Université de Nice Sophia Antipolis

Master 2 d'informatique
Spécialité : Programmation Modèle Langage et Technique
(PMLT)

Rapport de stage

présenté en juin 2005
par
Julien CHARLES

Vérification d'un composant Java: Le vérificateur de bytecode

Nom du reponsable de stage : Gilles Barthe
Nom du co-encadrant : Benjamin Grégoire
Nom du rapporteur : Nhan Le Thanh

Le stage à été effectué à l'INRIA, dans l'équipe de recherche Everest.

Projet Everest
INRIA Sophia-Antipolis
2004, Route des Lucioles
BP 93
06902 Sophia-Antipolis

Table des matières

1 Introduction

Aujourd'hui l'informatique ubiquitaire prend une place de plus en plus importante. Ce sont des réseaux hétérogènes composés de téléphones portables, de PDA ainsi que de petits objets portables sécurisés (POPSs) comme les cartes à puces (dans les cartes bancaires, ou dans les téléphones portables - avec les cartes SIM). Ces systèmes peut avoir à manipuler des données bancaires voire des données confidentielles. On se trouve donc dans un cadre où des logiciels doivent pouvoir interagir avec des systèmes très différents tout en demandant un niveau de sécurité maximum. Il est crucial de veiller au bon fonctionnement des programmes que l'on exécute sur ces systèmes. Pour cela il faut d'abord s'assurer que l'environnement d'exécution soit sûr, et ensuite que les applications le soient aussi.

Une partie importante de ces systèmes utilisent des applications programmées avec le langage Java. Java est un langage à typage fort, ce qui empêche les programmes de commettre un certain nombre d'erreurs lesquelles pourraient se produire avec la plupart des autres langages disponibles sur les systèmes embarqués. En Java, un programme est d'abord compilé vers du code binaire (bytecode) pour être par la suite exécuté sur n'importe quelle machine à l'aide d'une machine virtuelle (VM). Avant et pendant l'exécution du bytecode d'un programme, la machine virtuelle effectue un certain nombre de vérifications. Elle vérifie tout d'abord statiquement la correction du format du code chargé, et la vérification du bon typage du code à l'aide du vérificateur de bytecode. Ensuite lors de l'exécution du code on vérifie, grâce au Security Manager (figure 1), dynamiquement qu'il n'essaye pas d'effectuer des opérations illicites. Si le vérificateur de bytecode ou, plus généralement la plateforme d'exécution n'est pas correcte, on n'a aucune garantie sur la bonne exécution des programmes. Il est donc crucial de vérifier et de bien spécifier la machine virtuelle (ces spécifications sont décrites en détail dans [20]).

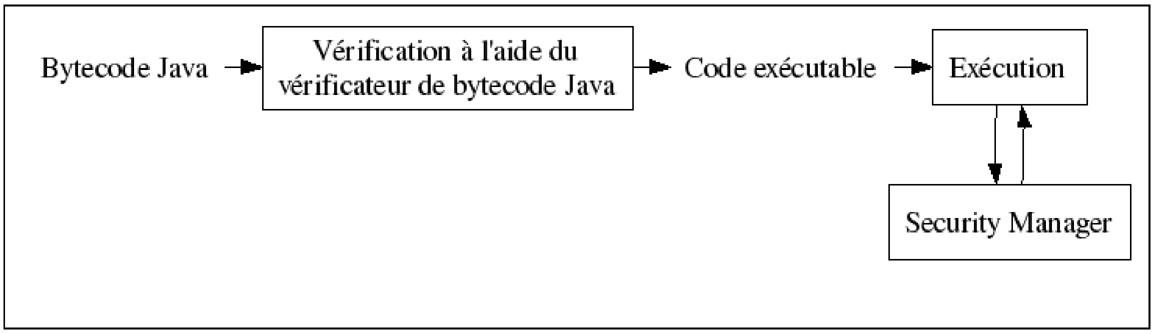


Fig. 1: Exécution d'un programme Java dans la JVM

Le but de mon stage était de développer de nouvelles technologies permettant de vérifier des composants systèmes écrits en Java. Pour ce faire nous avons pris comme exemple la vérification de l'implémentation d'un vérificateur de bytecode.

Dans des travaux précédents, le modèles du vérificateur de bytecode a été vérifié formellement. Pour ce type de vérification, le modèle du vérificateur est extrait depuis une spécification et il est ensuite prouvé à l'aide d'un assistant à la preuve : c'est ce qui a été fait dans la thèse de Guillaume Dufay avec Coq [14] et par Gerwin Klein et Tobias Nipkow avec Isabelle [16]. Pour mon stage, nous avons choisi une approche quelque peu différente : prouver directement l'implémentation et non une formalisation.

Une vérification d'une implémentation Java s'effectue en trois temps. En premier lieu il faut annoter le programme avec le langage de spécification JML, qui est un langage dédié à l'annotation des programmes Java. En second lieu les obligations de preuves doivent être générées à l'aide d'un logiciel, Jack [8]. Nous avons choisi Jack parce qu'il acceptait toutes les constructions JML et Java uilisées par mon vérificateur et de plus il est développé dans l'équipe où j'ai fait mon stage. En troisième lieu, nous devons prouver ces obligations de preuves.

Quand les preuves sont générées à partir d'une implémentation, toutes les preuves qui concernent la sémantique du langage (la vérification du non-déréférencement des pointeurs nuls ; de l'envoi ou

```
public abstract class D {
        /*@ requires tab != null && tab.length > 1;
          @ modifies tab[*];
          @ ensures tab[0] == tab[1];
          @*/
        public void a(C [] tab) {
                f(tab);
                tab[1] = tab[0];
        }


        /*@ requires t != null && t.length > 0;
          @ modifies t[*];
          @ ensures t[0] != null;
          @*/
        public abstract void f(Object [] t);

}
```

Fig. 2: Un programme annoté avec JML

du non-envoi d'exceptions) sont facilement résolubles avec un prouveur du premier ordre qui sont en général automatiques. En revanche les propriétés plus complexes (par exemple la terminaison d'une boucle) nécessitent quelque fois l'utilisation d'un prouveur d'ordre supérieur. En l'occurrence le vérificateur de bytecode Java nécessite pour la preuve de sa terminaison qu'on résolve les obligations de preuve avec un prouveur interactif.

Or Jack ne générait pas d'obligation de preuve vers des prouveurs interactifs mais vers des prouveurs automatiques (Simplify, haRVey). Pour réaliser mes preuves, j'ai commencé par écrire un traducteur des obligations de preuves de Jack vers Coq. Mais cela ne suffisait pas : Jack ayant été conçu pour être utilisé avec des prouveurs automatiques, il générait beaucoup d'obligations de preuve, ou des obligations de preuve relativement illisibles à cause notamment du nombre important d'hypothèses. C'est pour cela que j'ai ajouté des automatisations au niveau de mon traducteur vers Coq, pour simplifier voire résoudre automatiquement les obligations de preuves générées.

Mon stage s'est donc effectué autour de trois thèmes : l'écriture d'un traducteur des obligations de preuves de Jack vers Coq, l'ajout d'automatisations au niveau de Coq pour faciliter la résolution des obligations de preuve, et l'implémentation, l'annotation puis la preuve d'un vérificateur de bytecode.

2 Jack

2.1 Principe

2.1.1 Java Modelling Language (JML)

Le langage JML [17] est une surcouche de Java qui permet d'écrire des spécifications. Ces spécifications peuvent être interprétées pour effectuer de la vérification dynamique (générer des tests) ainsi que pour effectuer de la vérification statique (générer des obligations de preuves afin de prouver la véracité des annotations). Une fois vérifiée de manière statique, la propriété sera vérifiée lors de l'exécution.

