UNIVERSITÉ DU QUÉBEC À MONTRÉAL

VÉRIFICATION DE CODE-OCTET AVEC SOUS-ROUTINES PAR CODE-CERTIFIÉ

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

CONCENTRATION EN INFORMATIQUE SYSTÈME

PAR

MATHIEU CORBEIL

JUILLET 2007

Remerciements

La rédaction de ce mémoire a été rendue possible grâce à la collaboration de plusieurs personnes. Je remercie particulièrement Étienne M. Gagnon, professeur à l'Université du Québec à Montréal et qui fût mon directeur de recherche, pour son assistance et ses conseils judicieux lors de chaque étape de préparation de ce mémoire. Merci aussi pour son soutien financier, pour sa disponibilité et pour sa patience!

Des remerciements vont aussi à Jamal Lazaar, étudiant à la maîtrise à l'UQAM en 2005-06, pour sa participation au développement des bases du vérificateur en une phase et pour son travail sur la modélisation mathématique des équations de flot de données. Merci à Patrick Latifi, en stage de recherche de premier cycle CRSNG à l'été 2005, pour son apport au développement de l'abstraction logicielle du format de fichier *class* et au module de lecture/écriture de fichiers pour le vérificateur. Ils ont été de proches collaborateurs.

Je remercie également les membres du laboratoire LATECE de l'UQAM pour leur assistance technique, particulièrement Grzegorz Prokopski qui était étudiant au doctorat et administrateur système en 2005-2006, et les membres du groupe de recherche Sable de l'université McGill pour les échanges fructueux lors de mon court séjour parmi vous.

Enfin, j'adresse mes remerciements spéciaux à ma famille : mes filles Hanna et Miryam Corbeil pour leur amour et leur joie de vivre, elles sont une inspiration; mes parents Johanne et Denis Corbeil pour leurs encouragements et leur support.

Et le plus grand merci à mon épouse Júlia Hàrnàsi pour son amour, sa patience, sa compréhension et encore plus!

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

RÉSUMÉ

Des applications compilées en code-octet et encodées dans le format de fichier *class* sont normalement chargées et exécutées par une machine virtuelle. La vérification du code-octet est un processus utilisé par une machine virtuelle pour renforcer la sécurité des systèmes distribués en garantissant la conformité du code de classes avec les règles du langage.

Les algorithmes actuels de vérification du code-octet utilisent des approches comme l'analyse de flot de données, le *model checking* ou le code-certifié (similaire au *proof-carrying code*). Mais l'on dénote certains problèmes, en particulier dus aux sous-routines, avec soit des spécifications informelles et incomplètes, soit des algorithmes ne supportant qu'un sous-ensemble du jeu d'instructions du code-octet, ou encore des performances exponentielles proportionnellement à la taille du code des méthodes à vérifier.

Dans ce mémoire nous présentons une technique de vérification du code-octet avec sous-routines par code-certifié. Nous présentons en particulier la conception d'un format de certificat, d'un algorithme de calcul de certificat et d'un algorithme de vérification pour l'ensemble du jeu d'instructions du code-octet. Notre algorithme de vérification a une complexité linéaire en proportion de la taille du code des méthodes.

Le développement d'un vérificateur et d'un compilateur de certificats a servi à conduire des expérimentations qui montrent que le format de certificat proposé est suffisamment riche pour effectuer la vérification du code-octet. De plus, nous présentons les résultats de l'évaluation du coût en espace mémoire de la certification de classes avec une telle technique. L'observation montre un accroissement relativement faible de la taille d'un échantillon de plus de 35 000 classes suite à leur certification.

Mots clés : Vérification; Code-octet; Code-certifié; Analyse de flot de données; Sous-routines; Java.

ABSTRACT

Computer programs compiled to bytecode and encoded in the *class* file format are normally loaded and executed by a virtual machine. Bytecode verification is a process by which virtual machines strengthen security of distributed systems and that aims at checking whether the code of classes is in line with the rules of the language.

