Rapport de TER

Romain SALLES

26 mai 2009

Table des matières

Chapitre 1

Introduction

1.1 La vérification de programme

La vérification de programmes est au coeur de l'actualité informatique (en 2007, Edmund Clarke, Allen Emerson et Joseph Sifakis ont reçu le prix Turing pour leurs recherches sur ce thème).

Pour mieux comprendre la raison de ces recherches, voici un petit historique des "rattages" célèbres dus à des erreurs de programmation :

explosion de la fusée *Ariane V* (vol 501) : le programme comprenait la conversion d'un flottant vers un entier. Le flottant dépassait la valeur maximale pouvant être convertie, ce qui a causé une défaillance dans le système de positionnement. La fusée s'est alors détournée de sa trajectoire. Par mesure préventive, elle a déclenché son système d'auto-destruction. Coût de l'opération : 370 millions de dollars.

crash du réseau téléphonique d'*AT&T* : un *break* manquant dans un *switch* a entrainé une épidémie de redémarrages intempestifs causant 9h de pagaille et près de 70000 communications perdues (et par la même occasion, une sacrée réputation à la compagnie)

Le *Therac 25* : un accélérateur linéraire médical qui comportait plusieurs erreurs logicielles : certains patients ont reçu des doses radioactives mortelles. Bilan : 5 morts.

Ainsi, un logiciel mal contrôlé peut entrainer de graves répercussions matérielles, financières et mêmes humaines !

On comprend donc qu'il est nécessaire de contrôler ce type de programme pointu. Seulement, ce n'est pas toujours possible... Prenons par exemple un programme où l'utilisateur doit entrer un entier. Selon la machine, il pourra y avoir de 2^{32} à 2^{64} possibilités ; ce qui fera à n'en pas douter exploser les capacités de la machine s'il doit tester chacune de ces valeurs. Ajoutons maintenant des conditions, des entiers et des variables en tout genre et il est clair qu'il reste du chemin à parcourir avant de tester exhaustivement un programme.

Des solutions partielles à ce type de problème ont néanmoins été apportées :

- **programmation en respectant des règles de codage :** certains logiciels (**Checkstyle** pour Java,**MISRA-C** pour le C) permettent de contrôler qu'un code respecte cer-

taines normes de codage.

- **revue de programmes :** relire les codes en groupes de travail. Le programmeur explique et argumente ses parties de code. Certaines questions sont soulevées (*division par zéro ?*, *variables initialisées/déclarées ?*, ...)
- **batterie de tests :** on écrit une liste de tests la plus exaustive possible.
- **preuve assistée :** on donne les caractéristiques d'un programme (par exemple : *ne doit pas renvoyer de segfault*, ...) à l'aide de formules logiques à un outil qui vérifie leur validité. Cette preuve est formelle, donc n'a pas de faille. Son principal inconvénient : elle nécessite une grande intervention humaine (et comporte donc un risque d'erreur à ce niveau).
- **vérification déductive :** on vérifie directement le programme. Pour cela, on associe des invariants aux différentes lignes du programme et on s'assure qu'ils restent vrais tout au long des exécutions du programme.
- **résultats approchés :** avec de l'interprétation abstraite. Par exemple, l'*approximation conservative* consiste à prendre un sur-ensemble plus simple des états atteignables par le programme que l'on vérifie. On définit, comme pour la preuve assistée, une propriété qui doit être vérifiée (par exemple : *le programme ne doit pas faire de division par zéro*). Le programme de vérification peut alors répondre 2 choses :
  oui, la propriété est vérifiée
  je ne sais pas si la propriété est vérifiée
  Pourquoi ces deux réponses ?
  Si la propriété est vérifiée sur le sur-ensemble, alors par "contenance" la propriété est aussi vérifiée sur l'ensemble des états atteignables le pogramme a donc **prouvé** la propriété. Par contre, une réponse négative dira seulement qu'on *ne sait pas* si la propriété est falsifiée. En effet, on a deux cas possibles (cf. figure 1.1) :
  1er cas : l'ensemble des états qui falsifient la propriété a une *intersection vide* avec l'ensemble des états atteignables du programme (dans ce cas, *la propriété est vraie*)
  2eme cas : l'ensemble des états qui falsifient la propriété a une *intersection non vide* avec l'ensemble des états atteignables du programme (alors *la propriété n'est pas vérifiée*)
  Seulement, la distinction entre ces deux cas est **indécidable** : il est donc justifié que le programme réponde *je ne sais pas*.
- **model checking :** teste des propriétés sur des programmes. Dans le cas de programmes non bornés, il faut faire une approximation préalable. En pratique, ce modèle de vérification est souvent efficace.

Dans l'exercice de ce TER, nous nous tournerons vers la dernière méthode le **model checking**.