Le langage JML a une syntaxe relativement étendue, parce qu'il essaye de rassembler de nombreux concepts, ainsi que pour des raisons historiques [11]. Il s'inspire fortement d'Eiffel [18]. Comme ce dernier il permet d'écrire explicitement les préconditions et les postconditions liées aux méthodes, en permettant de les utiliser soit dans le cadre de la programmation par contrats,

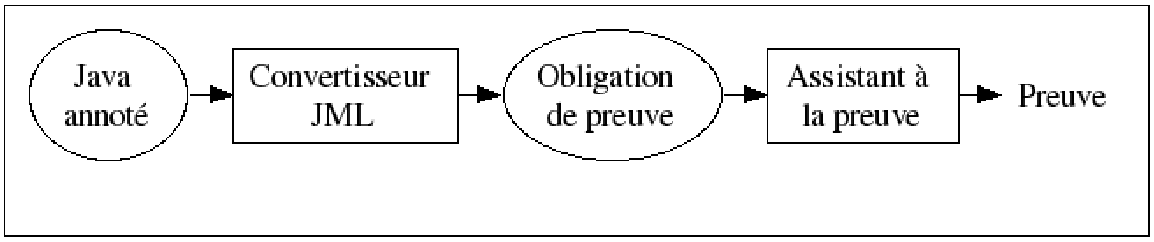


Fig. 3: Principe de fonctionnement de Jack

soit pour donner les pré- et post- conditions nécessaire au calcul de la plus faible précondition. JML permet de préciser différents comportements (`behaviours`) pour les méthodes Java. Ainsi on peut avoir des spécifications propres à un comportement 'normal' ou propres à un comportement 'exceptionnel' (c'est-à-dire quand une exception Java est lancée).

Une autre des idées importantes de JML est celle du raffinement. Ce concept est semblable à celui utilisé pour la méthode B [2]. Il consiste à écrire des spécifications incomplètes, puis à les raffiner en les complétant de manière incrémentale. Chacun des raffinement peut se trouver dans un fichier différent, mais tous se rapportent à une même classe.

Les spécifications écrites en JML se basent sur la syntaxe du langage Java (figure 2). JML permet d'utiliser des méthodes Java dans les spécifications, à condition qu'elles n'aient pas d'effet de bord au sens faible du terme [12] : elles ne doivent pas modifier les objets existants, mais elles ont le droit de renvoyer de nouveaux objets et de les initialiser. Les méthodes possédant ces caractéristiques sont dites `pure`. Ainsi un constructeur, n'ayant pas d'autres effets de bord que l'initialisation des champs de l'instance qui est créée, peut être considéré comme `pure`.

JML permet aussi de créer des variables utilisables uniquement à l'intérieur des annotations avec le mot clé `ghost`. La variable peut être ensuite modifiée avec l'instruction `set`. Ces variables sont très utiles pour modéliser les séquences et plus généralement les états avec JML. Pour chacun des états possibles, il est possible de faire correspondre un entier, et grâce à l'instruction `set` on peut le faire évoluer vers un autre état.

2.1.2 Le calcul de la plus faible précondition

Pour générer les obligations de preuve, Jack effectue un calcul de plus faible précondition [3]. Pour une postcondition donnée, la plus faible précondition est la précondition sur l'état initial qui, quel que soit cet état initial vérifiant la précondition garantit la postcondition. C'est une série de règles qui permettent de générer des obligations qui, si elles sont toutes vérifiées garantissent que le programme est prouvable en logique de Hoare.

Pour obtenir cette précondition on utilise des règles de calcul correspondant au langage de programmation sur lequel le calcul doit être effectué. Pour chacune des instructions du langage Java il existe donc une règle de calcul de plus faible précondition (celles plus spécifiques à JML se trouvent dans [15]). Une règle de calcul de plus faible précondition est une relation qui a comme paramètres une instruction i, une postcondition Q, et qui renvoie une précondition P. Une règle de calcul de plus faible précondition a donc la forme `wp(i, Q) = P`. Le calcul de plus faible précondition est en fait spécifique au langage de programmation et au langage d'annotations. Par exemple, dans le cadre de Java la règle de plus faible précondition utilisée pour l'affectation est la suivante : `wp( x = E, Q) = Q[x/E]`. Pour une postcondition donnée `Q` la précondition avant l'affectation est `Q[x/E]`.

2.2 Fonctionnement

Jack [6] est un outil qui, à partir d'un programme Java annoté avec JML, génère des obligations de preuves dans un langage intermédiaire (figure 3). Pour ce faire il effectue un calcul de plus faible précondition sur un programme Java annoté en JML. Ces obligations de preuve peuvent ensuite être résolues avec un prouveur, Simplify ou B par exemple.

Les obligations de preuve de Jack générées par le calcul de plus faible précondition ont deux types :

– Soit elles proviennent des propriétés que l'on veut vérifier grâce aux annotations JML.
– Soit elles sont directement liées à la sémantique de Java : on doit garantir qu'une méthode pure ne renvoie pas d'exception ; qu'un accès tableau est bien effectué dans les bornes de ce dernier ; le fait qu'on ne déréférence pas une référence nulle.

Dans JML il faut aussi spécifier explicitement quelle variable est modifiée ; et si Jack considère qu'une variable est modifiée alors qu'elle est marquée comme n'étant pas modifiée, il générera l'obligation de preuve pour montrer sa non modification.

Jack est fondé sur une architecture de plugin : c'est une extension de l'environnement de développement Eclipse. Il s'intègre à l'interface graphique d'Eclipse (des captures d'écran sont disponible [7]). Un des comportements intéressants de Jack est le fait qu'il découpe les obligations de preuve selon le cas d'exécution à vérifier, et il montre à quel morceau de programme correspond chacune des obligations de preuve.

Pour traduire les obligations de preuves vers les différents langages de prouveurs, Jack utilise lui-même des extensions : ainsi les traducteurs vers AtelierB, Simplify sont aussi des plugins. Quand aucun plugin de traduction n'est installé, Jack possède un traducteur vers JPOL (Java Proof Obligation Language), un pseudo langage de preuve avec une syntaxe proche de Java et JML. Le seul problème est qu'on ne peut pas utiliser JPOL pour effectuer des preuves. Jack n'avait donc pas de plugin de traduction vers un prouveur interactif. Pendant mon stage, j'ai réalisé le plugin de traduction vers le langage de preuve de l'assistant à la preuve Coq.

2.3 Le plugin Coq

2.3.1 Coq

Coq [4] est un assistant à la preuve interactif fondé sur le Calcul des Constructions Inductives. Dans Coq on peut donc exprimer des propriétés formelles, et prouver des théorèmes. Une preuve pour un théorème est construite en partant de l'énoncé à prouver, un 'but'. La preuve se présente sous la forme d'un script qui comporte des instructions qui sont appelées 'tactiques'. Une tactique est une commande qui peut être appliquée à une preuve. Le but de la plupart des tactiques est en partant d'un but de compléter la preuve afin de transformer ce but pour qu'il se rapproche des hypothèses. Ces tactiques en fait modifient directement le terme de preuve.

Coq a un certain nombre de tactiques de bases, notamment `intros` qui permet d'introduire de nouvelles hypothèses issues du but ou `apply` qui pernet d'applique une hypothèse précise à un but. Coq comporte aussi un langage, Ltac, qui permet de définir de nouvelles tactiques. Ltac donne la possibilité, entre autre, de créer des macros de tactiques. Ltac est une des fonctionnalités récentes de Coq [13]. Ltac est difficile à déboguer, mais il reste néanmoins très utile parce que c'est le seul moyen, en Coq, d'écrire facilement des tactiques.

2.3.2 Le prélude

Dans Jack, chacune des traductions vers un langage d'un prouveur est effectuée à l'aide d'un plugin. Pendant mon stage j'ai écrit, avec l'aide de Benjamin Grégoire (pour les structures de données en Coq), un traducteur d'obligations de preuves du langage interne de Jack vers Coq.

Les hypothèses et les buts se présentent sous la forme d'un arbre abstrait de syntaxe (AST) décoré avec les différents types des données. Ces types sont en fait définis pour la traduction, à l'intérieur d'un prélude qui est généré lors de la compilation de la classe par Jack. Ce prélude contient les fonctions de base liées à la sémantique Java permettant d'exprimer l'AST.

