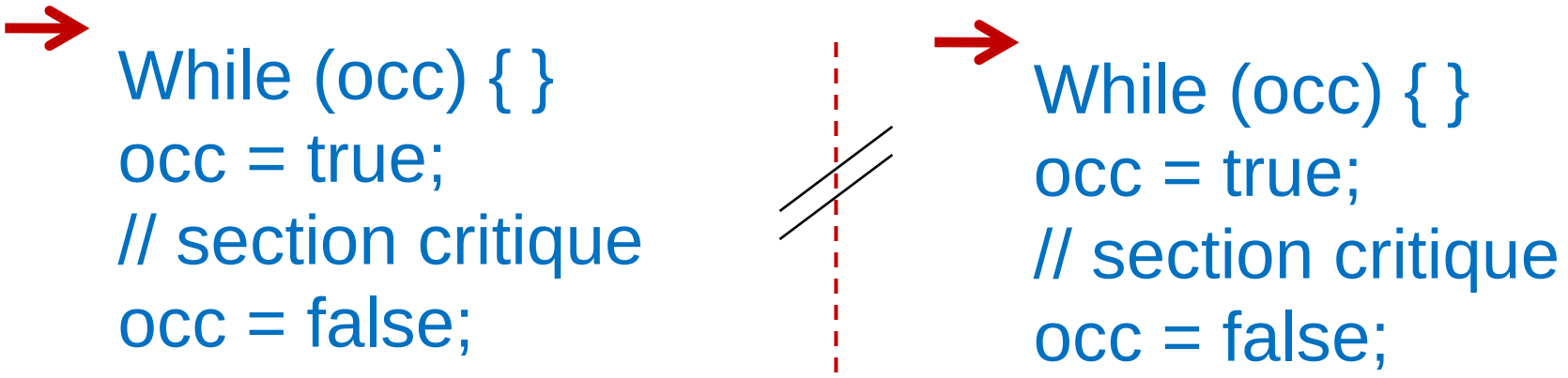
TER 2009

Model-checking de programmes Java

Romain SALLES

Vérifier un programme ? pas si facile que ça ...

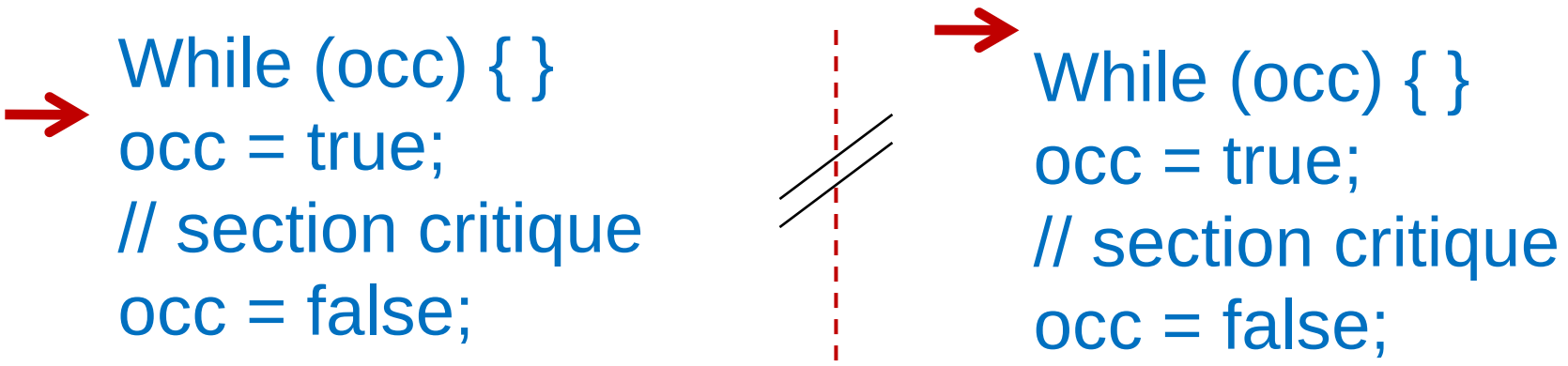
occ = false;



// suite du code

Vérifier un programme ?
pas si facile que ça ...