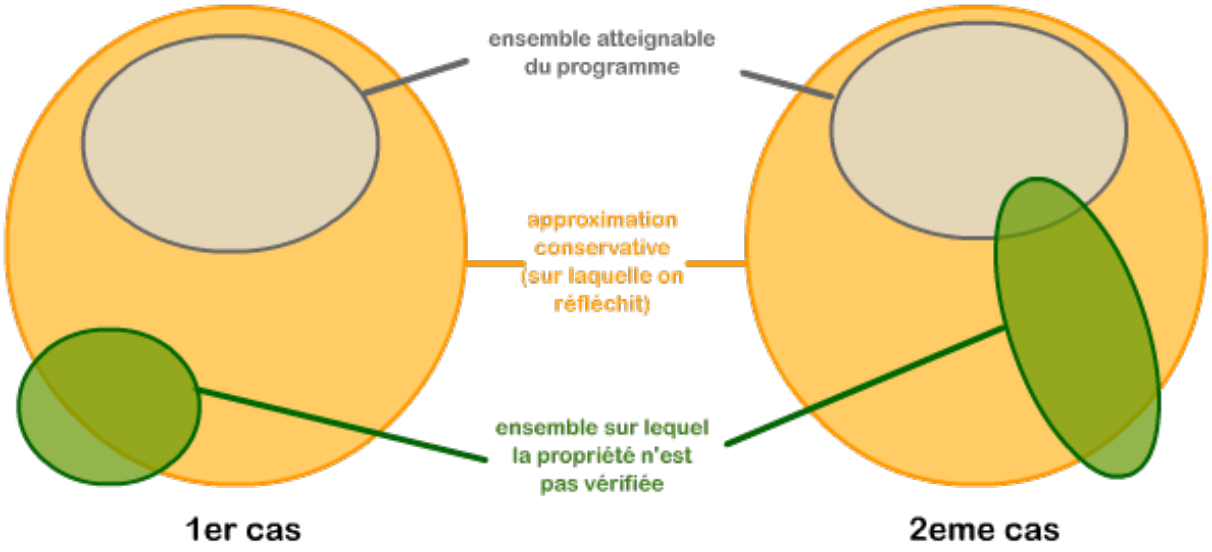


FIG. 1.1 – 2 cas possibles

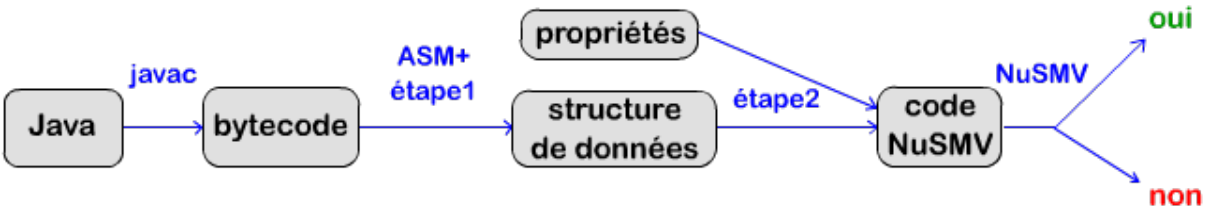


FIG. 1.2 – récapitulatif des étapes de la vérification

1.2 Le TER

1.2.1 contexte

On ne prétendra pas révolutionner le domaine du model checking. Ce projet a été pensé dans une optique pédagogique : c'est un exercice de style pour appréhender le métier de chercheur.

Néanmoins, ce TER a des objectifs concrets : construire un programme qui teste certaines propriétés sur des programmes Java (par exemple : *qu'il ne comporte pas de boucle infinie*).

Pour cela, on partira d'un code écrit en Java. Ce code, ainsi que les propriétés à vérifier, seront transformés en un automate compréhensible par le model checkeur **NuSMV** qui effectuera la vérification.

La figure 1.2 récapitule les différentes étapes de la vérification.

Plus précisément, on part d'un programme écrit en Java (bloc *Java* de la figure 1.2). Prenons par exemple :

```
boolean b = false;
boolean a = b | (b && !b);
if (a)
   a = false;
```

La ligne de code *javac* permet de compiler ce code et de le mettre sous la forme de *bytecode Java*. C'est une sorte de code assembleur portable. Voici la traduction du programme précédent en bytecode :

```
// boolean b = false;
     ICONST_0  // push(0)
     ISTORE 1  // pop(pile, variable1)
// boolean a = b | (b && !b);
     ILOAD 1   // push(variable1)
     ICONST_0  // push(0)
     IFEQ L0   // (si pile==1) saute en L0
     ILOAD 1   // push(variable1)
     IFNE L0   // (si pile==0) saute en L0
     ICONST_1  // push(1)
     GOTO L1   // saute en L1
   L0
     ICONST_0  // push(0)
   L1
     IOR   // pile-1 = OR(pile, pile-1)   et   pile--
     ISTORE 2  // variable2 = pile
// if(a)
     ILOAD 2   // push(variable2)
     IFEQ L2   // (si pile==1)  saute en L2
// a = false;
     ICONST_0  // push(0)
     ISTORE 2  // pop(variable2)
   L2
```

Ce code correspond à l'étape *bytecode* de la figure 1.2. Plus de détails sur la norme du bytecode Java sont disponibles sur le site de *Sun*(`http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html`).

A l'aide de la librairie *ASM*, nous analysons ce bytecode et nous le transposons dans une structure intermédiaire (bloc *structure de données* de la figure 1.2). Pour plus de détails sur la conception de cette structure intermédiaire, vous pouvez vous reporter à l'annexe 5.1

Sur la base de cette structure et des propriétés demandées par l'utilisateur de notre programme, nous construisons un automate compréhensible par le model checkeur *NuSMV* (bloc *code NuSMV* de la figure 1.2).

Il ne reste alors plus qu'à lancer NuSMV sur cet automate et vérifier que le programme testé possède les propriétés désirées.

1.2.2 objectifs

En résumé, ce programme :

part d'un vrai langage : le programme testé est codé en Java.

est petit : démontre qu'avec un petit programme, on peut faire du *model checking*.

réutilise l'existant : utilise le bytecode java, la librairie ASM pour l'interprétation du bytecode, ainsi que le programme NuSMV pour le model cheking.

utilise une structure intermédiaire : de telle sorte qu'on puisse utiliser plusieurs model checkeurs pour tester le programme

gère les threads : il est capable de tester un programme multi-threadé

fonctionne : il est capable de vérifier plusieurs propriétés sur des programmes (dead lock, boucle infinie, mauvaise gestion des sections critiques) et de donner un contre-exemple s'il y a lieu (le concepteur du programme pourra ainsi trouver la source du problème).

Ce programme a notamment été testé sur le problème de Dekker où les questions d'exclusion mutuelle et de deadlock [1] sont monnaie courante. Ainsi, sur de petits problèmes, le programme permet de trouver des erreurs qu'un humain ne verrait pas immédiatement.

1.2.3 contraintes

Il n'y avait que peu de temps, et que vu les objectifs fixés, les contraintes qui suivent découlent naturellement :

programme booléen : le programme ne pourra utiliser que des booléens

bytecode : le programme part du bytecode Java et ne construit pas l'arbre abstrait du code Java. Il utilise par ailleurs une librairie qui sait lire ce bytecode.

connexion à un outil existant : pour fonctionner, le programme a besoin d'un *model checkeur* extérieur (**NuSMV**) et ne peut fonctionner sans.

1.2.4 limitations

Enfin, le programme conçu dans le cadre de ce TER garde des limitations dus à des problèmes de :

consistance mémoire : Nous avons fait l'hypothèse que les actions d'écriture et de lecture de la mémoire des différents threads s'exécutent successivement sur une mémoire centralisée. Les actions d'un thread donné ont lieu dans l'ordre de son flot de contrôle, et la succession des actions sur la mémoire est un entrelacement des succession des actions des différents threads. En réalité, en raison des mécanismes de cache et de réordonnancement des instructions, ce n'est pas le cas, du moins si l'on n'utilise pas les primitives de synchronisation (barrières mémoire...). Nous avons cependant fait cette hypothèse simplificatrice car des modèles plus réalistes auraient été trop complexes.

[1] Interblocage d'un système du fait d'une boucle infinie, ou de deux processus s'attendant mutuellement.

réordonnancement du code par la machine : Afin d'optimiser l'utilisation du processeurs, certains programmes sont modifiés et les ordres d'exécutions des lignes modifiés lorsqu'elles sont jugées indépendantes. Ce type d'optimisation peut poser problème dans le cas où un *thread* guette les variations d'une variable pour continuer. Voici un exemple :
On suppose que *termine* est initialisé à *false* et *toto* à *true*.

thread 1	thread 2
toto = false ; (2)	while(!termine)
termine = true ;	return toto ;

Ici, inverser les lignes *(1)* et *(2)* pourra entrainer que le *thread 2* retourne *true* à la place de *false*, ce qui pourrait engendrer un bug inatendu dans la suite du code. Nous poserons l'hypothèse simplificatrice que **les lignes de code sont exécutées dans leur ordre d'écriture**.