Dans le plugin Coq, nous avons représenté les types primitifs Java (`int`, `short`, `long`, `byte`, `char` - `float` et `double` n'étant pas utilisables dans Jack) directement par le type entier relatif (`Z`) de Coq. Cette approche est pratique parce qu'elle permet d'utiliser les tactiques sur l'arithmétique présentes dans Coq (notamment `omega` et `ring`). Néanmoins nous avons dû ajouter un certain nombre de lemmes pour garantir le domaine de définition de ces entiers (parfois Jack insère explicitement le domaine de définitions de ces types dans les obligations de preuve). Nous avons dû

Construction	Type	Correspondance dans Java
t_int	Set	int
t_short	Set	short
t_long	Set	long
t_byte	Set	byte
t_char	Set	char
t_bool	Set	boolean
class c_Class	Classes →Types	Class
array (class c_Class) 1	Types →Z →Types	Class []
null	REFERENCES	null
subtypes	Types →Types →Prop	-
instances	REFERENCES →Prop	-
typeof	REFERENCES →Types	-

Fig. 4: Correspondance entre des constructions Java et des constructions du plugin Coq

aussi explicitement écrire les fonctions de conversion entre ces différents types de données primitifs de Java. Ainsi il existe des fonctions de conversion des longs vers les shorts, des longs vers les entiers... Toutes ces fonctions et ces lemmes sont définis dans le fichier `littleHelper1.v`.

A part les nombres tous les autres types dans Java sont des classes ou des tableaux. Ces deux types sont en fait des références (une adresse mémoire qui correspond à l'endroit où se trouve un objet). Dans le plugin Coq on leur a donné le type `REFERENCES`.

Les objets de type `REFERENCES` ont des propriétés particulières. Une `REFERENCES` représente toujours un objet qui est initialisé à l'aide d'un constructeur. A l'intérieur de ce constructeur, la `REFERENCES` de l'objet qui est créée n'est pas une instance, mais il a une valeur différente de null. A l'extérieur de son constructeur, une `REFERENCES` est soit une instance soit égale à null. Les objets pointés par les `REFERENCES` en Java appartiennent forcément à une classe. Cette propriété est exprimée en Coq grâce à la relation `subtypes`, qui associe une référence et un type de classe ou deux types de classe.

Les tableaux et les chaînes de caractères littérales (String) sont des cas particuliers dans les références, puisqu'ils sont de type primitif, ils ont des axiomes qui les utilisent.

Jack permet aussi de modéliser les aliassages des variables. Les fonctions Coq `overridingCoupleZ` pour une référence vers un nombre (puisque tous les nombres sont de type entier relatif), et `overridingCoupleRef` pour une référence vers une autre référence (une classe), et `overridingArray` dans le cas d'un tableau, sont utilisées pour traduire l'aliassage. Ces fonctions ont la forme suivante :

```
overridingCouple f obj inst res :=
    if (= obj inst) then
          res
    else
          f (obj).
```

Ces structures de données sont écrites dans 3 fichiers : un pour définir les opérations arithmétiques, un pour définir les constructions des références et les aliassages, et enfin un troisième contenant les définitions des classes, les relations de sous-typage et les définitions des champs d'une classe. Les deux premiers fichiers sont entièrement statiques, ils ne changent pas quelle que soit la classe qui est concernée, en revanche le troisième varie selon la classe à laquelle il se rapporte. On peut ainsi discerner 2 parties : un prélude statique qui est généré pour tous les programmes, et qui contient la sémantique de base du langage Java, et un prélude dynamique qui contient des éléments qui ne peuvent être déterminés qu'au moment de la génération du prélude dans le contexte d'une classe précise.

2.3.3 Traduction des obligations

Une fois le prélude complet, la partie importante du plugin Coq était la traduction de l'AST. Une des difficultés de cette traduction est qu'en JML aucune différence n'est faite entre le type booléen est le type proposition, ce qui n'est pas le cas en Coq. De cette manière on peut écrire en JML : `result == (\forall int a ; a == 3 ==> b == 1) || b == 2` ; où `result`, `b == 2` et `(\forall int a ; a == 3 ==> b == 1)` ont tous le type booléen, alors qu'en Coq `(\forall int a ; a == 3 ==> b == 1)` correspond à une proposition. La traduction pour cette construction devient alors : `((result = true) ↔ (∀ a : t_int, a = 3 →b = 1) ∨ (b = 2)) ∧ ((result = false) ↔ ∼ ((∀ a : t_int, a = 3 →b = 1) ∨ (b = 2)))`. Il a donc fallu séparer les constructions contenant les propositions des constructions contenant des booléens.

Le même problème s'est posé en Jack pour les méthodes `pure`. Pour Jack ces méthodes sont considérées comme des macros de spécification. Si on les trouve à l'intérieur d'une spécification JML elles sont remplacées par `(\forall Result ; Precondition ==> Postcondition ==> Result)` où precondition et postcondition sont respectivement les requires et ensures de la méthode annotée en JML et `Result` correspond au symbol JML `\result`. Souvent les méthodes `pure` sont utilisées dans le cadre d'une affectation : `b == (\forall Result ; Precondition ==> Postcondition ==> Result)` avec b qui a le même type que Result. Or dans Coq cet expression est mal typée, il faudrait plutôt avoir une expression ressemblant à celle-ci : `(∀ Result, Precondition → Postcondition → b = Result)`. En fait le problème provient principalement de la manière dont Jack traduit les méthodes `pure`. Il faudrait probablement changer la traduction des méthodes `pure` de JML en Jack tel que proposé dans [10] ou plus récemment dans [12].

Jack n'avait pas assez été testé au niveau du typage de l'arbre syntaxique ; j'ai donc trouvé un certain nombre d'erreurs dans Jack liées au typage fort de Coq. Le reste de la traduction s'est effectuée avec quelques difficultés liées au fait que Jack avait d'abord été conçu pour un seul langage de preuve (B) et donc n'avait pas réellement toutes les structures de données adéquates pour une sortie vers un prouveur comme Coq.

3 Automatisation

Jack étant avant tout un générateur d'obligations de preuves vers des prouveurs automatiques, il génère beaucoup d'obligations de preuve, répétitives et souvent simples. Or avec Coq ces obligations de preuves sont souvent longues à résoudre et demandent de prouver des buts et sous-buts semblables. Ainsi on doit tester si la taille d'un tableau est bien supérieure à 0, ou alors si un déréférencement ne se fait pas sur un pointeur nul ; qu'une classe est bien le sous-type d'une autre classe, qu'un invariant est bien conservé... Ces preuves sont répétitives, et, la plupart du temps faciles à résoudre. C'est pour cela que nous avons décidé d'utiliser `Ltac` pour faciliter la résolution des obligations de preuves.

La plupart de nos tactiques soit simplifient les buts, soit résolvent des buts précis qui reviennent fréquemment. Cette approche est à opposer à celle utilisée dans la plupart des prouveurs automatiques, qui utilisent une même méthode pour prouver tous les buts.

3.1 Les tactiques du plugin Coq

3.1.1 Quelques tactiques

Nous avons écrit deux sortes de tactiques, des tactiques générales qui modifient la structure des hypothèses et des buts, et des tactiques plus spécifiques aux obligations de preuves générées par Jack.

Dans les tactiques générales nous avons la tactique `elimand` qui sert à éliminer la présence des conjonctions dans les hypothèses : si une hypothèse est de la forme `H : a ∧ b` elle sera divisée en `H1 : a, H2 : b`, et si une hypothèse de la forme `H : a → b ∧ c` elle sera remplacée par `H1 : a → b, H2 : a → c`. Cette tactique a été réellement utile, parce que Jack met en général tous les invariants d'une même classe à l'intérieur d'une même hypothèse séparée par des conjonctions.

```
Ltac tryApply :=
  match goal with
   | [ H : _ ⊢ _ ] ⇒ ((intros;apply H;autoJack) ||
                        (generalize H;clear H;tryApply))
   | _ ⇒ intros
  end.
```

Fig. 5: La tactique tryApply

Un autre cas se présente lorsqu'on écrit une conjonction dans des spécifications JML, ce qui arrive souvent. De la même manière on a aussi ajouté la tactique `elimor` servant à supprimer les disjonctions dans les hypothèses. Si on a une hypothèse de la forme `H : a ∨ b → c`, elle devient avec `elimor` : `H1 : a → c, H2 : b → c`.

La tactique `tryApply` (figure 5), quant à elle, essaye d'appliquer au but en cours chacune des hypothèses. C'est une tactique utile quand le but à résoudre contient beaucoup d'hypothèses et qu'on ne peut pas facilement choisir la bonne, ou quand la preuve doit être effectuée de manière automatique. La plupart de ces tactiques générales sont comprises dans des tactiques existantes de Coq, notamment `intuition` ou `eapply`, le seul problème étant que ces tactiques ont une complexité algorithmique trop importante dès qu'il y a de nombreuses hypothèses pour le but que l'on cherche à résoudre. Ce problème arrive fréquemment dans Jack ; alors nous avons dû développer des tactiques encore plus précises pour réduire la complexité des tactiques à utiliser.