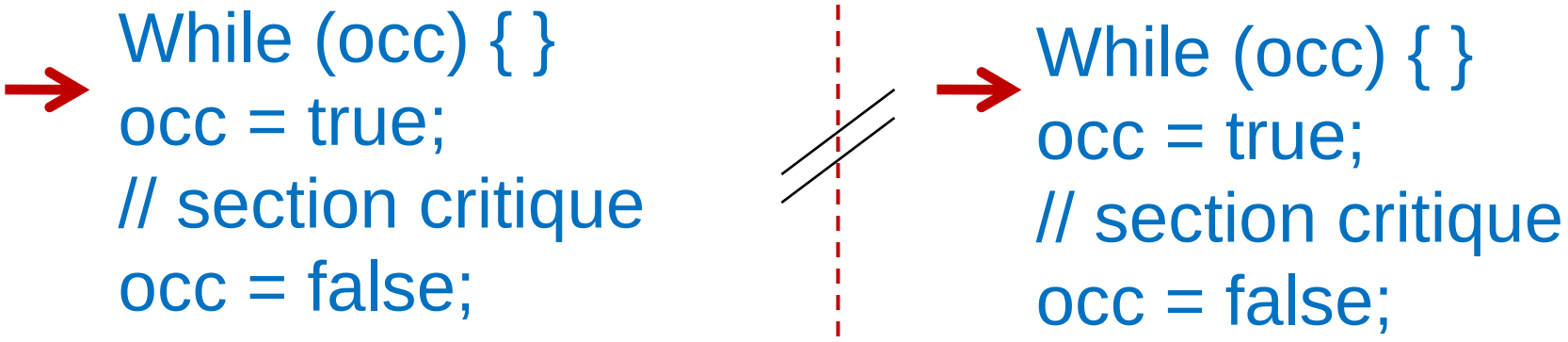
occ = false;



// suite du code

Vérifier un programme ? pas si facile que ça ...

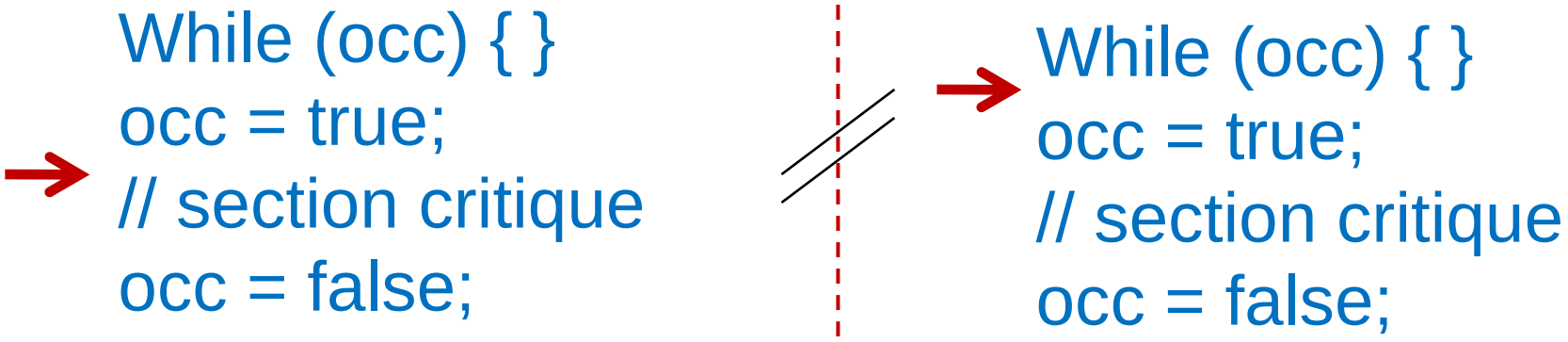
occ = false;



// suite du code

Vérifier un programme ? pas si facile que ça ...