Chapitre 2

Model checking

Nous allons désormais nous concentrer sur le **model checking**.

Ce terme désigne une famille de techniques de vérification automatique de programmes. Pour résumer, ces techniques vérifent algorithmiquement qu'un modèle donné (le système donné ou une abstraction de ce système) satisfait une spécification. Par exemple : *le programme ne devra jamais faire de dead lock* ou bien encore *les sections critiques ne pourront être accédées que par une seule portion de code à la fois*. Ces spécifications sont souvent formulées en terme de logique temporelle (chapitre *Temporal Logics* de **??**).

Les premiers travaux sur le model checking de formules de logique temporelle ont été menés par *Edmund M. Clarke* et *E. Allen Emerson* en 1981, ainsi que par *Jean-Pierre Queille* et *Joseph Sifakis* en 1982. *Clarke*, *Emerson* et *Sifakis* ont reçu le **Prix Turing 2007** pour leurs travaux sur le model checking.

2.1 les différentes méthodes de model checking

2.1.1 méthodes explicites

Une première méthode consiste à représenter le système sous la forme d'un automate dans lequel :

un noeud : représente un état du système

une transition : symbolise un changement d'état potentiel du système

Chaque noeud du graphe est étiqueté par l'ensemble des propositions atomiques vraies à ce point d'exécution. Voici un exemple d'automate associé au code suivant :

```
Random r = new Random(2);
int a = 3;
a = r.nextInt(0);
if (a) {
    a++;
} else {
    a--;
}
```

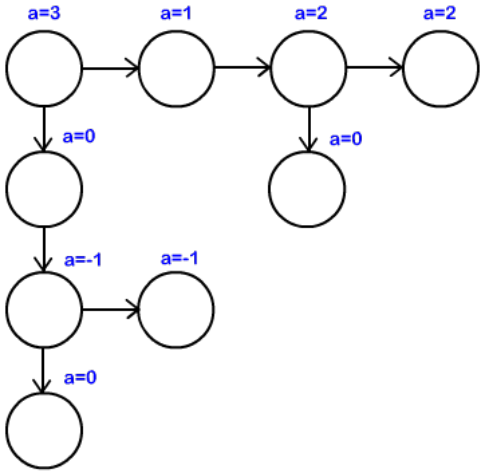


FIG. 2.1 – Exemple d'automate

a = a*r.nextInt(1) ;

Le programme va parcourir exhaustivement les états atteignables par le programme et vérifier que les propriétés sont vérifiées sur chacun d'eux.

2.1.2 méthodes symboliques

Les méthodes symboliques fonctionnent d'une autre manière et utilisent généralement des BDD (*Arbres de Décision Binaires*).

Un BDD est arbre binaire utilisé pour représenter une fonction booléenne.

Chaque noeud de décision du BDD est étiqueté par une variable booléenne et possède deux noeuds fils :

fils bas : représente l'affectation de la variable à 0

fils haut : représente l'affectation de la variable à 1

Les noeud terminaux sont *1* et *0* qui signifient que l'état correspondant appartient/n'appartient pas à l'ensemble. Pour plus de détails sur les BDD, vous pouvez vous reporter au chapitre *Binary Decision Diagram* de **??**).

La figure 2.2 représente le BDD de l'ensemble explicité à sa gauche.

Mais les model checkeur utilisant ce type de structure ne s'arrêtent pas là puisqu'ils optimisent ces BDD. Le graphe 2.3 peut par exemple être optimisé en regroupant les 2 zones bleues qui sont identiques. Ainsi, un BDD peut être **réduit**. En ajoutant un ordre sur les variables booléennes (sur la figure 2.3 : $x>y>z>t$), on obtient un ROBDD (Diagramme de Décision Binaire Ordonné Réduit). L'avantage d'un ROBDD est qu'il est **unique** : il est donc aisé de déterminer l'équivalence de 2 fonctions. Ainsi, une formule sera toujours vraie si son ROBDD est réduit au noeud racine **1**.

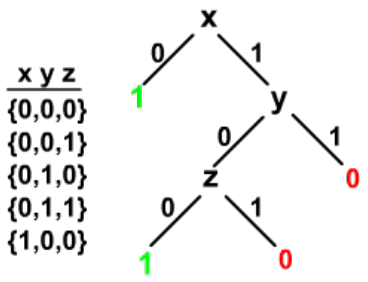


FIG. 2.2 – Exemple de BDD

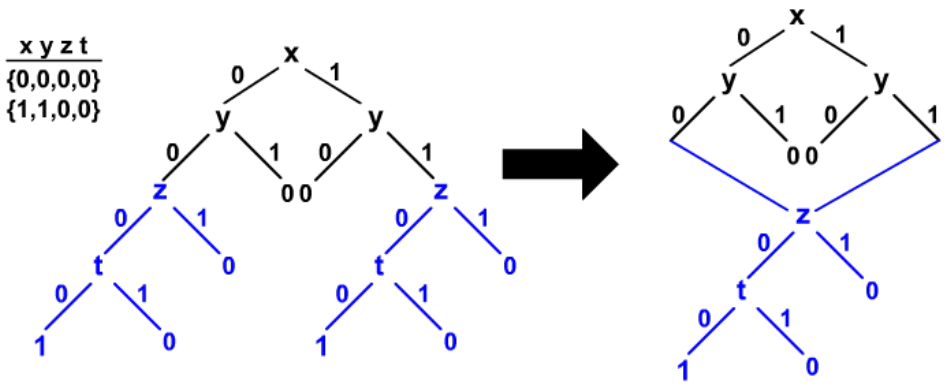


FIG. 2.3 – Exemple de ROBDD

2.1.3 SAT

Il existe enfin une autre méthode : au lieu de considérer l'ensemble des traces d'exécution du système, on peut se limiter à des traces finies, de longueur bornée. En effet, l'existance d'une trace de longueur finie est équivalente à la satisfaisabilité d'une formule de logique.

Ce type de preuve correspond au problème de la satisfiabilité d'une formule propositionnelle (problème SAT).

2.2 NuSMV

2.2.1 à la base était SMV

L'un des *model checkeur* les plus connu est **SMV** (Symbolic Model Verifier) : c'est le **premier** model checkeur **symbolique**. Il a été conçu par K. MacMillan.

Il prend en entrée des modules (=le système) et des formules CTL (Computational Tree Logic, cf chapitre *Temporal Logic* de **??**) où chaque état à plusieurs successeurs possibles. Si les propriétés sont vraies pour tous les états initiaux, il renvoie vrai. Sinon, il produit un contre-exemple.

2.2.2 NuSMV

SMV avait un inconvénient pour ses utilisateurs : il n'était pas open source. Un nouveau programme à vu le jour : **NuSMV**. Il est basé sur les mêmes idées que **SMV**. Seulement, comme il est open source, on peut aller regarder les sources en cas de bug.