Current bytecode verification algorithms use such techniques as dataflow analysis, model checking or certificates (similar to proof-carrying code). But some problems still arise, caused particularly by subroutines, either with some specifications being informal or incomplete, either algorithms which only support a subset of the bytecode instruction set or else with performances that are exponential to the code size of verified methods.

In this thesis, we present a technique for bytecode verification with subroutines, using certificates. We introduce a certificate format, an algorithm to generate such certificates and a verification algorithm for the whole bytecode instruction set. Our algorithm has a linear complexity in proportion of code size.

The development of a bytecode verifier and a certificate compiler allowed us to conduct experimentations which indicate that the proposed certificate format is sufficiently rich to verify bytecode programs. Moreover, we present experimental results of the effects of certification on class file size. Observations show a relatively small increase in size for a sample of over 35 000 certified classes.

CHAPITRE I

INTRODUCTION ET CONTRIBUTIONS

1.1 Informatique distribuée, sécurité et code-octet

Depuis une quinzaine d'années, l'usage des technologies de l'information et des télécommunications se démocratise rapidement. Nous observons ainsi une croissance dans le nombre d'usagers et d'intervenants, dans le nombre d'applications logicielles utilisées et dans la quantité de protocoles et de normes en concurrence. L'informatique distribuée pose un nouveau défi à la sécurité de l'information. En effet, les systèmes d'exécution sont vulnérables face à la difficulté d'authentifier la source de données ou pour identifier d'éventuelles erreurs de transmission des données. De nouveaux moyens ont dû être pris afin de garantir la disponibilité, l'intégrité et la confidentialité des données d'un système informatique.

La plate-forme Java [JdkX] est une de ces technologies qui a su s'imposer tant dans les milieux scientifiques que dans l'industrie logicielle. Grâce à son écosystème riche, composé d'un langage de programmation orienté-objet de haut niveau relativement simple, de bibliothèques de fonctionnalités (réseaux, bases de données, encryption, ...), d'environnements de développement et d'exécution. Java est également très portable : le code source est généralement compilé en code-octet, langage intermédiaire distribuable sur plusieurs architectures matérielles (des téléphones mobiles aux ordinateurs centraux) et

logicielles. Le code-octet est alors interprété ou recompilé en code natif par la machine virtuelle. La spécification de la machine virtuelle Java [LinY99] définit les règles de ce langage intermédiaire et nous en présentons les principes fondamentaux au chapitre 2.

1.2 Sécurité, code-octet et vérification

Le langage Java permet de renforcer la sécurité de haut niveau des applications par plusieurs moyens : contrôle d'accès avec la conception orientée-objet (*public, private*), bibliothèques de fonctionnalités de cryptographie, de gestion des usagers et des droits d'accès et autres (*Security Manager, Java Authentication and Authorization Service (JAAS)*).

La sécurité de bas niveau a aussi fait l'objet d'une conception élaborée. Le langage ne permet pas l'arithmétique de pointeur, les pointeurs à *null* sont surveillés par la machine virtuelle, les programmes évoluent dans un espace mémoire restreint et doivent se soumettre à un contrôle d'accès aux ressources (système de fichier, ports réseau, ...) et des contraintes de chargement s'assurent que si, lors de l'exécution d'un programme, plusieurs classes portent le même nom, elles sont effectivement les mêmes.

Parce qu'on a souvent reproché à Java sa lenteur, plusieurs techniques d'optimisation de compilation et d'exécution ont été développées. Une caractéristique, qui concerne particulièrement notre travail, est que la machine virtuelle est « agressive » (par opposition à défensive), c'est-à-dire qu'elle exécute le code de méthodes sans se demander à chaque occasion si ce code est syntaxiquement et structurellement correct. Le code-octet étant en effet un langage typé, les instructions sont spécialisées pour manipuler des types de données précis. Ainsi, une simple altération dans la séquence d'instructions et la machine virtuelle tombe en panne, mettant en péril la sécurité de l'information et la disponibilité de l'application.