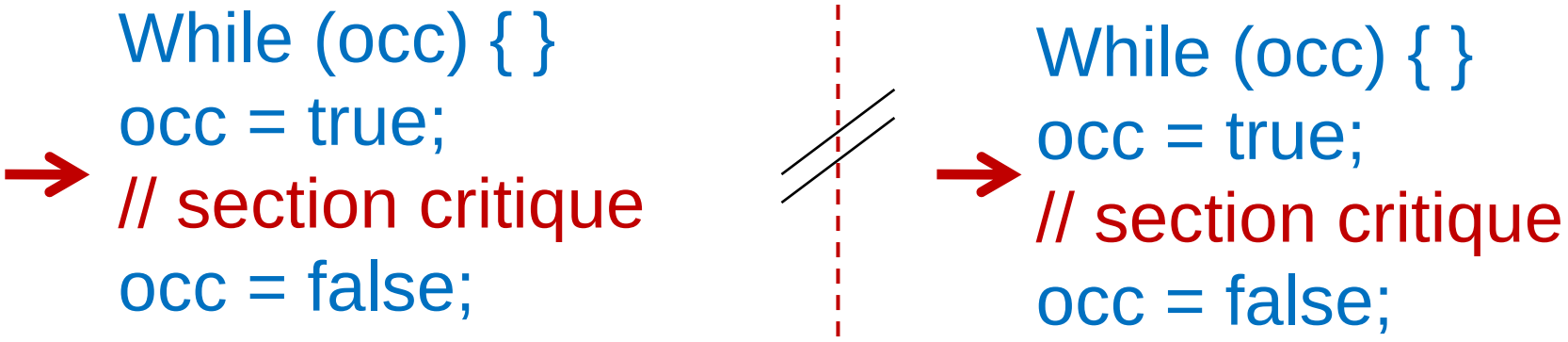
occ = false;



// suite du code

Vérifier un programme ? pas si facile que ça ...

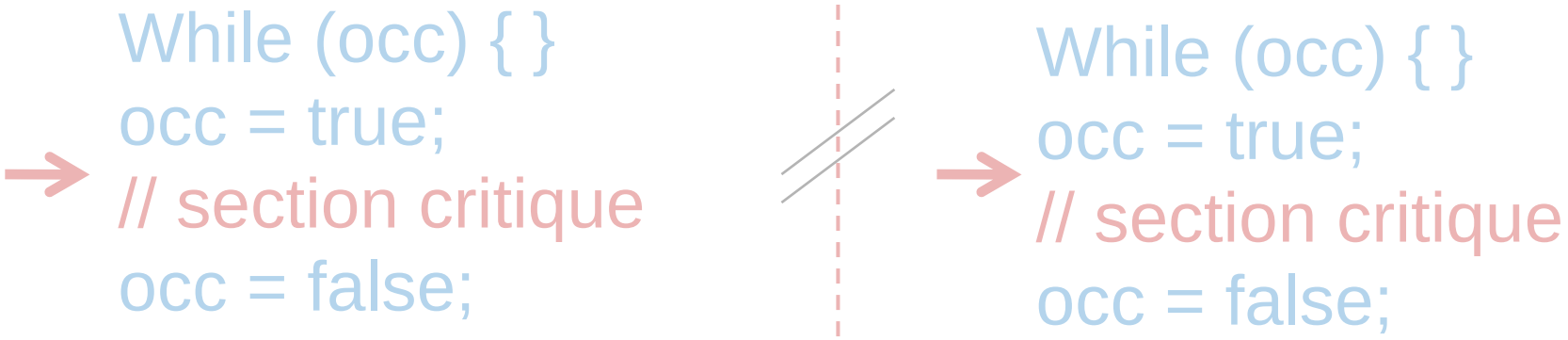
occ = false;



// suite du code

Vérifier un programme ? pas si facile que ça ...

occ = false;



// suite du code

ERREUR

Spécifier des contraintes

Logique temporelle :

- **E**(xists) : il existe une exécution
- **G**(lobaly) : pour toutes les exécutions
- **F**(uture) : dans le futur
- **A**(lways) : à tout instant