Voici une présentation rapide de **NuSMV** :

synchrone/asynchrone

On peut l'utiliser à la fois en mode synchrone et asynchrone (à l'aide du mot clef *process*) : il va donc permettre de tester des programmes multi-threadés.

automates équationnels

C'est un automate équationnel : on écrira par exemple *next(var) = 3;* pour modifier une variable.

propriété CTL

Il utilise, comme SMV, des propriétés CTL.

Chapitre 3

Outil

3.1 présentation du programme

Dans le cadre du TER, nous avons décidé de développer un programme qui serait capable de vérifier différentes propriétés sur des programmes Java :

absence de boucle infinie ou de deadlock

bonne gestion des sections critiques

Par exemple, il sera capable de distinguer si ces différents codes sont corrects ou non (par correct, on entendra *pas de dead lock* et *gestion efficace des sections critiques*). Le **sc1** qui pourra apparaître dans les différents codes sert à annoncer une **section critique**.

prog1 : sections critiques mal gérées

```
static boolean occ = false;
static boolean sc1 = false;

void f() {
  while (occ) {}
  occ = true;
  sc1 = true;
  sc1 = false;
  occ = false;
}

void g() {
  while (occ) {}
  occ = true;
  sc1 = true;
  sc1 = false;
  occ = false;
}
```

prog2 : interblocage

```
static boolean sc1 = false;
static boolean demande_de_f = false;
static boolean demande_de_g = false;

void f() {
  demande_de_f = true;
  while(demande_de_g) {}
  sc1 = true;
  sc1 = false;
  demande_de_f = false;
}

void g() {
  demande_de_g = true;
  while(demande_de_f) {}
  sc1 = true;
  sc1 = false;
  demande_de_g = false;
}
```

prog3 : ok

```
static boolean attente = false;
static boolean sc1 = false;

void f() {
  sc1 = true;
  sc1 = false;
  attente = true;
}

void g() {
  while (!attente) {}
  sc1 = true;
  sc1 = false;
}
```

3.2 le fonctionnement du programme

Entrons dans le vif du sujet et regardons comment le programme fonctionne.

Pour cela, nous présenterons les différentes étapes de son exécution (cf figure 1.2). Nous expliquerons ensuite les choix d'implémentation qui ont été fait.

Nous supposons ici que le code à tester est déjà sous la forme de bytecode Java.

3.2.1 Etape 1 : lecture des instructions du bytecode Java*

La première étape consiste à transposer le bytecode Java en une structure de données que nous avons définie (5.1). Nous avons fait le choix de développer cette structure intermédiaire où cas où l'on voudrait utiliser un autre modèle de vérification. En effet, si l'on veut changer de mode de vérification, on n'aura plus qu'à adapter la traduction de cette structure de données (toute l'étude du bytecode aura déjà été effectuée). Cependant, cette structure intermédiaire reste totalement dépendante de la librairie ASM (`http://asm.ow2.org/`) puisqu'elle l'utilise pour son instantiation et son utilisation.

Pour traduire le bytecode en structure de données, le programme va parcourir l'ensemble des instructions du bytecode Java à l'aide de la librairie ASM*. Il va ensuite suivre 2 phases principales :

1 : associer un nombre à chaque instruction (une sortie d'identifiant unique)

2 : gérer les sauts (le GOTO de l'assembleur)

3.2.2 Etape 2 : finalisation

La deuxième étape consiste à traduire cette structure intermédiaire en un automate compréhensible par NuSMV, le model checkeur.

Le programme va commencer par coder un automate NuSMV pour chaque méthode. Puis il va entamer une procédure de finalisation où il va regrouper les automates des différentes méthodes dans un même fichier. Il ajoutera à ce fichier la gestion des sections critiques et des threads s'il y a lieu d'être. Enfin, il génèrera une fonction de transition par variable statique (le **next** de NuSMV) et ajoutera les expressions des propriétés recherchées (nous reviendrons dessus 3.3.11).

3.3 Les différents choix de conception

Plusieurs problèmes se sont posés lors de l'élaboration du programme : les voici accompagnés de leur solutions.

3.3.1 interprétation du programme Java : ObjectWeb ASM

le format d'entrée

La première question était de définir le format d'entrée du programme à tester.

En Java, 2 formats principaux sont disponibles :

code Java : le code tel qu'il est tapé par le programmeur.

bytecode Java : le code compilé par la machine. C'est ce format de fichier qui permet à Java d'être multiplateforme. Pour plus de détails, vous pouvez vous référer à **??**.

Le premier format aurait nécessité de construire un package entier pour l'interprétation du code. Dans le cadre du TER, il était clair que le temps ne le permettait pas.

On s'est donc tout naturellement tourné vers le bytecode Java. D'autant plus que plusieurs librairies font déjà le travail pour nous (ObjectWeb ASM, BCEL, JClassLib, ...). Nous avons choisi **ObjectWeb ASM** qui d'après de nombreux utilisateurs était plus rapide et efficace que ses concurrents.

ObjectWeb ASM

La librairie **ObjectWeb ASM** est composée d'interfaces à implémenter. Nous nous sommes principalement intéressés à une : **MethodVisitor**.

C'est cette interface qui est appelée lorsque la librairie explore une méthode. Nous avons récupéré une implémentation de cette interface codée directement dans **ObjectWeb ASM** et nous l'avons modifiée : nous lui avons ajouté la capacité de construire une liste chainée des instructions, et donc celle de créer une structure intermédiaire pour la traduction du bytecode Java.

L'étape 1 de la figure 1.2 est réalisée à l'aide de cette librairie.

3.3.2 une structure intermédiaire

Les données interprétées par **ObjectWeb ASM** sont enregistrées grâce au package **codeGeneration** (notre structure intermédiaire) que nous avons créé et qui trie les données utiles pour la suite.

On aurait pu se passer de ce package et faire la traduction pour le model checker à la volée. Cependant, si nous voulions changer de model checker ou même de méthode de vérification par la suite, il faudrait recommencer toute l'interprétation du bytecode à zéro.

Ici, puisque les données du bytecode sont stockées dans une structure intermédiaire, on pourra utiliser un autre programme sans trop de modifications en repartant du travail d'interprétation déjà effectué (notamment à l'aide patron de conception *Stratégie*).

3.3.3 un automate pour NuSMV

Comme nous l'avons vu précédemment, l'objectif de notre programme est de traduire le bytecode java en un automate utilisable par NuSMV.

Afin de comprendre la suite, regardons brièvement comme est écrit un automate sous NuSMV à travers ce petit exemple :

```
MODULE f
- - du code - -

MODULE main
```

```
VAR
    request : boolean ;
    state : ready, busy ;
    proc_1 : process f ;
    proc_2 : process f ;
ASSIGN
    init(state) := ready ;
    next(state) := case
       state = ready & request = 1 : busy ;
       1 : ready, busy ;
    esac ;
```

Les automates NuSMV sont composés de différents modules. Le seul étant exécuté par défaut étant *main*. Chaque module à ses variables propres qu'il déclare dans la section suivant le mot clef *VAR*. L'assignation de différentes valeurs à ses variables se fait dans la section annoncée par le mot clef *ASSIGN*.

