffit de taper en ligne de commande : `spin -f "! [] (p1 -> p2)"`. Cela va retourner un automate qui représente la propriété. Il suffit alors de copier coller cette formule dans notre description du système :

```
#define p1 (philoState[1]==2)
#define p2 (len(stick[1])==1 && len(stick[2])==1)

never {
T0_init:
        if
        :: (! ((p2)) && (p1)) -> goto accept_all
        :: (1) -> goto T0_init
        fi;
accept_all:
        skip
}
```

Les états acceptants de l'automate de Büchi sont ceux dont le nom commence par `accept_`. Nous pouvons alors lancer spin en invoquant la ligne de commande :

```
spin -a phi.spi
gcc -O2 -o pan pan.c
pan -a
```

Dans la sortie nous remarquons la ligne :

```
State-vector 60 byte, depth reached 185, errors: 0
```

Cela signifie que le modèle a été vérifiée et que la propriété est vraie.

4.5 Que faire quand une erreur se produit

Dans cette section nous vérifions le modèle avec le blocage. Il suffit de taper les commandes suivantes pour vérifier la présence de blocages :

```
spin -a phi.spi
gcc -O2 -o pan pan.c
pan
```

Dans la sortie on trouve alors :

```
State-vector 52 byte, depth reached 48, errors: 1
```

Il y a une erreur. Un fichier phi.spi.trail a été généré. On peut alors rejouer l'exécution du système qui conduit à l'erreur.

```
spin -v -t phi.spi
```

Ceci permet de comprendre sur un exemple concret la raison du blocage. On peut rechercher la profondeur minimum déclenchant l'erreur. Il faut pour cela lancer pan avec l'option -m200, pour se limiter à 200 pas de profondeur. Ici on remarque que la profondeur minimum est de 102 pas pour avoir l'erreur.

Ici c'est relativement facile, le résultat de l'exécution du système est :

```
....
 99:    proc  5 (Philo) line  23 "phi.spi" (state 3)   [((fork[(_pid-1)]==0))]
 99:    proc  5 (Philo) line  23 "phi.spi" (state 2)   [fork[(_pid-1)] = 1]
100:    proc  1 (Philo) line  23 "phi.spi" (state 3)   [((fork[(_pid-1)]==0))]
100:    proc  1 (Philo) line  23 "phi.spi" (state 2)   [fork[(_pid-1)] = 1]
spin: trail ends after 100 steps
#processes: 12
                fork[0] = 1
                fork[1] = 1
                fork[2] = 1
                fork[3] = 1
                fork[4] = 1
                fork[5] = 1
                fork[6] = 1
                fork[7] = 1
                fork[8] = 1
                fork[9] = 1
                fork[10] = 1
100:    proc 11 (Philo) line  24 "phi.spi" (state 11)
100:    proc 10 (Philo) line  24 "phi.spi" (state 11)
100:    proc  9 (Philo) line  24 "phi.spi" (state 11)
100:    proc  8 (Philo) line  24 "phi.spi" (state 11)
100:    proc  7 (Philo) line  24 "phi.spi" (state 11)
100:    proc  6 (Philo) line  24 "phi.spi" (state 11)
100:    proc  5 (Philo) line  24 "phi.spi" (state 11)
100:    proc  4 (Philo) line  24 "phi.spi" (state 11)
100:    proc  3 (Philo) line  24 "phi.spi" (state 11)
100:    proc  2 (Philo) line  24 "phi.spi" (state 11)
100:    proc  1 (Philo) line  24 "phi.spi" (state 11)
100:    proc  0 (:init:) line  68 "phi.spi" (state 12)
12 processes created
```

On remarque que tous les philosophes on pris une baguette. 12 processus ont été créés (11 pour les philosophes et 1 pour le processus init). Les philosophes sont tous dans l'état 11 (à la ligne 24 du fichier phi.spi).

Ce fichier est téléchargeable à l'url http://www.derepas.com/epita/mc.

4.6 Résultats

Voici le nombre d'états du système généré pour différentes valeurs de N :

N	temps (sec.)	nb. états
5	0.5	1204
10	3	1.7118e+06
11	16	6.82331e+06

Pour des valeurs de N plus grandes il faut plus de mémoire. Ici 1Gb de mémoire a été utilisé.

5 Preuve de l'algorithme 2

5.1 Preuve mathématique

Dans cette section nous donnons une preuve de correction de l'algorithme 2 décrit dans la section 1.3.

