Vérification d'un composant Java

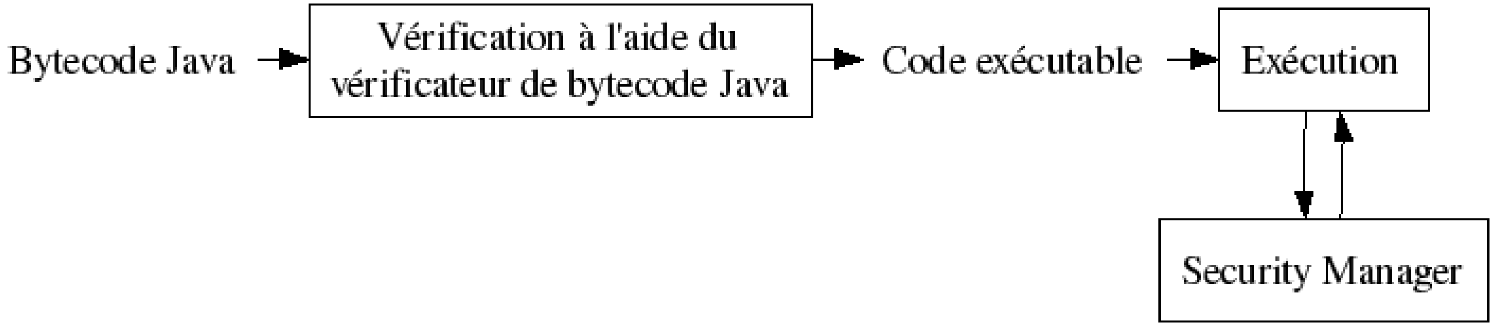
Julien Charles

Encadrants : Gilles Barthe
Benjamin Grégoire

Vendredi 24 juin 2005

Introduction

- Developper des moyens afin de vérifier un composant système Java.
- Le cas choisi : le vérificateur de bytecode Java.
  → Un des composants les plus critiques de la machine virtuelle Java.



- Dans des travaux précédents, on a vérifié plutôt un modèle.
- Pendant mon stage, j'ai vérifié directement l'implémentation.

Méthode basée sur JML

1. Annoter le composant avec JML.
2. Utiliser un outil pour générer les obligations de preuves (ESC/Java, Krakatoa, Jack).
3. Résoudre les obligations de preuve à l'aide d'un assistant à la preuve (Simplify, haRVey...).

Java Modelling Language (JML)

Un langage de spécifications de Java :

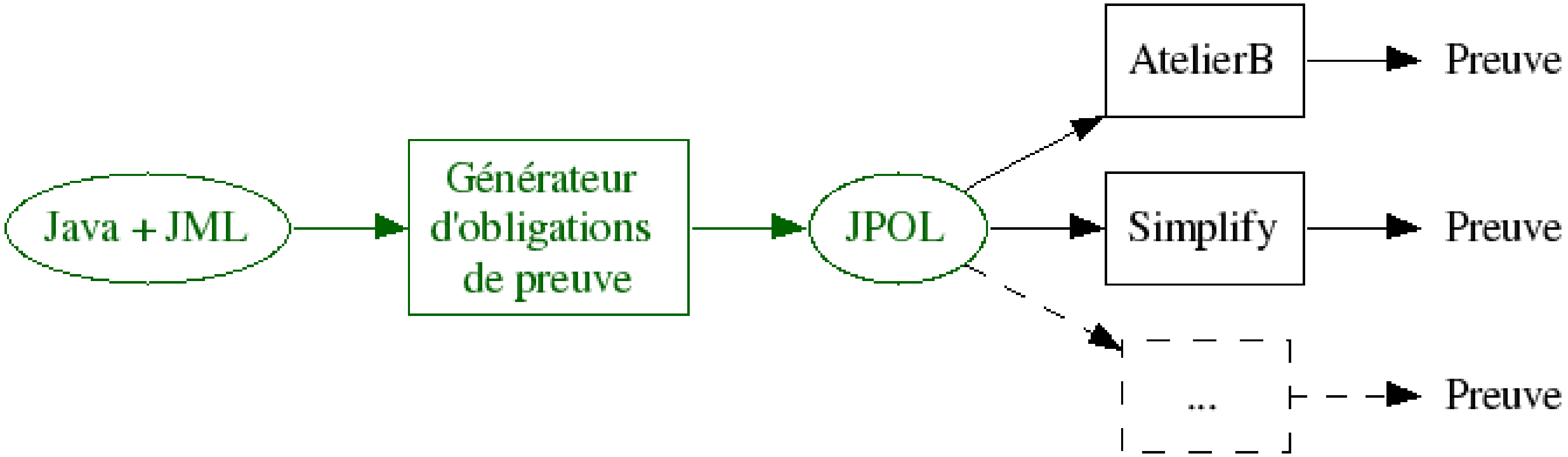
- Vérifications dynamiques (assertions)
- Vérifications statiques (logique de Hoare)

Il permet d'exprimer :

- les pré- et post- conditions
- les invariants de boucles
- les comportements (normaux ou exceptionnels)

Jack

Jack génère des obligations de preuves à partir d'un programme Java annoté en JML.