On déclare ici 2 variables :

request : un booléen qui peut prendre les valeurs *0* et *1*.

state : une variable scalaire qui peut prendre les valeurs symboliques *ready* et *busy*.

La syntaxe **init** permet d'initialiser les variables

La syntaxe **next** permet de spécifier l'état suivant de la variable en fonction de son état courant (*1* étant le cas par défaut).

La syntaxe **process** permet de lancer un module en parallèle (c'est grâce à cette commande que nous gèrerons les thread).

Pour plus de détails sur la syntaxe NuSMV, vous pouvez vous reporter à son manuel utilisateur (`http://nusmv.fbk.eu/NuSMV/userman/v24/nusmv.pdf`).

Les bases étant posées, regardons les différents éléments de la traduction du bytecode en automate NuSMV.

3.3.4 les piles

Le bytecode Java, comme vu précédemment, est une machine à pile.

Dans un premier temps, nous avions décidé de coder l'automate sans cette notion de pile. En effet, elle n'était utilisée que pour les calculs ou les assignations. Nous avions donc décidé de construire une structure d'arbre qui permettait de reconstruire l'expression et de la coder en dur dans NuSMV (cf schéma 3.1). La recherche de l'arbre de l'expression a très bien fonctionné jusqu'à l'introduction des *ET/OU paresseux* qui sont gérés bien différemment des ET/OU classiques dans le bytecode.

Ces ET/OU paresseux sont une succession d'empilement suivis de tests sur la valeur obtenue après calcul. Dans le cas où le test est concluant, le bytecode effectue un saut sur l'instruction suivante sans continuer l'évaluation de l'expression booléenne. Il n'est alors plus possible de remonter de manière simple le bytecode pour construire un arbre de

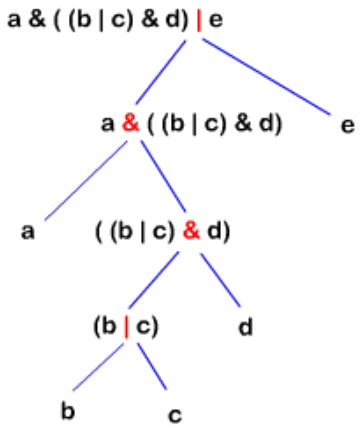


FIG. 3.1 – arbre d'expression

l'expression. La courte période de recherche liée au TER ne nous permettait de construire un tel outil, nous avons donc dû trouver une autre solution.

construction des piles

Celle qui s'imposait était de recontruire la pile dans NuSMV. Seul problème : NuSMV ne gère pas les piles.

On a utilisé une méthode brutale qui consiste, lors de la première étape de la traduction du bytecode 3.2.1, à évaluer la profondeur maximale qu'aura la pile d'une méthode (nous avons décidé de construire une pile par méthode). Cette taille calculée, nous assignons une variable par profondeur de pile. Ainsi, pour une pile de profondeur maximale de 3, on génèrera 3 variables de pile (prof_1, prof_2, prof_3). Les *push* et les *pop* reviennent donc maintenant à faire varier ces variables.

La question de la pile étant réglée, regardons maintenant le codage d'un état.

3.3.5 codage d'un état

Rappelons tout d'abord que chaque méthode du code d'origine est maintenant codée sous forme d'un module. Aussi, nous parlerons de manière indissociée de *méthode* et *module*.

L'état d'un module correspond à plusieurs choses :

une instruction courante : connue grâce à une variable qui contient le numéro de cette instruction (cf 3.2.1). Cette variable est propre à la méthode.

l'ensemble des valeurs des variables de la méthode : connues grâce a des *boolean* déclarés en début de module (le *VAR* de NuSMV).

l'etat de la pile : connu grâce à un ensemble de variables (il y en a autant que la profondeur maximale de la pile de la méthode : cf 3.3.4).

3.3.6 codage d'une transition

Le codage d'une transition sous NuSMV s'effectue en décrivant les valeurs des variables dans l'état suivant, connaissant l'état présent. Ainsi, pour déterminer les nouvelles valeurs de chaque variable, le programme va décrire une succession de cas (cf 3.3.6). La gestion du numéro de l'instruction courante ainsi que celle de la pile seront effectuées de la même manière.

Voici par exemple la gestion de la variable de pile de profondeur 2 du prog1 (cf 3.1) :

```
next(pile_f_2) := case
      state_f=0 : 1 ;
      state_f=2 : var_demande_de_g ;
      1 : pile_f_2 ;
esac ;
```

Ce qui se traduit par :

si state_f = 0 : alors, au prochain tour *pile_f_2* vaudra *1*

si state_f = 2 : alors, au prochain tour *pile_f_2* vaudra la même valeur que *var_demande_de_g*

dans le cas général : *pile_f_2* ne changera pas de valeur

3.3.7 gestion des méthodes

Chaque méthode est codée indépendamment des autres dans un *module* qui porte son nom. On y déclare l'ensemble des variables propres à la méthode (pour le moment, seulement des booléens) ainsi qu'une variable d'état qui contiendra le numéro de l'instruction courante de la méthode. C'est aussi dans ces modules que l'on gère les piles propres à chaque méthode (cf 3.3.4).

3.3.8 gestion des variables *static*

Les variables dites *static* sont déclarées et initialisées dans le module *main* de l'automate. Ces variables sont passées en paramètre aux modules des méthodes qui les modifieront de la même manière que leurs variables.

3.3.9 gestion des sections critiques

Les sections critiques doivent être encadrées par l'utilisateur à l'aide de booléens pour que notre programme puisse les détecter (l'utilisateur passe en paramètre de notre programme le nom de ces variables pour qu'on puisse les distinguer).

Dans le cas où l'on a une ou plusieurs sections critiques à surveiller, on déclare des instances d'un module **sectionCritique** (code en annexe 5.2) qui permettra de vérifier que 2 portions de codes ne rentrent pas en même temps dans une section critique identique. Le principe de ce module est simple : à tout instant, on compte le nombre de méthodes entrant en même temps dans la section critique. S'il s'avère qu'elles sont plus d'une, le booléen *bug* du module est passé à *1*.

On compte le nombre de méthodes entrant dans la section critique de la manière suivante : les modules générés pour chaque méthodes ont en paramètre l'ensemble des sections critiques. Ainsi, si a un instant donné ils doivent rentrer dans l'une d'elles, ils incrémentent la variable *cptrMutex* du module **sectionCritique** correspondant. Inversement, lorsqu'ils sortent de la section critique, ils décrémentent cette variable.

Dans le cas où cette variable vaut 1, une autre variable (*bug*) passe à *1* : on a détecté une mauvaise gestion de la section critique ! L'utilisateur de notre programme sera averti du problème grâce à la vérification d'une propriété explicitée plus loin 3.3.11.