Propriété 3. *Pour un état du système de l'algorithme 2 si il y a une baguette sur la table alors cet état n'est pas un deadlock.*

Proof. Soit s un état du système de l'algorithme 2. Soit i l'index d'une baguette sur la table dans l'état s. Soit p_1 et p_2 les deux voisins partagent la baguette i. Notons j l'index de la baguette de p_1, et k l'index de l'autre baguette de p_2. De part la symétrie nous supposons $j < k$. Si un philosophe est dans l'étape 3, 4 ou 5 alors une transition vers un autre état est possible et s n'est donc pas un état bloquant. Ainsi nous supposons que tous les philosophes sont dans les états 1 ou 2 :

Il y a trois possibilités pour les valeurs de i, j et k:

- $i < j < k$, p_1 et p_2 ne peuvent être dans l'étape 2 puisque i est sur la table, Ils sont alors dans l'étape 1, ainsi chacun peut prendre la baguette i, et donc s n'est pas un état bloquant.
- $j < i < k$ p_2 ne peut pas être dans l'étape 2 car i est sur la table, il est donc dans l'étape 1, il peut alors prendre la baguette i, et donc s n'est pas un état bloquant.
- $j < k < i$ si p_1 ou p_2 sont dans l'étape 2 alors il peut prendre la baguette i et alors s n'est pas bloquant. Si p_1 et p_2 sont dans l'étape 1 alors :
  - Si il y a une baguette libre autre que i alors on est ramené à l'un des cas précédent où $i < j < k$ ou bien $j < i < k$, ce qui veut dire que s n'est pas bloquant.
  - Si il n'y a pas de baguette libre autre que i, alors puisqu'aucun philosophe n'est dans l'état 3, 4 ou 5 alors il y a $N-1$ baguette pour $N-2$ philosophe, ainsi un philosophe a deux baguettes, ce qui est impossible (personne dans l'état 3, 4 ou 5).

□

Propriété 4. *Il n'y a pas de blocage dans l'algorithme 2.*

Proof. Soit s un état du système dans l'algorithme 2. Si il y a une baguette sur la table alors d'après la propriété 3, s n'est pas un état bloquant.

Supposons maintenant qu'il n'y a pas de baguette sur la table dans l'état s. Si un philosophe est dans l'état 3, 4 ou 5 alors une transition vers un autre état est possible, et s n'est pas bloquant. Supposons maintenant que tous les philosophes soient dans l'état 1 ou 2. Puisqu'il y a N baguettes pour N philosophe, et qu'aucun philosophe ne possède deux baguettes, alors chaque philosophe a exactement une baguette. Notons p_1 le philosophe qui tient la baguette N. Puisque N est la baguette de plus haute priorité p_1 devrait posséder une autre baguette de priorité moindre. Ceci est impossible puisque p_1 n'a qu'une baguette. Cette situation est impossible.

Dans chaque état possible s n'est pas bloquant. □

5.2 Vers une autre approche

Une approche alternative au model checking qui se restreint à un N donné pourrait être d'effectuer sous forme informatique la preuve mathématique écrite précédemment. On serait ainsi vraiment sûr que pour tout N l'algorithme est bon. Il n'existe pas à l'heure actuelle d'algorithme permettant de générer automatiquement une preuve comme celle ci-dessus. Il faut donc tout d'abord rédiger une preuve manuelle, qui exige de comprendre finement les propriétés à vérifier et pourquoi elles le sont, puis de donner cette preuve à un logiciel pouvant vérifier la preuve.

De tels outils sont appelés assistants de preuve, puisqu'ils permettent simplement de vérifier la correction d'une preuve et non de trouver la preuve. On trouve par exemple [Coq] ou [Pvs].

5.3 Exemple en C++

Voici un exemple de tel mécanisme permettant de prouver que le nombre 13 est premier. Ce programme a été écrit par Dan Piponi (`http://www.sigfpe.com`).

Ici les entiers ne seront pas représentés directement sous forme de bits mais uniquement par les relations logiques des fondements mathématiques qui les définissent. En mathématique les entiers sont définis comme suit : on dispose d'un nombre nommé

O (la lettre “o” en majuscule) qui représente le nombre zéro. On définit l’opération S qui à un entier naturel associe son successeur, c’est à dire sa valeur incrémentée de 1. Ces deux affirmations constituent les axiomes de Péano et définissent les entiers naturels.

Définissons les entiers naturels dans un type nommé `Value` :

```
template<class V> struct Value { typedef V value; };
```

On peut alors définir zéro comme étant un type particulier dérivant de `Value` :

```
struct zero : public Value<zero> { };
```

On définit alors l’opérateur Successeur comme le type suivant :

```
template<class C> struct S
        : public Value<S<C> > { typedef C predecessor; };
```

Il s’agit d’un template, c’est à dire que pour un type donné par exemple `zero`, on peut créer le nouveau type ici `S<zero>`. On peut donc créer des types spécifiques pour les nombres jusqu’à 10 :

```
typedef S<zero> one;
typedef S<one> two;
typedef S<two> three;
typedef S<three> four;
typedef S<four> five;
typedef S<five> six;
typedef S<six> seven;
typedef S<seven> eight;
typedef S<eight> nine;
typedef S<nine> ten;
```