Les tactiques plus spécifiques sont par exemple `arrtac` qui aide à résoudre certains buts sur les tableaux. `arrtac` fonctionne très bien quand on doit résoudre les buts de la forme `instances var` ou `0 <= arraylength var` avec `var` qui est un tableau. Le rôle de cette tactique est d'instancier les bons axiomes (dans un cas `ArrayTypeAx` et dans l'autre `arraylenAx`) et de générer les bons sous-buts.

Une autre des tactiques plus spécifiques est la tactique `solveOver`. Son but est de simplifier les formules qui contiennent la construction `overridingCouple`. La fonction `overridingCouple` est en fait une construction conditionnelle qu'on peut simplifier par une analyse de cas (on considère les deux cas : soit la condition est fausse, soit elle est vraie).

Ces tactiques ont été écrites pour simplifier la preuve et accélérer l'écriture des scripts de preuve. Mais la plupart du temps ce n'était pas assez : les preuves étaient facilement résolubles avec Coq, mais le prouveur Simplify n'arrivait pas à les résoudre parce que les hypothèses générées par Jack étaient de taille trop importante. C'est la raison pour laquelle nous avons décidé d'ajouter de vraies automatisations dans le plugin Coq, même si elles n'ont pas été aussi poussées que celles décrites dans [5].

3.1.2 Automatisations

Nous avons ajouté trois modes *automatiques* de preuve avec le plugin Coq de Jack. Ces modes *automatiques* ne le sont pas au sens des prouveurs du premier ordre (comme Simplify) qui utilisent une forme de résolveur du problème SAT amélioré. On applique automatiquement une tactique ou une série de tactiques à plusieurs buts ; un peu à la manière de ce qui est fait dans Krakatoa.

Les trois modes sont donc les suivants : le mode léger, qui utilise la tactique `lightAutoJack` pour prouver les buts ; le mode dur, qui utilise la tactique `toughAutoJack` et un mode semi-interactif qui demande à l'utilisateur quel tactique il veut appliquer sur les buts qu'il a sélectionné.

`lightAutoJack` appelle tout d'abord lightStartJack, une tactique semblable à `startJack`. `startJack` utilise une tactique pour simplifier l'arithmétique et nettoie aussi un peu les hypothèses. Ensuite lightAutoJack essaye j_omega (la version de omega pour Jack) et autoJack. De nombreux buts se résolvent entièrement avec ces tactiques en apparence pas très puissantes. L'intérêt de `lightAutoJack` est qu'on est sûr que sa complexité n'augmentera pas trop.

`toughAutoJack` utilise la même chose que `lightAutoJack` mais ensuite il utilise `tryApply`, `autoJack`, `elimIF` (une autre version de `solveOver`) et intuition deux fois. Les trois premières tactiques ne tendent pas vers une trop grande complexité quand il y a trop d'hypothèses. En revanche, `intuition` dès qu'il y a plus de 30 hypothèses contenant des disjonctions tend à boucler, et à demander trop de mémoire. C'est pour cela que j'ai ajouté un temps de grâce dans mon plugin, afin d'éviter ces désagréments.

Le mode semi-automatique n'est pas un vrai mode automatique. Il utilise simplement une base de données de tactiques à employer, et demande à l'utilisateur, pour la série de buts qu'il veut résoudre quelle tactique il veut utiliser. Ce mode est très utile quand Jack demande de prouver un cas absurde, où les hypothèses se contredisent et donc permettent de résoudre un certain nombre de buts trivialement.

Ces automatisations sont pratiques ; mais sauf sur les petits buts (avec la tactique `toughAutoJack`), elles ne permettent pas de résoudre un but ou des hypothèses contenant des disjonctions utiles à la résolution de la preuve. Dans ces cas-là, le seul moyen d'arriver à un résultat convenable est d'utiliser le mode semi-automatique.

3.2 Exemple de preuve

Jack génère pour une classe comme celle de la figure 2, environ une trentaine d'obligations de preuve. Si on n'a pas à montrer de propriété complexe (c'est le cas pour cette classe), toutes les obligations générées sont de la difficulté de l'obligation suivante :

```
Variable this : REFERENCES
Variable l_tab : REFERENCES
Variable hyp1 : l_tab <> null
Variable hyp2 : interval 0 (j_sub (arraylength l_tab) 1) 0
Variable hyp4 : interval 0 (j_sub (arraylength l_tab) 1) 1
Variable hyp5 : refelements_0 l_tab 0 = null ∨
        subtypes (typeof (refelements_0 l_tab 0))
                 (elemtype (typeof l_tab))
Variable hyp6 : forall (r a : REFERENCES) (b : t_int),
        r = refelements_0 a b → r = null ∨ instances r
Variable hyp7 : refelements_0 l_tab 0 <> null
Variable hyp8 : forall x234 : REFERENCES,
        ~ singleton REFERENCES l_tab x234 →
        refelements_0 x234 = refelements x234
Variable hyp9 : instances this
Variable hyp10 : subtypes (typeof this) (class c_D)
Variable hyp11 : union REFERENCES instances
                      (singleton REFERENCES null) l_tab
Variable hyp12 : l_tab <> null → subtypes (typeof l_tab)
                                   (array (class c_C) 1)
Variable hyp13 : 1 < arraylength l_tab
Lemma l :
forall x228 : REFERENCES,
~ singleton REFERENCES l_tab x228 →
forall x229 : t_int ,
instances x228 →
overridingCoupleRef (t_int → REFERENCES) refelements_0 l_tab
  (overridingCoupleZ REFERENCES (refelements_0 l_tab) 1
     (refelements_0 l_tab 0)) x228 x229 = refelements x228 x229.
Proof.
(* Write your proof here *)
Qed.
```

Cette obligation sert à garantir qu'après l'affectation `tab[1] = tab[0]`, tab[0] est bien un alias de tab[1] après l'exécution de la fonction `f`. Elle est générée par la postcondition exprimée dans JML par le `ensures`.

Pour commencer une preuve dans Jack on utilise en général la tactique `startJack`, une macro qui introduit certaines hypothèses (en utilisant la tactique `intros` de Coq) et essaye de simplifier les hypothèses en utilisant les tactiques `elimor` et `elimand`. Elle sert uniquement à rendre plus lisible et moins complexe l'obligation de preuve.

```
startJack.
```

```
1 subgoal
...
hyp15 : 1 < arraylength l_tab
x228 : REFERENCES
H : l_tab <> x228
x229 : t_int
H0 : instances x228
H4 : 1 <= arraylength l_tab - 1
H6 : 0 <= arraylength l_tab - 1
============================================(1/1)
overridingCoupleRef (t_int →REFERENCES) refelements_0 l_tab
(overridingCoupleZ REFERENCES (refelements_0 l_tab) 1
(refelements_0 l_tab 0)) x228 x229 = refelements x228 x229
```

On se retrouve avec un but à prouver qui contient la construction `overridingCouple` laquelle sert à gérer les aliassages de variables. Cette construction est en fait une instruction conditionnelle. La tactique `solveOver` est utilisée pour générer de nouveaux buts pour chacun des cas. La plupart des buts générés étant absurdes, on utilise ensuite `autoJack` pour enlever les buts trivialement résolubles.

```
solveOver; autoJack.
```

```
1 subgoal
...
hyp8 : forall x234 : REFERENCES, l_tab <> x234 → refelements_0 x234 = refelements x234
...
x228 : REFERENCES
H : l_tab <> x228
x229 : t_int
H0 : instances x228
H4 : 1 <= arraylength l_tab - 1
H6 : 0 <= arraylength l_tab - 1
H2 : l_tab <> x228
============================================(1/1)
refelements_0 x228 x229 = refelements x228 x229
```

Maintenant on a le but réel à prouver (que `refelements` après un appel de fonction est bien égal à l'ancien `refelements`), et on a même une hypothèse appropriée (hyp8). Si on instancie l'hypothèse 8 avec la variable x228 puis avec l'hypothèse H2, on obtient une égalité qui devrait nous permettre de résoudre notre but. Pour ce faire on doit utiliser la tactique `assert` de Coq.

```
assert(h1 := hyp8 x228 H2).
```

```
1 subgoal
...
hyp8 : forall x234 : REFERENCES, l_tab <> x234 → refelements_0 x234 = refelements x234
...
```

```
x228 : REFERENCES
H : l_tab <> x228
x229 : t_int
H0 : instances x228
H4 : 1 <= arraylength l_tab - 1
H6 : 0 <= arraylength l_tab - 1
H2 : l_tab <> x228
h1 : refelements_0 x228 = refelements x228
========================================(1/1)
refelements_0 x228 x229 = refelements x228 x229
```

On a presque résolu la preuve : il suffit de réécrire l'hypothèse h1 dans le but, c'est à dire de remplacer `refelements_0 x228` par `refelements x228` dans le but. Cela se fait grâce à la tactique `rewrite` de Coq.

```
rewrite h1.
```

```
1 subgoal
...
h1 : refelements_0 x228 = refelements x228
========================================(1/1)
refelements x228 x229 = refelements x228 x229
```

L'égalité restante est triviale à résoudre.

```
trivial.
```

Proof Completed.