Solutions

MANUELLES
- revue de programmes
- batterie de tests

INFORMATISEES
- preuve assistée
- model checking

MODEL CHECKING

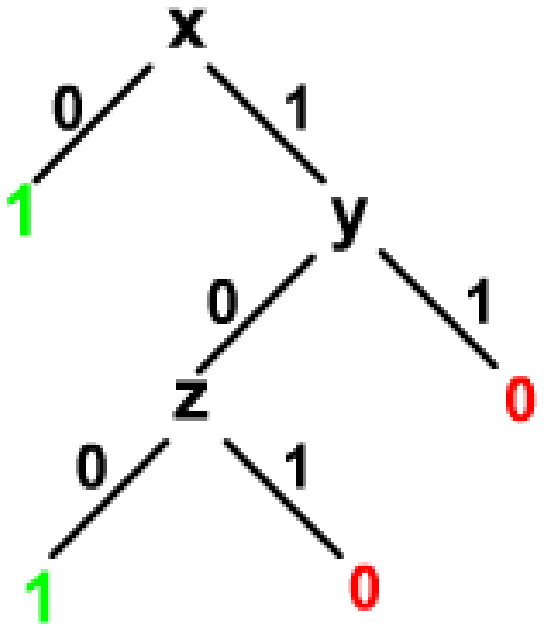
MODEL CHECKING

programme Machine d’états

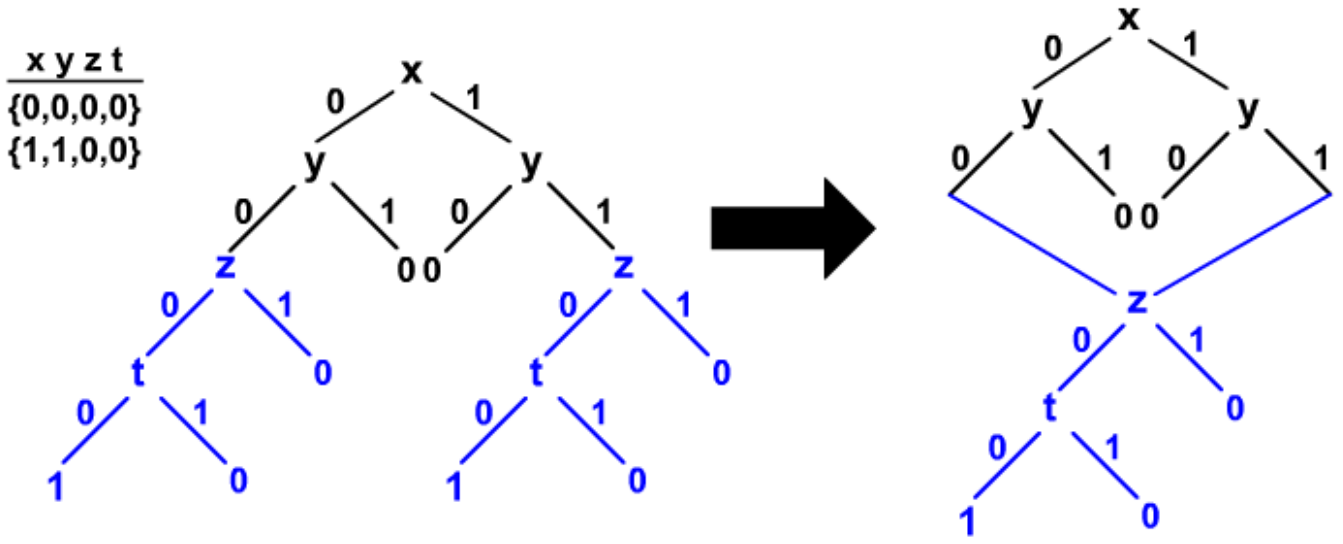
Binary Decision Diagram

représente un ensemble d'états

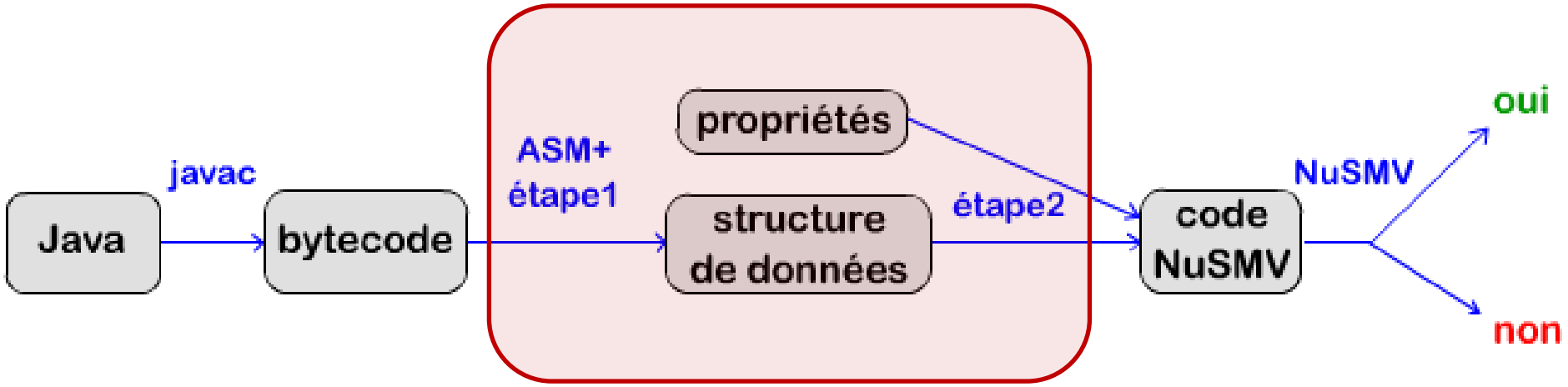
x y z
{0,0,0}
{0,0,1}
{0,1,0}
{0,1,1}
{1,0,0}