Les obligations de preuve sont générées dans le Java Proof Obligation Language - JPOL

Le vérificateur de bytecode

Basé sur l'algorithme de Kildall :
A chaque instruction i, on associe un état de typage St, une abstraction de l'état de la VM S.
En itérant sur les instructions,

- On calcule l'état de typage St_{exec} obtenu après l'exécution de l'instruction. $St_{\text{exec}} = \text{exec}(i, St)$
- On unifie St_{exec} avec les états de typage des successeurs. $St'_{\text{succ}(i)} = St_{\text{succ}(i)} \cup St_{\text{exec}}$
- Quand les états ne changent plus, l'algorithme se termine

Propriétés à démontrer

- Les débordements arithmétiques et les renvois d'exception
- Terminaison : $St_{succ(i)} \leq exec(i, St_i) \cup St_{succ(i)}$
- Correction :

$$\begin{tikzcd}
St_i \arrow[r, "\mathrm{exec}"] & St_{\mathrm{succ}(i)} = \mathrm{exec}(i, St_i) \cup St_{\mathrm{succ}(i)} \\
S_i \arrow[u] \arrow[r, "eval"'] & S_{\mathrm{succ}(i)} \arrow[u]
\end{tikzcd}$$

Architecture Choisie

4 classes principales :

- VerificationException
- State
- Instruction
- Verifier

Les classes State et Instruction sont ensuites raffinées pour correspondre à une machine virtuelle Java, les classes Verifier et VerificationException n'ont plus à être modifiées.

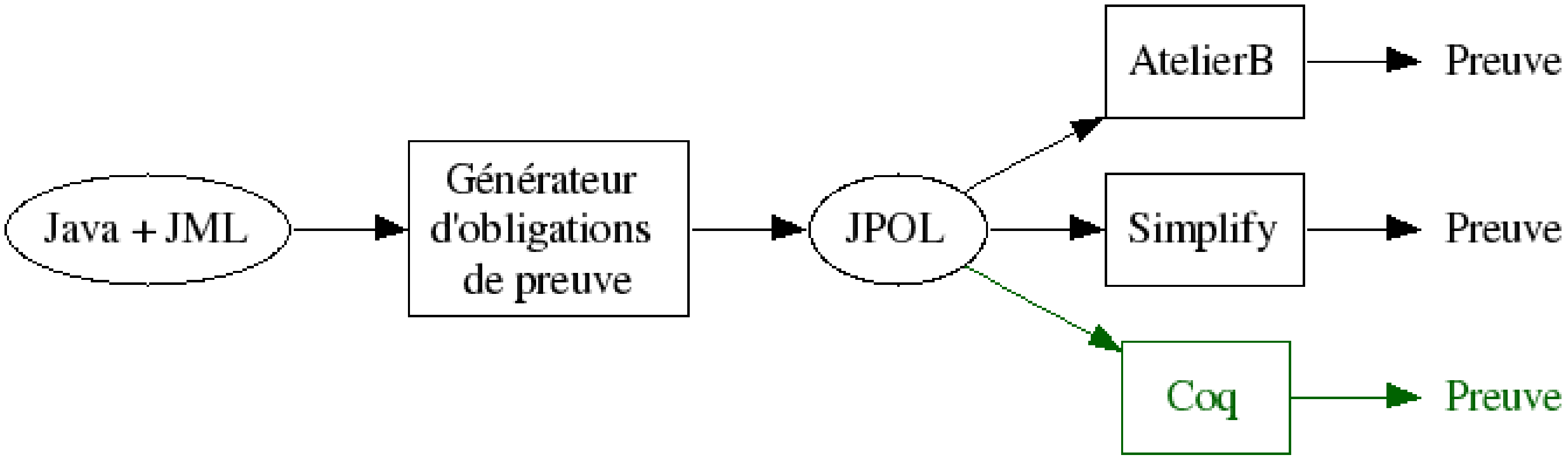
Quelques statistiques

Classes :	VerificationException	State	Instruction	Verifier
Lignes de code :	13	14	47	66
Lignes d'annotations :	5	20	54	81
Nbr. d'OPs :	60	26	129	627
Simplify :	60	26	53	5

Un plugin Coq

Pourquoi ?

- Jack n'avait pas de véritable sortie vers un prouveur interactif.
- Conserver une trace des preuves (PCC)



Coq est un langage de preuve fondé sur le calcul des constructions inductives.