Le script de preuve obtenu est donc le suivant :

```
startJack.
solveOver; autoJack.
assert(h1 := hyp8 x228 H2).
rewrite h1.
trivial.
```

Cette preuve était donc assez simple à résoudre à la main ; mais elle n'aurait pas réellement pu être résolue avec les automatisations du plugin Coq : aucune automatisation pour l'instant dans le plugin ne permet d'inférer les arguments de la tactique `assert`.

4 Le vérificateur de bytecode

4.1 Vérifier un vérificateur

L'objectif principal de mon stage était la preuve d'un vérificateur de bytecode. Le vérificateur de bytecode sert à vérifier le bon typage du bytecode Java. Cette propriété de typage assure une exécution sans erreur des programmes Java avec la machine virtuelle Java. Pour formaliser le vérificateur je l'ai annoté avec JML puis j'ai généré les obligations de preuve avec Jack. Peu de choses semblables ont été effectuées avec JML (un des exemples d'annotation les plus connus étant celui de Pacap, un porte monnaie électronique [9]).

4.2 Algorithme de Kildall

Le vérificateur de bytecode est exécuté sur chacune des classes Java avant leur exécution. Il effectue une analyse de flot de données sur une machine virtuelle abstraite o seul les types sont représentés. L'algorithme utilisé est celui de Kildall. Pour cet algorithme on associe à chaque instruction un état de typage, qui est une abstraction de l'état de mémoire où les valeurs sont remplacées par leur types. En itérant sur les instructions, on calcule le type de l'état après exécution de chacune des instructions, puis on unifie ce nouvel état de typage avec les états de typage des successeurs de l'instruction. Pour unifier deux états il faut :

- qu'un des deux états n'aie pas été défini, dans ce cas l'unification des deux états est celui qui est défini
- que les deux états aient la même hauteur de pile, les mêmes emplacements des variables occupés et que les types qu'ils contiennent soient unifiables. Quand deux types sont unifiés ils sont remplacés par le plus petit surtype commun. Nous avons en fait un treillis de types qui détermine si deux types peuvent être unifiés (cette relation est détaillée dans [19]).

Quand les états de typage ne sont plus modifiés par cette fusion, on a trouvé un point fixe, l'algorithme s'arrête. Le bytecode a été vérifié. Si une instruction ne peut pas s'exécuter sur un état, par exemple la pile n'a pas assez d'éléments, ou le type des variables n'est pas le bon, la vérification échoue. Deux états peuvent s'unifier uniquement s'ils ont le même nombre de variables locales, si leurs variables ont des types compatibles, et si les deux états ont la même taille de pile. Dans le cas où une fusion entre deux états ne peut pas être effectuée, le code est rejeté. Cette vérification correspond au vérificateur de bytecode monomorphe.

En général l'algorithme est implémenté autour de 5 passes :
A chaque instruction on associe un état et un marqueur 'modifiée'. On marque la première instruction comme modifiée et les suivantes non-modifiées. Toutes les instructions sont associées à un état non-défini. Un état défini est associé à la première instruction, qui correspond à l'état de la mémoire au moment de l'entrée dans la méthode. La mémoire à l'entrée de la méthode contient en général les arguments qui ont été passés à la méthode.

1. Sélectionner la première instruction notée modifiée. Si aucune instruction n'est marquée comme modifiée l'algorithme se termine. Sinon on sélectionne l'instruction et on la marque non-modifiée.
2. Calculer l'état de typage après l'exécution de l'instruction. Si le calcul échoue la vérification échoue par la même occasion.
3. Déterminer les successeurs de l'instruction en cours.
4. L'état calculé est fusionné avec les états des successeurs de l'instruction en cours. Si la fusion échoue, la vérification s'arrête.
5. Recommencer à l'étape 1.

Ce qui donne en pseudo-code Java l'algorithme suivant (le code exact de mon implémentation du vérificateur de bytecode est disponible à l'adresse [1]) :

```java
void main(Instruction [] instr) {
    boolean bHasModified;
    instr[0].setModified(true);
    do {
        bHasModified = false;
        for (int i = 0; i < instr.length; i++) {
            if(instr[i].isModified()) {
                instr[i].setModified(false);
                if(check(instr, i)) {
                    bHasModified = true; // les états ont été modifiés
                }
            }
        }
    } while(bHasModified);
```

```
}

boolean check(Instruction[] instrArray, int pos) {
    Instruction instr = instrArray[pos];
    boolean bHasModified = false;
    State s = instr.buildNewState(); // on évalue l'instruction sur son état de typage
    int[] nextInstrPos = instr.getNextPos();
    for(int i = 0; i <  nextInstrPos.length; i++){
        Instruction current = instrArray[nextInstrPos[i]];
        State newState = s.doLub(current.getState()); // on fusionne les états
        if(!newState.equals(current.getState())) {
            current.setState(newState);
            current.setModified(true);
            bHasModified = true;
        }
    }
    return bHasModified; // == true si l'état d'un successeur a été modifié
}
```

Une des propriétés importantes à vérifier sur cet algorithme est sa terminaison. Dans cet algorithme à chaque itération de la boucle, les états de typages associés aux instructions sont supérieurs ou égaux aux états de typage de l'itération précédente. Donc une notion de poids pourrait être associée à chacun des états. Quand deux états sont fusionnés il est garanti que l'état résultant a un poids plus élevé ou égal au poids de ses deux parents. Cette relation nous donne un ordre partiel sur les états de typage. Cela veut dire qu'on a un paramètre de l'algorithme qui croît à chaque itération. Le nombre des états de typages de la JVM est fini. Donc on a bien une propriété pour montrer que la boucle se termine.

4.3 Choix d'implémentation

Le programme pouvant facilement atteindre une taille importante, j'ai choisi de le diviser en plusieurs parties indépendantes, afin de rendre mon implémentation plus abstraite : j'ai implémenté d'un côté l'itération principale qui restera inchangée quelle que soit les spécifications de la machine virtuelle Java choisies, et de l'autre les instructions et les états mémoires qui dépendent du modèle choisi. L'itération principale est dans un package qui contient des classes abstraites : les instructions et les états sont ainsi implémentés de façon plus générique. Le package contenant l'implémentation est composé des instructions pour les différents types Java utilisés et des états de typages de la mémoire. Je vais maintenant détailler les différentes classes abstraites et la manière dont elles ont été implémentées quand c'est nécessaire.

4.3.1 Les erreurs de vérifications.

La première classe importante de mon vérificateur, est la classe VerificationException. C'est une classe qui représentent une exception envoyée pour signaler une erreur de vérification à l'utilisateur. Elle est relativement simple et contient uniquement un champ de message. Cette classe étant une classe utilitaire, elle ne modifie aucune des données extérieures, et uniquement son champ si besoin. Elle a donc été rapide à annoter et à prouver.

4.3.2 Les états mémoire

Les états mémoires sont représentés par la classe State, qui est une classe abstraite. Elle ne contient pas de définition précise de la mémoire : on n'a aucune information sur la pile ou sur la table des variables locales, qui correspondent à une partie de l'implémentation. En revanche, cette classe permet de déterminer si un état est défini ou non-défini. Cette notion est représentée par un champ booléen.