Optimisations sur les BDD



Travail réalisé



OBJECTIF : tester certaines propriétés sur un programme Java

Bytecode Java

Bytecode de la ligne :

b = a | false;

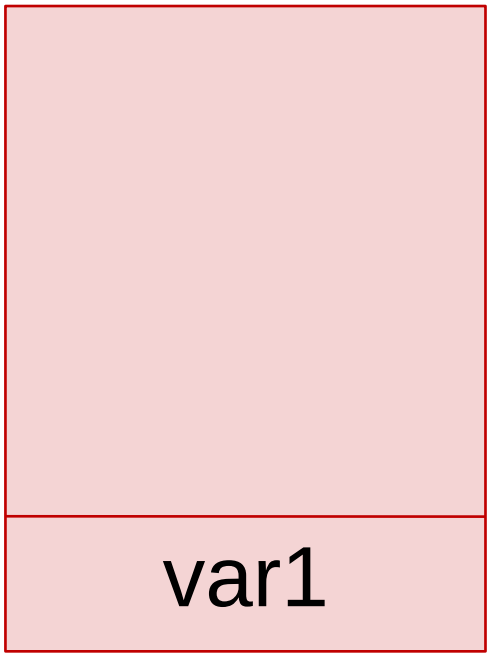
push var1
push false
or
pop var2

La pile

Bytecode de la ligne :

b = a | false;

push var1
push false
or
pop var2

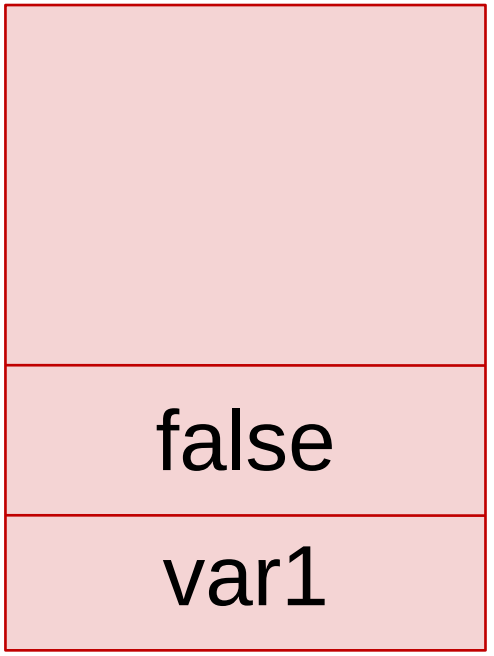


La pile

Bytecode de la ligne :