3.3.10 gestion des threads

L'un des objectifs de notre programme est de détecter des *dead lock*, c'est à dire une portion de code qui en attend une autre éternellement.

Un cas évident étant celui où l'on a deux threads et que l'ordonnanceur décide de toujours donner la main au premier. Le 2ème sera donc dans un état bloqué. Pour éviter ce type de réponse triviale, nous ajouterons une hypothèse d'équité entre les threads dans l'automate généré (mot clef *FAIRNESS* dans NuSMV). Cette *FAIRNESS* garantit que *si un processus demande la main une infinité de fois, il l'aura une infinité de fois.* Ainsi, le programme suivant ne buggera jamais sur le cas où l'ordonnanceur laisserait toujours la main au premier thread (la variable *termine* est initialisée à *false*) :

thread 1	thread 2

while(!termine) {}	if(!termine) {
	  termine=true
	}

3.3.11 expression des propriétés sous NuSMV

Une fois lancé dans NuSMV, l'automate généré par notre programme, doit pouvoir signaler un cas de famine ou l'entrée simultannée de deux méthodes dans une section critique. Pour détecter ce type d'anomalie, il a fallu exprimer la bonne propriété sous NuSMV.

Voici un bref topo sur les propriétés de NuSMV :

les 4 opérateurs CTL principaux

Un model checker vérifie certaines propriétés. Sous **NuSMV**, ces propriétés sont exprimées à l'aide de 4 opérateurs CTL principaux :

E : *il existe un chemin*

A : *pour tous les chemins, à partir du début*

F : *éventuellement dans le futur*

G : *continuellement vrai*

Il est ensuite possible d'associer ces opérateurs pour traduire des propriétés.

exemples

AG !p : *la condition p n'est jamais vérifiée*
EF p : *il existe un chemin où, éventuellement dans le futur, p sera vérifiée*
AG EF p : *il est toujours possible de trouver un état où p est vraie*

Pour plus de renseignements sur la logique CTL, vous pourrez vous référer au chapitre *Temporal Logics* de [EMCP45].

Nous allons maintenant utiliser ces opérateurs pour gérer différents problèmes...

les sections critiques

Comme vu précédemment (3.3.9), les sections critiques sont symbolisées par l'utilisation de modules **sectionCritique** contenant une variable *bug*. Si à la fin de l'exécution de l'automate, cette variable vaut toujours *0*, c'est qu'il n'y a pas eu de problème au niveau des sections critiques. Au contraire, si une de ces variables passe à *1*, cela signifie que 2 méthodes sont entrées simultanément dans une section critique.

L'expression utilisée vérifie donc la propriété suivante : *dans tous les états de l'automate, peu importe le chemin choisi, toutes les variables bug des modules* ***sectionCritique*** *déclarés doivent être différentes de 1.* Ce qui se traduit sous NuSMV par :

$$SPEC\ AG\ !(sc\_sc_1.bug = 1 \mid ... \mid sc\_sc_n.bug = 1)$$

famine ou boucle infinie

Les propriétés de famine et de boucle infinie sont testées de la même manière par le programme. Seul l'utilisateur saura les distinguer. Elles se traduisent par le fait qu'une méthode ne finisse jamais. La détection de ce type de problème est simple : puisque nous connaissons le nombre d'instructions qui doivent être exécutées dans une méthode donnée, nous savons aussi qu'elle est la dernière de ces instructions. Ainsi, il nous suffit de vérifier que chaque méthode exécute un jour sa dernière instruction.

L'expression utilisée vérifie donc la propriété suivante : *Peu importe le chemin choisi, toutes les méthodes de l'automate doivent atteindre la dernière de leur instruction et l'exécuter.* Ce qui se traduit sous NuSMV par :

$$SPEC\ AF\ (\ proc\_meth_1.state = inst\_fin\_meth_1\ \&\ ...\ \&\ proc\_meth_n.state = inst\_fin\_meth_n\ )$$

3.4 Traces d'exécution du programme

Voilà les résultats obtenus si on lance notre programme sur le programme 1 décrit précédemment (3.1) :

automate généré : voir annexe 5.4.1
trace de NuSMV 2 : voir annexe 5.4.2

Chapitre 4

conclusion

4.1 Et après ?...

Les prochaines améliorations à apporter à ce programme sont les suivantes :

4.1.1 intégrer totalement NuSMV

Pour le moment, le programme ne fait que générer l'automate NuSMV pour le test. C'est l'utilisateur qui doit lancer derrière l'automate avec NuSMV.

Une première amélioration serait donc d'intégrer totalement NuSMV à notre programme. Ainsi, l'automate serait lancé lors de l'exécution de notre programme qui récupèrerait le message d'erreur. A partir de ce dernier, il indiquerait à l'utilisateur le type de problème ainsi que le numéro de la ligne du bytecode où le problème est apparu.

4.1.2 gestion des int

Notre programme ne gère pour le moment que les booléens. Il pourrait etre intéressant de lui ajouter les entier. Un premier pas serait de ne gérer que les entiers dont les valeurs seraient prédéterminées (autrement dit, pas de *random* ou d'autres choses du même style). Ainsi, on n'aura pas d'explosion combinatoire.