Au départ un état de mémoire est caractérisé comme indéfini. Il devient défini uniquement si il est fusionné avec un autre état défini. Un état défini possède un certain nombre de propriétés sur sa pile d'opérande ainsi que sur ses tables de variables. La fusion s'effectue par la méthode abstraite `doLub`. Si on fusionne deux états indéfinis, l'état résultant est aussi indéfini. Pour avoir une postcondition plus intéressante j'ai fait en sorte que cette fonction impose qu'un des deux états à fusionner soit défini, et qu'on ne fusionne jamais deux états indéfinis. De cette manière, l'état généré par `doLub` ne peut pas être indéfini. Cette spécification m'a permis de simplifier un certain nombre d'obligations de preuve de la classe Verifier.

Une fois un état défini, on a une relation d'ordre partiel sur cet état. On sait qu'il est plus grand ou égal que les deux états à partir desquels il a été construit. Et il est aussi plus grand que ou égal à, n'importe quel état indéfini. Cet relation est calculée par la fonction pure `greater_or_equal`. Elle est déclarée comme pure afin d'être utilisée dans les annotations. Ainsi `doLub` garantit dans sa postcondition que le nouvel état est `greater_or_equal` de ses deux parents.

L'implémentation de cette classe est relativement simple : c'est une classe qui contient une pile de types et un tableau des types des variables locales. Des fonctions servent à accéder simplement à ces structures de données et à générer les erreurs de vérification dans les cas de mauvaise utilisation (on essaye de retirer un élément à la pile alors qu'elle est vide, on demande un élément d'un certain type incompatible avec l'élément de tête de la pile...).

4.3.3 Les instructions

Les instructions sont aussi représentées par une classe abstraite ; la classe Instruction. Puisque dans l'algorithme de Kildall à chaque instruction est associé un état de mémoire ; la classe Instruction a un champ de type State. Une instruction peut aussi avoir un ou plusieurs successeurs. Cette relation est représentée par un champ qui est la liste des successeurs de l'instruction. Un des autres aspects est le fait qu'à chaque Instruction on a associé un champ booléen pour déterminer si elle a été modifiée ou non.

Plusieurs des vérifications des propriétés de sécurité du vérificateur de bytecode sont faites au niveau de la classe Instruction. Tout d'abord on vérifie que les successeurs de l'instruction sont bien compris dans les autres instructions du programme. Si ces succeseurs pointaient vers des instructions extérieures une erreur de vérification serait renvoyée. Cela produit donc deux comportements pour le résultat de la méthode : si aucune exception n'est envoyée tous les successeurs sont bien à l'intérieur des bornes ; si une VerificationException est générée ; il existe un successeur à l'extérieur des bornes du tableau d'instruction.

Les autres propriétés importantes sont celles liées à la fonction pure `buildNewState`. Cette fonction construit l'état de typage correspondant à l'exécution de l'instruction sur l'état courant. Cette construction peut échouer si l'instruction essaye de dépiler un élément et la pile est vide ou par exemple si un accès à une variable locale inexistante est effectué. Si en revanche elle réussit le nouvel état est forcément un état différent de nul, et cet état est défini si et seulement si l'état d'après lequel il a été calculé était défini.

Une dizaine d'instructions ont été implémentées : `load` et `store` pour les accès aux variables locales, `push` et `pop` pour obtenir ou mettre des éléments dans la pile, `op1` et `op2` qui sont deux opérateurs qui consomment les deux éléments de tête de pile et qui les remplacent par un résultat d'un certain type, `ifle` et `jump` des instructions de saut vers une autre instruction successeur, `nop` l'instruction qui ne fait rien et enfin `stop` qui est une instruction qui n'a pas de successeur. Ces instructions ont un type associé défini dans la classe OperandType, qui peut être None, Type1 ou Type2. Ce sont les instructions minimales pour avoir un simili-programme Java.

4.3.4 La boucle principale

La boucle principale de mon vérificateur est implémentée dans la classe Verifier. Elle n'est pas une classe abstraite parce qu'elle utilise les propriétés des classes abstraites State et Instruction pour spécifier un jeu d'instructions et d'états particuliers ; et l'algorithme de Kildall ne dépend pas réellement de leur implémentation exacte. Cette classe est organisée autour de deux fonctions, la fonction `verify` dans laquelle la boucle est écrite et la fonction `check` qui sert à vérifier une instruction précise.

La méthode check assure que tous les états des successeurs d'une instruction donnée, sont plus grands ou égaux que les états avant l'exécution de la méthode. Cette propriété semble simple à exprimer mais elle implique plusieurs prérequis. Tout d'abord il faut garantir que les successeurs des instructions pointent tous vers des instructions valides. Ensuite que toutes les instructions sont différentes de nul et que leurs états sont différents de nul eux aussi. Le calcul de plus faible précondition de Jack nous force à rajouter ces propriétés liées à la sémantique du langage Java. .

La méthode `verify` est la boucle principale du vérificateur de bytecode. Elle est composée de deux boucles imbriquées. La plus interne est une boucle `for` qui itère sur les instructions et vérifie toutes celles marquées modifiées (à la manière de l'algorithme de Kildall). La terminaison de la boucle interne est facile à prouver. Le `for` de cette boucle s'exécute autant de fois qu'il y a de nombres dans le tableau. La plus externe est un `while` qui fait s'arrêter l'algorithme quand plus aucun état de typage d'une instruction n'est modifié. Mais cette terminaison n'est pas évidente à prouver. Surtout avec JML qui ne permet de prouver la terminaison des boucles qu'en utilisant une variable-compteur sous la forme d'un entier.

Puisque je devais utiliser les états pour montrer la terminaison de l'algorithme, j'ai donc essayé de montrer la terminaison en faisant correspondre chaque état à un entier différent. Ainsi à chaque itération de la boucle de mon vérificateur les entiers associés aux états soit augmentent soit conservent la même valeur. Et il existe un entier supérieur à tous les entiers associés aux états : donc je peux prouver la terminaison avec l'aide de JML. Je n'ai pas encore écrit la fonction de transformation état de typage vers entier. Mais je devrais l'effectuer avant la fin de mon stage.

4.4 Preuves

Les premières preuves sont relativement faciles. La classe State se prouve presque entièrement automatiquement ; sauf au niveau du constructeur, où pour prouver la conservation des invariants il faut casser une disjonction (`instance s` $\vee$ `s = null`). En tout on a environ une soixantaine de preuves et une dizaine concernent des invariants à résoudre en utilisant la disjonction.

La classe Instruction a été elle aussi relativement facile à prouver. Un nombre important de preuves se faisaient automatiquement (environ 90%) ; ensuite la plupart des preuves pouvaient se résoudre de manière triviale, sauf quelques preuves concernant la décroissance d'une boucle : ces preuves nous ont fait intégrer une nouvelle tactique dans le plugin Coq, `destrJlt`. Cette tactique est utile quand on a une variable qui est inférieure ou égale à une autre variable. On détermine alors deux cas ; le cas où la première est égale à la seconde ; et le cas où la première est strictement inférieure à la seconde. Cette classe comportait environ 130 obligations de preuves et seules 4 demandaient d'utiliser cette tactique. Mais cette tactique s'est révélée très utile lors des preuves liées aux boucles de la classe Verifier.

Enfin la classe Verifier a été plus dure à prouver. Les preuves ont commencées à avoir beaucoup trop d'hypothèses pour arriver à les prouver automatiquement. J'ai alors été obligé de toutes les

Classes :	VerificationException	State	Instruction	Verifier
Nombre de lignes de code :	13	14	47	66
Nombre de lignes d'annotations :	5	20	54	81
Nombre d'obligations de preuve générées :	60	26	129	627
Nombre d'obligations de preuve prouvées automatiquement :	45	17	93	112
Longueur moyenne d'une preuve non-automatique :	3	3	6	12

Fig. 6: Quelques statistiques sur les preuves

effectuer interactivement. J'ai dû résoudre autour de 500 obligations de preuves. Soit ces obligations étaient triviales : un certain nombre se résolvaient selon une méthode semblable mais difficilement automatisable. Soit les obligations étaient complexes : elles demandaient une certaine méthode qui peut faire que le script de preuve est un peu grand (une moyenne de 30 étapes). Ce sont les preuves des invariants de la boucle de la méthode verify, ainsi que les preuves de son initialisation qui m'ont donné le plus de difficultés.