b = a | false;

push var1
push false
or
pop var2

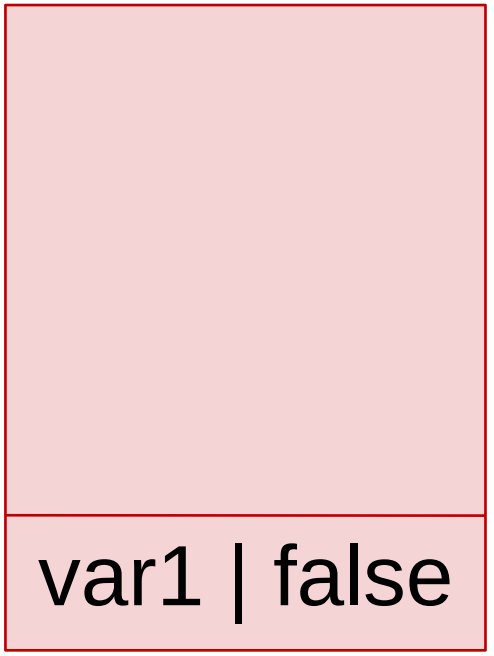


La pile

Bytecode de la ligne :

b = a | false;

push var1
push false
or
pop var2



PILE

La pile

Bytecode de la ligne :

b = a | false;