Comment dès lors définir l’addition entre deux nombres ? Il faut faire un template qui prend deux types et en renvoie. Le type résultant devra représenter le résultat et être plus facile à calculer :

```
template<class C,class D> struct plus
        : public S<plus<C,typename D::predecessor> > { };
```

Il nous faut rajouter un axiome de base qui spécifie que `zero` est l’élément neutre pour l’addition :

```
template<class C> struct plus<C,zero> : public C { };
```

De même on peut définir le moins :

```
template<class C,class D> struct minus
        : public minus<C,typename D::predecessor>::predecessor { };
template<class C> struct minus<C,zero>
        : public C { };
```

la multiplication :

```
template<class C,class D> struct times
        : public plus<C,typename times<C,typename D::predecessor>::value> { };

template<class C> struct times<C,zero>
        : public zero { };
```

Nous allons maintenant définir l’ordre sur les entiers avec l’opérateur `ge` (greater or equal) :

```
template<class C,class D> struct ge
        : public ge<typename C::predecessor,typename D::predecessor> { };

template<class C> struct ge<C,zero>
```

```
        : public one { };

template<class C> struct ge<zero,C>
        : public zero { };

template<> struct ge<zero,zero>
        : public one { };
```

Rajoutons alors un test de divisibilité :

```
template<class C,class D,class E = S<S<zero> > > struct Divisible { };

template<class C,class D> struct Divisible<C,D,S<S<zero> > >
        : public Divisible<C,D,typename ge<C,D>::value> { };
```

Si $C < D$ alors D divise C si et seulement si C est zero :

```
template<class C,class D> struct Divisible<C,D,zero>
        : public zero { };

template<class C> struct Divisible<zero,C,zero>
        : public one { };
```

Si $C \geq D$ alors D divise C si et seulement si D divise $C - D$

```
template<class C,class D> struct Divisible<C,D,S<zero> >
        : public Divisible<typename minus<C,D>::value,D> { };
```

On définit alors le test de primalité comme suit :

```
template<class C,class D> struct Prime<C,D,zero,zero,zero>
        : public Prime<C,D,one,zero,typename ge<D,C>::value> { };
// Test la division par D. Essayer avec le successeur en cas d'échec.
template<class C,class D> struct Prime<C,D,one,zero,zero>
        : public Prime<C,S<D>,zero,typename Divisible<C,D>::value,zero> { };
// Un facteur a été trouvé !
template<class C,class D> struct Prime<C,D,zero,one,zero>
        : public zero { };

// Aucun facteur n'a été trouvé !
template<class C,class D> struct Prime<C,D,one,zero,one>
        : public one { };
```

On définit alors un axiome permettant d'utiliser l'écriture décimale :

```
template<class C,class D> struct Decimal
        : public plus<typename times<ten,C>::value,D> { };
```

Il ne reste plus qu'à écrire deux fonctions d'affichage :

```
template<> char *output(zero) { return "No"; }
template<> char *output(one) { return "Yes"; }
```

Et pour prouver que 13 est premier il nous suffit d'écrire :

```
main() {
    puts(output(Prime<Decimal<one,three>::value>::value()));
}
```

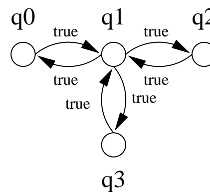
6 Exercices

Exercice 1

Représenter un automate de Büchi correspondant à la formule LTL $\neg(P \mathrm{U} Q)$.

Exercice 2

On se donne la structure de Kripke K ci-dessous :



On se donne deux propriétés P et Q. On suppose P vraie seulement dans les états q_0 et q_1 et Q vraie seulement dans q_2.

Prouvez en utilisant l'automate de l'exercice précédent que les exécutions de K ne vérifient pas $P\mathrm{U}Q$.

Caractérisez les deux types d'exécutions violant $P\mathrm{U}Q$.

References

[Coq] `http://coq.inria.fr`.

[GO01] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th Conference on Computer Aided Verification (CAV'01)*, number 2102 in Lecture Notes in Computer Science, pages 53–65. Springer Verlag, 2001. `http://www.liafa.jussieu.fr/~oddoux/CAV01.pdf`.

[Hol97] G. J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997. `http://spinroot.com/spin/Doc/ieee97.pdf`.

[Pvs] `http://pvs.csl.sri.com`.

[Sch99] Philippe Schnoebelen. *Vérification de logiciels*. Vuibert, 1999.

[Spi] `http://spinroot.com`.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. *Proc. First IEEE Symp. on Logic in Computer Science*, pages 322–331, 1986. `http://spinroot.com/spin/Doc/lics86.pdf`.

Le lecteur désirant un ouvrage en français accessible et relativement complet pourra se reporter à [Sch99].