5 Conclusion

Pendant mon stage, j'ai écrit une sortie pour Jack vers Coq. J'ai ajouté quelques automatisations et j'ai prouvé avec cet outil une implémentation abstraite d'un vérificateur de bytecode Java. Un des aspects intéressants a été la preuve de l'implémentation d'un vérificateur de bytecode. Jusqu'à présent peu de programmes entiers ont été annotés en JML, et pour ceux qui l'ont été il s'agissait le plus souvent de vérifier des propriétés liées à la sémantique de Java plus que des propriétés liées à l'algorithme lui-même. La preuve de l'implémentation est aussi pratique parce qu'elle garantit une meilleure conformité entre les obligations de preuves et le programme à vérifier. Ce travail a permis de tester Jack sur un exemple réel. C'était la première fois que Jack était utilisé sur un exemple semblable ; et cela nous a permis de corriger un certain nombre d'erreurs qui subsistaient dans Jack.

C'est une première étape vers plus de sécurité pour les environnements d'exécution des systèmes embarqués. L'utilisation d'un vérificateur de bytecode vérifié garantit que les programmes que la machine virtuelle Java essaye d'exécuter sont bien typés. Donc il permet d'assurer une certaine fiabilité d'exécution. Cependant, il nous faudrait vérifier les autres composants systèmes comme l'ordonnanceur ou le SecurityManager, voire le Garbage Collector pour avoir une machine virtuelle vraiment sûre. Malheureusement ce n'est pas possible en l'état. Tout d'abord, pour l'ordonnanceur, il faudrait étendre la syntaxe et la sémantique JML pour pouvoir modéliser le parallélisme. Ensuite pour la plupart de ces composants systèmes les modifications principales seraient dans Jack : il serait utile de le modifier pour qu'il tienne compte des politiques de sécurité Java modifiables ainsi que des appels au SecurityManager pendant les exécutions de programmes. Ces améliorations permettraient de spécifier les programmes d'une manière plus précise.

Un autre des aspects qu'il serait intéressant de compléter est l'automatisation des preuves pour Coq. Plus les spécifications sont précises, plus il y a de preuves qui impliquent des résolutions semblables. Ce serait un gain vraiment intéressant de pouvoir résoudre encore plus de preuves automatiquement. Une des extensions qui pourraient améliorer cet aspect est l'écriture d'une tactique comme `auto` de Coq. La tactique `auto` permet de préciser explicitement quelle tactique utiliser quand un but a une certaine forme. Il serait intéressant de l'étendre pour qu'on applique une tactique aussi quand les hypothèses présentent une certaine forme. Si cette tactique est efficace elle permettrait de se concentrer davantage sur les propriétés importantes des programmes.

Références

[1] Les classes du vérificateur de bytecode annoté. Disponibles à l'adresse `http://www-sop.inria.fr/everest/personnel/Julien.Charles/bcv`.

[2] Site web de l'atelier B. `http://www.atelierb.societe.com`.

[3] G. Barthe, T. Rezk et A. Saabas. Proof obligations preserving compilation. A paraître dans FAST'05, disponible à l'adresse `http://www-sop.inria.fr/everest/personnel/Tamara.Rezk/publications.php`.

[4] Y. Bertot et P. Castéran. *Interactive Theorem Proving and Program Development ; Coq'Art : The Calculus of Inductive Constructions*. Series : Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[5] E. Broch Johnsen et C. Lüth. Theorem reuse by proof term transformation. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'04)*, volume 3223 of *Lecture Notes in Computer Science*, pages 152–167. Springer-Verlag, September 2004.

[6] L. Burdy, A. Requet et J.-L. Lanet. Java applet correctness : A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003 : Formal Methods : International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.

[7] L. Burdy et al. Le site web de Jack, 2004-2005. `http://www-sop.inria.fr/everest/soft/Jack`.

[8] L. Burdy, Y. Cheon , D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Rustan, M. Leino et Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 2005.

[9] N. Cataño et M. Huisman. Formal specification of Gemplus' electronic purse case study. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME : Formal Methods Europe*, volume 2391 of *Lecture Notes in Computer Science*, pages 272–289, Copenhagen, Denmark, July 22-24 2002. Springer.

[10] D. Cok. Reasoning with specifications containing method calls in jml and first-order provers. 6th Workshop on Formal Techniques for Java-like Programs (FTfJP) at ECOOP, 2004.

[11] D. Cok et J. Kiniry. Esc/java2 : Uniting esc/java and jml. In *CASSIS*, pages 108–128, 2004.

[12] Á. Darvas et P. Müller. Reasoning About Method Calls in JML Specifications. Soumis à FTfJP 2005. Disponible à l'adresse `http://sct.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=DarvasMueller.pdf`, 2005.

[13] D. Delahaye. A tactic language for the system Coq. In *LPAR*, pages 85–95, 2000.

[14] G. Dufay. *Vérification formelle de la plate-forme Java Card*. PhD thesis, Université de Nice Sophia-Antipolis, December 2003.

[15] B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58 :61–88, 2004.

[16] G. Klein et T. Nipkow. Verified bytecode verifiers. *Theor. Comput. Sci.*, 298(3) :583–626, 2003.

[17] G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok et J. Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Disponible à l'adresse `http://www.jmlspecs.org`, April 2003.

[18] G. Leavens et Y. Cheon. Design by contract with JML. L'article est disponible à l'adresse : `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`.

[19] X. Leroy. Java bytecode verification : an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification, CAV 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.

[20] T. Lindholm et F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999. ISBN 0-201-43294-3.

Annexe

A Jack's Coq Plugin's User Documentation

A.1 Goal

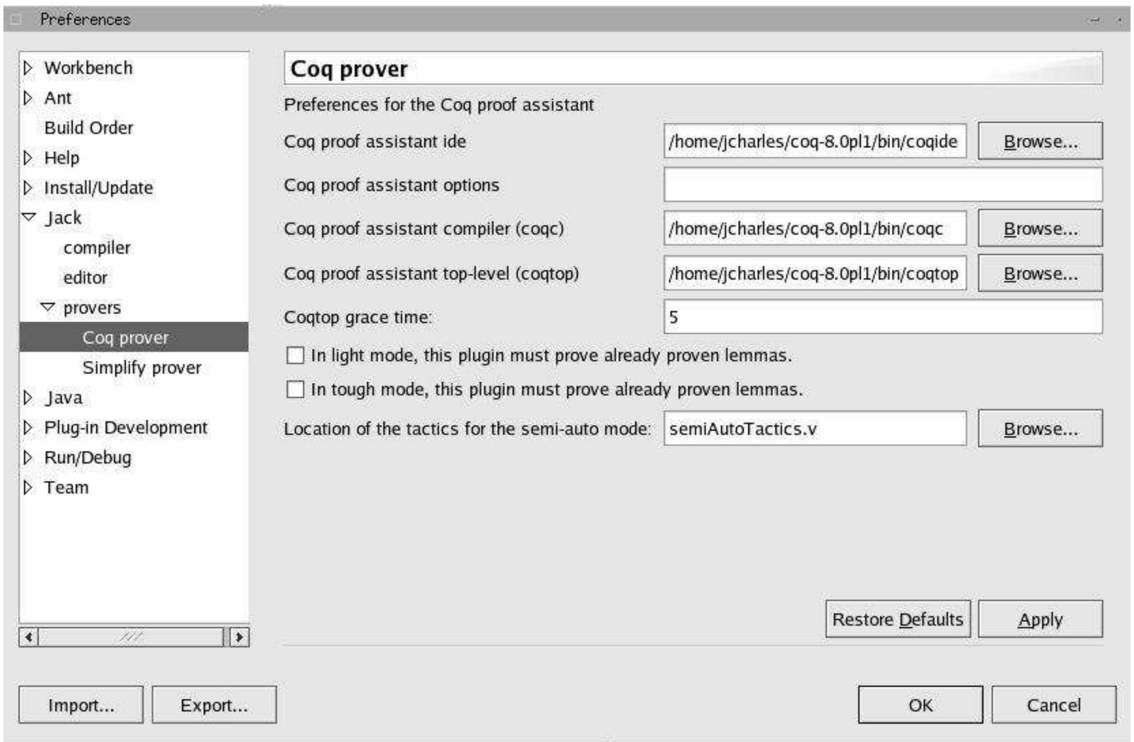
The aim of this documentation is to guide the user through the early steps in using Jack's Coq plugin. It is really small and simple and it is gui-oriented.