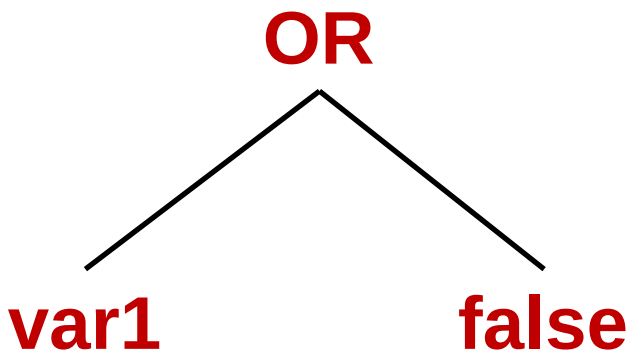
push var1
push false
or
pop var2

Var2 = var1 | false

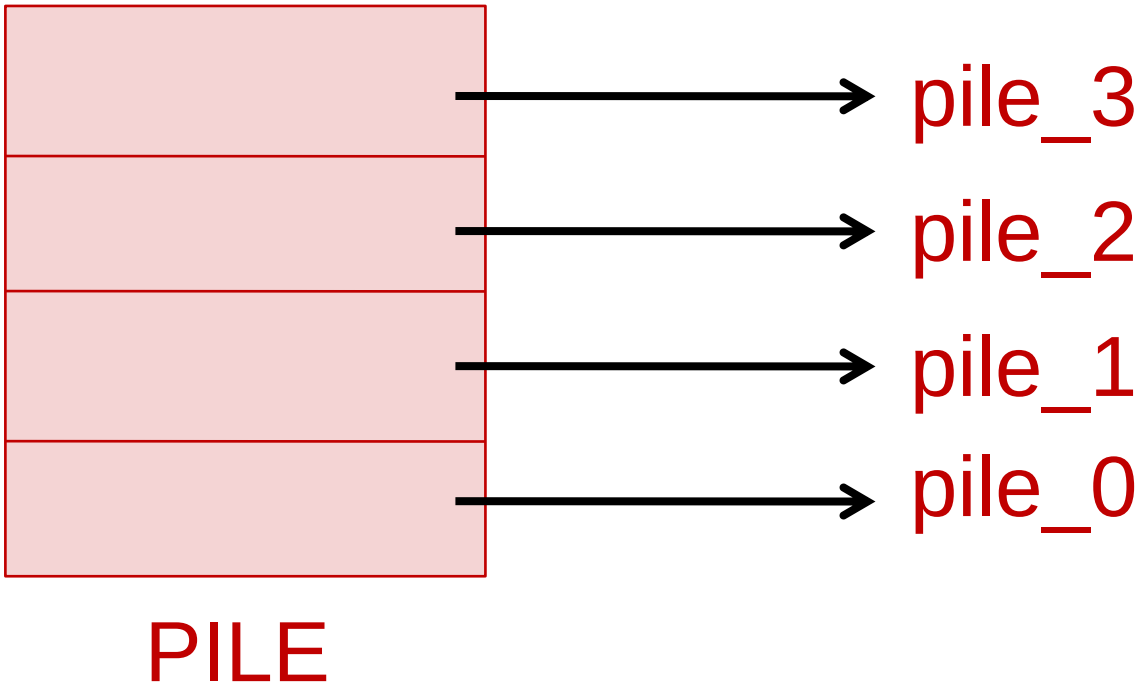
La pile

Bytecode de la ligne :

b = a | false;



Oui mais…

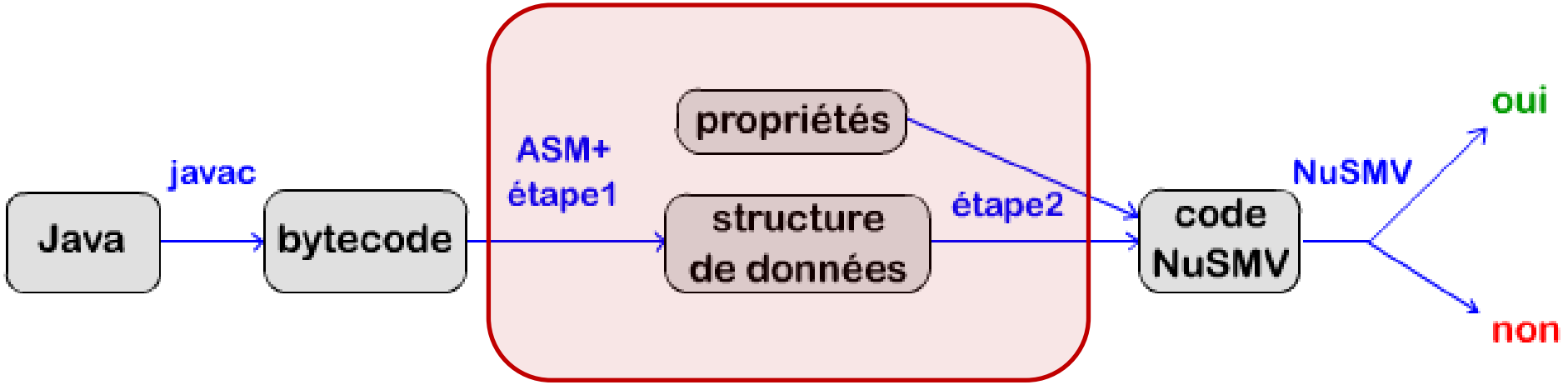


Traduction

push var1
push false
or
pop var2

```
pile_0 = var1;
pile_1 = false;
pile_0 = pile_1 | pile_0;
var2 = pile_0;
```

propriétés vérifiées



Détection de la mauvaise gestion des sections critiques

```
occ = false;

While (occ) { }          ||          While (occ) { }
occ = true;              ||          occ = true;
// section critique      ||          // section critique
occ = false;             ||          occ = false;

// suite du code
```

Détection de la mauvaise gestion des sections critiques

- un module par section critique

Logique temporelle :

- **E**(xists) : il existe une exécution
- **G**(lobaly) : pour toutes les exécutions
- **F**(uture) : dans le futur
- **A**(lways) : à tout instant

Détection de la mauvaise gestion des sections critiques

- un module par section critique

- Peu importe le chemin choisi, à tout instant aucun des modules ne doit avoir sa variable annonçant une erreur à 1