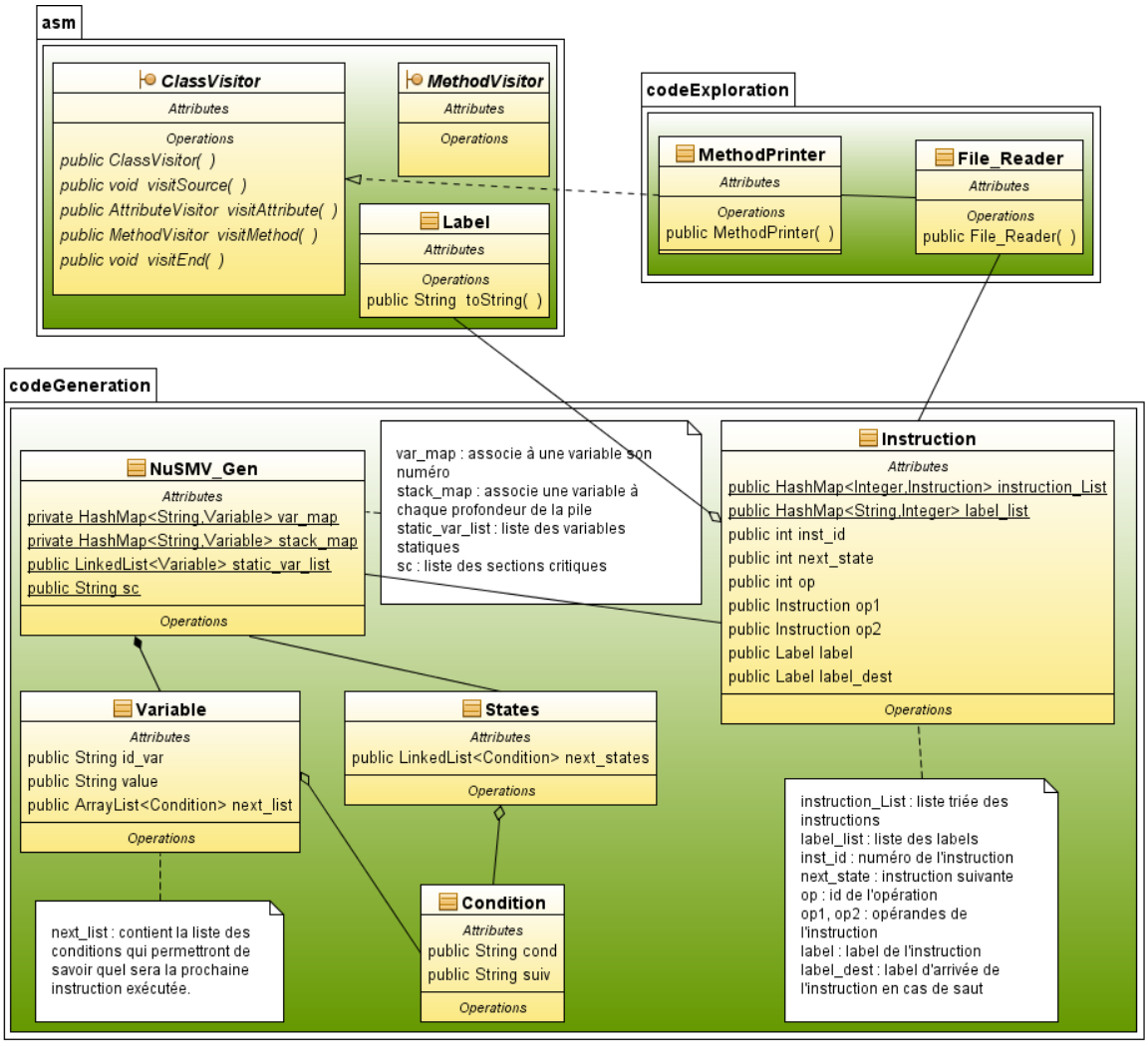
Il serait d'ailleurs intéressant de coupler notre programme à celui d'un étudiant de l'année dernière qui a conçu un programme qui détermine les valeurs d'un entier à tout moment dans un programme java (on pourrait ainsi déterminer les bornes à fixer sur ces entiers, ce qui est nécessaire lorsqu'on déclare un entier sous NuSMV).

4.2 remerciements

Chapitre 5

Annexe

5.1 Diagramme de classe de la structure intermédiaire



5.2 code de la section critique sous NuSMV

```
MODULE sectionCritique()
VAR
mutex: {unlocked, locked};
demandeEntree: boolean;
demandeSortie: boolean;
    cptrMutex: 0..1024;
bug: boolean;
ASSIGN
    init(cptrMutex) := 0;
    next(cptrMutex) := cptrMutex;

init(mutex) := unlocked;
next(mutex) := case
mutex=unlocked & demandeEntree=1 : locked;
mutex=locked & demandeSortie=1 : unlocked;
1             : mutex;
esac;

init(bug) := 0;
next(bug) := case
--demandeEntree & mutex=locked : 1;
        cptrMutex>1 : 1;
1 : bug;
esac;

init (demandeEntree) :=0;
next(demandeEntree) := 0;

init (demandeSortie) :=0;
next(demandeSortie) := 0;
```

5.3 Utilisation de notre programme

5.3.1 méthodes

On commence par donner en paramètre le nom des différentes méthodes qui seront lancées en parallèle.

5.3.2 sections critiques

On annonce les *sections critiques* à l'aide de l'option –**sc** puis on précise les différentes sections critiques.

5.3.3 exemple

java Main_ Prog f g –sc sc1 : va lancer les methodes **f** et **g** en parralèle et surveillera la section critique **sc1**.

5.4 traces d'exécution du programme 1

5.4.1 automate

```
MODULE sectionCritique()
VAR
mutex: {unlocked, locked};
demandeEntree: boolean;
demandeSortie: boolean;
    cptrMutex: 0..1024;
bug: boolean;
ASSIGN


    init(cptrMutex) := 0;
    next(cptrMutex) := cptrMutex;

init(mutex) := unlocked;
next(mutex) := case
mutex=unlocked & demandeEntree=1 : locked;
mutex=locked & demandeSortie=1 : unlocked;
1             : mutex;
esac;

init(bug) := 0;
next(bug) := case
--demandeEntree & mutex=locked : 1;
        cptrMutex>1 : 1;
1 : bug;
esac;

init (demandeEntree) :=0;
next(demandeEntree) := 0;

init (demandeSortie) :=0;
next(demandeSortie) := 0;


MODULE f (sc_sc1,var_occ,var_sc1)
```

```
VAR
   state_f : 0..11;
   pile_f_1 : boolean;
   pile_f_2 : boolean;
   pile_f_3 : boolean;
ASSIGN
   init(state_f) := 0;
   init(pile_f_1) := 0;
   init(pile_f_2) := 0;
   init(pile_f_3) := 0;


   next(var_occ) := case
         state_f=3 : pile_f_3;
         state_f=9 : pile_f_3;
         1 : var_occ;
     esac;
   next(var_sc1) := case
         state_f=5 : 1;
         state_f=7 : 0;
         1 : var_sc1;
     esac;
   next(sc_sc1.demandeEntree) := case
         state_f=5 : 1;
         1 : sc_sc1.demandeEntree;
     esac;
   next(sc_sc1.cptrMutex) := case
         state_f=5 & sc_sc1.cptrMutex<1024 : sc_sc1.cptrMutex + 1;
         state_f=7 & sc_sc1.cptrMutex>0 : sc_sc1.cptrMutex - 1;
         1 : sc_sc1.cptrMutex;
     esac;
   next(sc_sc1.demandeSortie) := case
         state_f=7 : 1;
         1 : sc_sc1.demandeSortie;
     esac;
   next(pile_f_1) := case
         1 : pile_f_1;
     esac;
   next(pile_f_2) := case
         state_f=0 : var_occ;
         1 : pile_f_2;
     esac;
   next(pile_f_3) := case
         state_f=2 : 1;
```

```
        state_f=4 : 1;
        state_f=6 : 0;
        state_f=8 : 0;
        1 : pile_f_3;
      esac;


   next(state_f) := case
                  state_f=0 : 1;
                  state_f=1 & !pile_f_2 : 2;
                  state_f=1 & pile_f_2 : 0;
                  state_f=2 : 3;
                  state_f=3 : 4;
                  state_f=4 : 5;
                  state_f=5 : 6;
                  state_f=6 : 7;
                  state_f=7 : 8;
                  state_f=8 : 9;
                  state_f=9 : 10;
        1 : state_f;
      esac;


MODULE g (sc_sc1,var_occ,var_sc1)
VAR
   state_g : 0..11;
   pile_g_1 : boolean;
   pile_g_2 : boolean;
   pile_g_3 : boolean;
ASSIGN
   init(state_g) := 0;
   init(pile_g_1) := 0;
   init(pile_g_2) := 0;
   init(pile_g_3) := 0;


   next(var_occ) := case
         state_g=3 : pile_g_3;
         state_g=9 : pile_g_3;
         1 : var_occ;
      esac;
   next(var_sc1) := case
         state_g=5 : 1;
         state_g=7 : 0;
```

```
      1 : var_sc1;
  esac;
next(sc_sc1.demandeEntree) := case
      state_g=5 : 1;
      1 : sc_sc1.demandeEntree;
  esac;
next(sc_sc1.cptrMutex) := case
      state_g=5 & sc_sc1.cptrMutex<1024 : sc_sc1.cptrMutex + 1;
      state_g=7 & sc_sc1.cptrMutex>0 : sc_sc1.cptrMutex - 1;
      1 : sc_sc1.cptrMutex;
  esac;
next(sc_sc1.demandeSortie) := case
      state_g=7 : 1;
      1 : sc_sc1.demandeSortie;
  esac;
next(pile_g_1) := case
      1 : pile_g_1;
  esac;
next(pile_g_2) := case
      state_g=0 : var_occ;
      1 : pile_g_2;
  esac;
next(pile_g_3) := case
      state_g=2 : 1;
      state_g=4 : 1;
      state_g=6 : 0;
      state_g=8 : 0;
      1 : pile_g_3;
  esac;


next(state_g) := case
               state_g=0 : 1;
               state_g=1 & !pile_g_2 : 2;
               state_g=1 & pile_g_2 : 0;
               state_g=2 : 3;
               state_g=3 : 4;
               state_g=4 : 5;
               state_g=5 : 6;
               state_g=6 : 7;
               state_g=7 : 8;
               state_g=8 : 9;
               state_g=9 : 10;
      1 : state_g;
```

```
     esac;


MODULE main
VAR
   proc_f : process f(sc_sc1,var_occ,var_sc1);
   proc_g : process g(sc_sc1,var_occ,var_sc1);
   sc_sc1 : process sectionCritique();
   var_occ : boolean;
   var_sc1 : boolean;
ASSIGN
   init(var_occ) := 0;
   init(var_sc1) := 0;


   next(var_occ) := var_occ;
   next(var_sc1) := var_sc1;




FAIRNESS
      proc_f.running
FAIRNESS
      proc_g.running

SPEC AG !(sc_sc1.bug=1)
SPEC AF ( proc_f.state_f=10 & proc_g.state_g=10  )
```