A.2 Installation and requirements

A.2.1 Requirements :

- First you need a working Jack installation.
- Usually the Coq plugin is installed together with Jack; but if it isn't the case you should be able to obtain it through its update site : `http://www-sop.inria.fr/everest/soft/Jack/UpdateSite` or through this site : `http://www-sop.inria.fr/everest/soft/Jack/download`.
- A working Coq installation (with a proper ide and with `coqtop`).

A.2.2 Installation

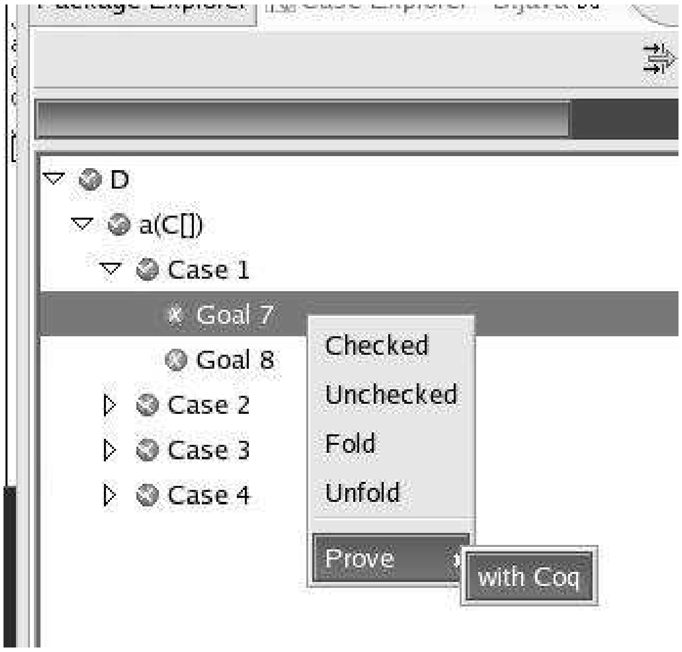


In eclipse go to Help > Find And Install; select the option 'Search for new feature to install'; click on Next; then click on 'New Remote Site'... and then that's done.

After that you should restart eclipse. When it is done, go to Window > Preferences; in the Jack menu in the compiler subsection tick the 'Generate Coq output file' check box, and in provers > Coq prover fill out the preferences.

A.3 The Proof Modes

A.3.1 The Interactive Mode

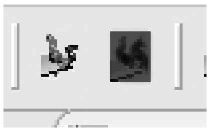


This is the usual way to use Coq. To access the proof obligation, in Jack's Case Viewer click with the right button on the goal you want to prove ; and there you are, the plugin launches the Coq Ide you have chosen.

After you have edited the proof obligation it is checked with coqc and saved inside the Jack's Jpo. If the checking succeeds ; the goal is considered solved.

An interactive proof always begins with 'Proof with autoJack.' followed by the tactic startJack.

A.3.2 The Automatic Modes

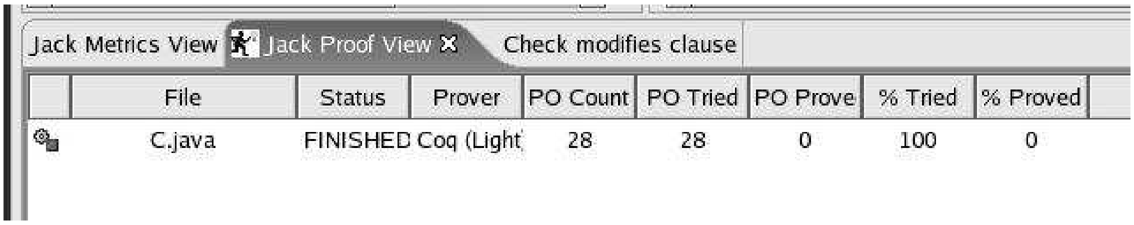


Strangely in this plugin you can try to prove several goals automatically using some tactics written in Ltac, namely lightAutoJack and toughAutoJack.

It uses coqtop to check the different goals. To prove automatically there is two buttons with a coq.

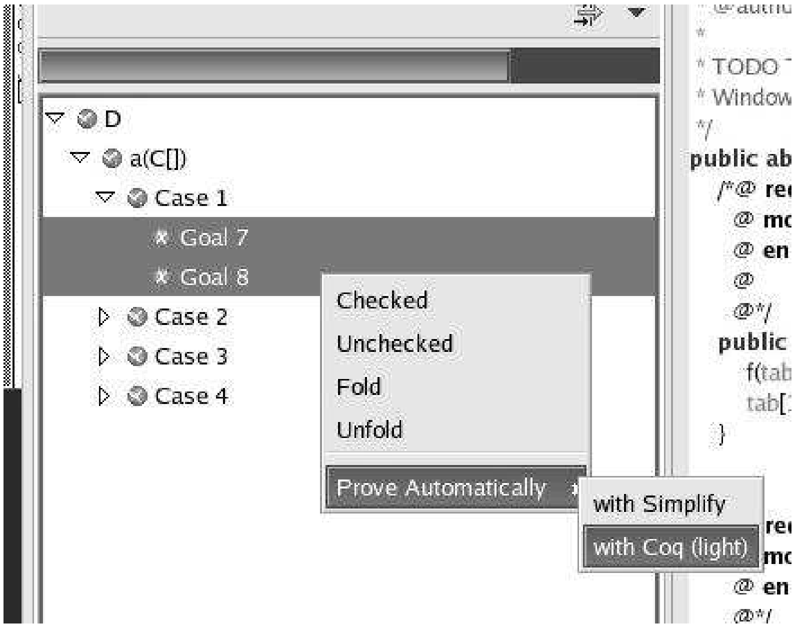
The Light Mode : To prove automatically with the light mode (this means using the tactic lightAutoJack) you must first select a file and the press the button which has a yellow coq over it.

The Tough Mode : To prove automatically with the tough mode (this means using the tactic toughAutoJack) you must first select a file and the press the button which has a red coq over it.



If some proofs take too much time (indeed more seconds than the number allowed by the grace time) ; coqtop is restarted.

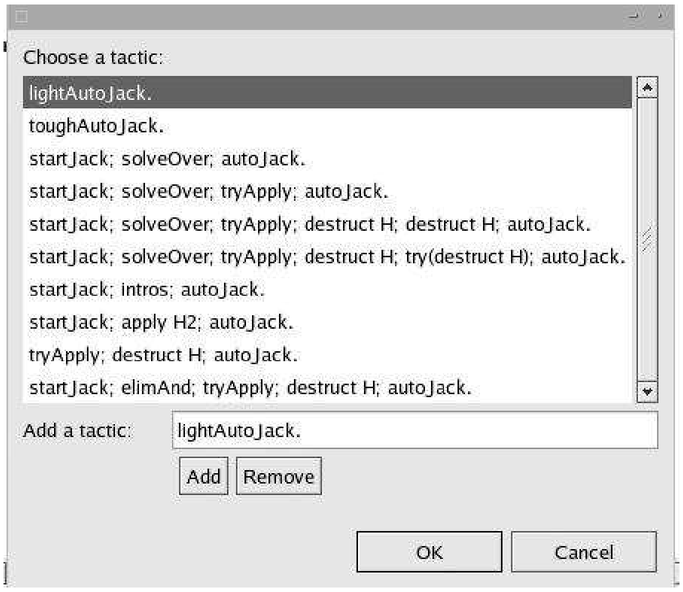
A.3.3 The Semi-Automatic Mode



When you select several goals in the case viewer, and you click with the right button you have a pop up dialog that opens with the option 'Prove automatically with > Coq (Light Mode)', in fact it is not done through the light mode but through the Semi-Auto Mode.

A dialog box will appear asking you to choose a tactic to apply to the goals you have chosen to solve automatically. You can add your own fave tactics with the buttons add and remove.

Then press the Ok, or Cancel button (cancel solves using the lightAutoTactic). It will try to solve the goals with the chosen tactic.



The tactic list is stored in the file chosen in Window > Preference > Jack > provers > Coq Prover > Location of the tactics for the semi auto mode ; one tactic per line.