AG !(varBug_0=1 | … | varBug_n=1)

Détection d’un dead lock

- toutes les portions de code doivent se terminer

Détection d'un dead lock

- toutes les portions de code doivent se terminer

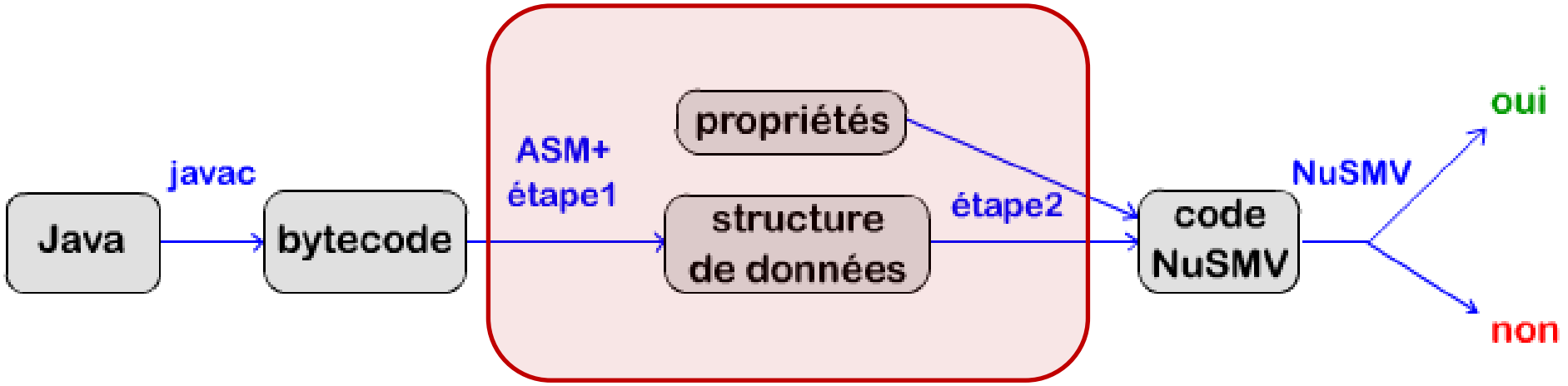
- Peu importe le chemin, il existe un moment où toutes les portions de codes se terminent

AF ("code_1 termine" & ... & "code_n termine")

Ce qu'il reste à faire

- intégrer totalement NuSMV

Ce qu'il reste à faire



Ce qu'il reste à faire

- intégrer totalement NuSMV
- gérer les entiers

Ce qu'il reste à faire

- intégrer totalement NuSMV
- gérer les entiers
- trouver l'arbre de l'expression, même dans le cas d'un **ou paresseux**

Limitations

- dans le cas général: indécidable
- le **scheduler** du processeur n’est pas pris en compte

Un programme fonctionnel

DEAD LOCK :

```
demande_de_f = true;
while(demande_de_g) {}

sc1 = true;
sc1 = false;

demande_de_f = false;
```

```
demande_de_g = true;
while(demande_de_f) {}

sc1 = true;
sc1 = false;

demande_de_g = false;
```

MAUVAISE GESTION DES SECTIONS CRITIQUES :

```
                    occ = false;
while (occ) { }                          while (occ) { }
occ = true;                              occ = true;
// section critique                      // section critique
occ = false;                             occ = false;
                    // suite du code
```

Un programme fonctionnel

PETERSON:

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

```
demande_de_f = true;
dernier = false; // dernier = f

while (demande_de_g) {
        if (dernier==false) {
                continue;
        } else {
                break;
        }
}
sc1 = true;
sc1 = false;
demande_de_f = false;
```

```
demande_de_g = true;
dernier = true; // dernier = g

while (demande_de_f) {
        if (dernier==true) {
                continue;
        } else {
                break;
        }
}
sc1 = true;
sc1 = false;
demande_de_g = false;
```

merci d'avoir écouté

TER 2009

Model-checking de programmes Java

Romain SALLES