Une preuve est constituée : d'hypothèses, de buts, de scripts de preuve (avec les tactiques).

Traduction JPOL vers Coq

- Traduction de JPOL vers Coq.
- Le modèle est décrit avec des structures de données décrites dans un prélude statique :
  - Arithmétique
  - Modèle mémoire

  et un prélude dynamique :
  - Classes / Relations de sous-typage
  - Définitions de champs et de propriétés sur les champs

Automatisations

Beaucoup d'obligations de preuve à résoudre souvent semblables.
$\longrightarrow$ Simplifier le travail de l'utilisateur : Ltac.

2 types de tactiques :

- celles générales :
  elimor, elimand...
- celles spécifiques à Jack :
  startJack, autoJack, solveOverriding...

Automatismes

La combinaison simple des tactiques permettait de résoudre complétement certaines obligations.
$\longrightarrow$ Trois modes 'automatiques' :

- Mode simple

  utilise principalement des tactiques qui terminent rapidement

- Mode renforcé

  utilise des tactiques plus puissantes et plus couteuses

- Mode semi automatique :

  l'utilisateur choisit la séquence de tactiques

Résultats du mode simple

Classes :	VerificationException	State	Instruction	Verifier
Nbr. d'OPs :	60	26	129	627
Simplify :	60	26	53	5
Mode simple :	45	17	93	112
Longueur script :	3	3	6	12

Conclusion

- Jack a maintenant un plugin Coq.
- J'ai réalisé la preuve du vérificateur presque entièrement.
- J'ai developpé des techniques pour automatiser les preuves en Coq.

Perspectives

- la JVM modulaire
- le multi-threading,
- les automatisations.

Un exemple

```
int i ; int[] tab = new int[3] ;
/*@ loop_modifies i, tab[*] ;
  @ loop_invariant (0 <= i) && (i <= tab.length) &&
  @                (\forall int j ; (0 <= j) && (j < i) ; tab[j] == 0) ;
  @ decreases (tab.length - i) ;
  @*/
for (i = 0 ; i < tab.length ; i++) {
       tab[i] = 0 ;
}
```

$\longrightarrow$ Jack génère 13 obligations de preuve, dont 5 concernent la boucle.

Nous allons résoudre l'obligation de preuve de la conservation de l'invariant.

Preuve de la conservation de l'invariant

intelements : REFERENCES $\rightarrow$ t_int $\rightarrow$ t_int

l_i : t_int

newObject_9 : REFERENCES

hyp1 : $0 \leq$ l_i

hyp4 : $\forall$ l_j : t_int, $0 \leq$ l_j $\wedge$ l_j $<$ l_i $\rightarrow$ intelements newObject_9 l_j $= 0$

hyp8 : interval 0 (j_sub (overridingCoupleRef t_int arraylength newObject_9 3 newObject_9) 1) l_i

hyp9 : l_i $<$ overridingCoupleRef t_int arraylength newObject_9 3 newObject_9

hyp10 : l_i $\leq$ overridingCoupleRef t_int arraylength newObject_9 3 newObject_9

———————————————————————————— (1/1)

$\forall$ l_j : t_int, $0 \leq$ l_j $\wedge$ l_j $<$ l_i $+ 1$ $\rightarrow$
overridingCoupleZ t_int (intelements newObject_9) l_i 0 l_j $= 0$

Preuve (1)

`startJack.`

1 subgoal

...

$hyp9 : l\_i <$

$overridingCoupleRef\ t\_int\ arraylength\ newObject\_9\ 3\ newObject\_9$

$H : 0 \leq l\_j$

$H0 : l\_j < l\_i + 1$

$H4 : l\_i \leq$

$overridingCoupleRef\ t\_int\ arraylength\ newObject\_9\ 3\ newObject\_9$ - 1

———————————————————————————————— *(1/1)*

$overridingCoupleZ\ t\_int\ (intelements\ newObject\_9)\ l\_i\ 0\ l\_j = 0$

Preuve (2)

`solveOverriding ; trivial`

1 subgoal

...

$hyp4 : \forall\ l\_j : t\_int,$

$0 \leq l\_j \wedge l\_j < l\_i \ \rightarrow intelements\ newObject\_9\ l\_j = 0$

$H : 0 \leq l\_j$

$H0 : l\_j < l\_i + 1$

$H2 : l\_i <> l\_j$

———————————————————————————— *(1/1)*

$intelements\ newObject\_9\ l\_j = 0$

`destrJle (l_j ≤ l_i) ; autoJack.`

Proof Completed.

Preuve (3)

En résumé, on obtient le script de preuve :

```
startJack.
solveOverriding; trivial.
destrJle (l_j <= l_i); autoJack.
```