5.4.2 trace de NuSMV

```
-- specification AG !(sc_sc1.bug = 1)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  proc_f.state_f = 0
  proc_f.pile_f_1 = 0
  proc_f.pile_f_2 = 0
  proc_f.pile_f_3 = 0
  proc_g.state_g = 0
  proc_g.pile_g_1 = 0
  proc_g.pile_g_2 = 0
  proc_g.pile_g_3 = 0
```

```
  sc_sc1.mutex = unlocked
  sc_sc1.demandeEntree = 0
  sc_sc1.demandeSortie = 0
  sc_sc1.cptrMutex = 0
  sc_sc1.bug = 0
  var_occ = 0
  var_sc1 = 0
-> Input: 1.2 <-
  _process_selector_ = proc_f
  running = 0
  sc_sc1.running = 0
  proc_g.running = 0
  proc_f.running = 1
-> State: 1.2 <-
  proc_f.state_f = 1
-> Input: 1.3 <-
  _process_selector_ = proc_f
  running = 0
  sc_sc1.running = 0
  proc_g.running = 0
  proc_f.running = 1
-> State: 1.3 <-
  proc_f.state_f = 2
-> Input: 1.4 <-
  _process_selector_ = proc_f
  running = 0
  sc_sc1.running = 0
  proc_g.running = 0
  proc_f.running = 1
-> State: 1.4 <-
  proc_f.state_f = 3
  proc_f.pile_f_3 = 1
-> Input: 1.5 <-
  _process_selector_ = proc_g
  running = 0
  sc_sc1.running = 0
  proc_g.running = 1
  proc_f.running = 0
-> State: 1.5 <-
  proc_g.state_g = 1
-> Input: 1.6 <-
  _process_selector_ = proc_f
  running = 0
  sc_sc1.running = 0
```

```
  proc_g.running = 0
  proc_f.running = 1
-> State: 1.6 <-
  proc_f.state_f = 4
  var_occ = 1
-> Input: 1.7 <-
  _process_selector_ = proc_f
  running = 0
  sc_sc1.running = 0
  proc_g.running = 0
  proc_f.running = 1
-> State: 1.7 <-
  proc_f.state_f = 5
-> Input: 1.8 <-
  _process_selector_ = proc_g
  running = 0
  sc_sc1.running = 0
  proc_g.running = 1
  proc_f.running = 0
-> State: 1.8 <-
  proc_g.state_g = 2
-> Input: 1.9 <-
  _process_selector_ = proc_g
  running = 0
  sc_sc1.running = 0
  proc_g.running = 1
  proc_f.running = 0
-> State: 1.9 <-
  proc_g.state_g = 3
  proc_g.pile_g_3 = 1
-> Input: 1.10 <-
  _process_selector_ = proc_g
  running = 0
  sc_sc1.running = 0
  proc_g.running = 1
  proc_f.running = 0
-> State: 1.10 <-
  proc_g.state_g = 4
-> Input: 1.11 <-
  _process_selector_ = proc_g
  running = 0
  sc_sc1.running = 0
  proc_g.running = 1
  proc_f.running = 0
```

```
-> State: 1.11 <-
  proc_g.state_g = 5
-> Input: 1.12 <-
  _process_selector_ = proc_g
  running = 0
  sc_sc1.running = 0
  proc_g.running = 1
  proc_f.running = 0
-> State: 1.12 <-
  proc_g.state_g = 6
  sc_sc1.demandeEntree = 1
  sc_sc1.cptrMutex = 1
  var_sc1 = 1
-> Input: 1.13 <-
  _process_selector_ = proc_f
  running = 0
  sc_sc1.running = 0
  proc_g.running = 0
  proc_f.running = 1
-> State: 1.13 <-
  proc_f.state_f = 6
  sc_sc1.cptrMutex = 2
-> Input: 1.14 <-
  _process_selector_ = sc_sc1
  running = 0
  sc_sc1.running = 1
  proc_g.running = 0
  proc_f.running = 0
-> State: 1.14 <-
  sc_sc1.mutex = locked
  sc_sc1.demandeEntree = 0
  sc_sc1.bug = 1
-- specification AF (proc_f.state_f = 10 & proc_g.state_g = 10)  is true
```

Bibliographie

[EMCP45] Orna Grumberg Edmund M. Clarke, Jr. and Doron A. Peled. *Model Checking*. 1945.