D'où l'importance de la vérification du code-octet. Ce processus est effectué par la machine virtuelle lors du chargement de classe (qui n'a lieu qu'une fois par classe, préalablement à son exécution) et permet de s'assurer que le code est « sûr », c'est-à-dire qu'il est bien structuré. La spécification de la machine virtuelle Java [LinY99] suggère, de façon informelle, des indications pour effectuer la vérification de code-octet produit par un compilateur Java standard. Cependant, le code-octet est un langage en soi et peut servir de langage intermédiaire à d'autres langages de haut niveau ou peut même être directement programmé. La condition du compilateur standard est donc restreignante. Une littérature scientifique abondante a aussi été publiée présentant des formalisations parfois compliquées ou ne considérant seulement qu'un sous-ensemble du code-octet (voir chapitre 6).

Il en ressort que la vérification du code-octet n'est pas aussi simple qu'il n'y paraît et qu'il n'y a pas encore de consensus autour d'un algorithme complet et économique. L'élément majeur de complexité est la vérification des instructions `jsr` et `ret`, généralement associées aux sous-routines, qui ne s'intègrent pas bien à l'analyse de flot de données utilisée par la plupart des algorithmes. D'ailleurs certains ont tout simplement éliminé ces instructions en rendant le code de sous-routines *inline*, comme [RosR98], [Rose02] ou même les dernières versions du JDK 1.5 et 1.6 de Sun [JdkX], qui dans leurs travaux se préoccupent d'optimiser l'algorithme de vérification avec le code-certifié. Les sous-routines sont pourtant une richesse du langage qui offre un mécanisme de réutilisation du code à l'intérieur d'une même méthode. De plus, rendre le code des sous-routines *inline* peut causer la croissance exponentielle de la taille du code d'une méthode [Lero01].

Ce problème a été investigué en détail dans [SaVM06]. Ce travail de recherche, qui sert de base à la technique de vérification par code-certifié présentée dans ce mémoire, a été réalisé à l'UQAM conjointement avec Jamal Lazaar et Étienne M. Gagnon. Nous y présentons, principalement, les bases mathématiques d'un algorithme d'analyse de flot de

données polyvariant pour l'ensemble du code-octet et permettant un usage très permissif des sous-routines.

1.3 Objectifs du mémoire

Dans ce contexte, nous avons voulu présenter une technique de vérification du code-octet qui soit complète et qui tire avantage des travaux déjà réalisés dans le domaine. Aussi, un objectif de ce mémoire était d'analyser les techniques de vérification existantes afin d'en dégager une synthèse qui puisse relever les défis techniques posés par la vérification du code-octet. Nommément : l'analyse de flot de données, les sous-routines, la vérification de l'héritage multiple des interfaces, des contraintes de sous-type et de nombreux autres éléments du langage.

1.4 Contributions

L'originalité de notre travail est que nous faisons la première présentation d'un formalisme de vérification utilisant des certificats supportant les sous-routines, de l'implémentation des algorithmes et de l'évaluation de leurs performances. Il est important de distinguer cette technique de celle de certificat électronique utilisée en sécurité informatique pour l'authentification d'entités (données, machines, personnes, ...). Cette dernière fait en général appel à la cryptographie et ne permet pas de garantir que les données (en l'occurrence, le code-octet des fichiers *class*) sont exemptes d'anomalies.

Les contributions du mémoire sont donc :

- L'introduction d'un certificat permettant la vérification, en deux phases, de code-octet incluant les sous-routines. Le but principal est de diminuer la complexité asymptotique

du processus de vérification « standard » [LinY99] requérant une analyse de flot de données (processus en une phase). Notre technique est inspirée du travail présenté par Rose [RosR98] [Rose02], mais va plus loin en proposant un format de certificat différent qui permet à notre algorithme de supporter l'ensemble du jeu d'instructions tel que présenté dans la spécification de la machine virtuelle Java [LinY99] (incluant donc les sous-routines) et en permettant la vérification des contraintes de sous-type, notamment pour les interfaces, deux choses qui n'étaient pas présentes dans les travaux de Rose.

La présentation de cette technique comprend, plus précisément :

i. la conception du format de certificat et la démonstration de sa validité. Un certificat comporte suffisamment d'informations sur les types manipulés par le code d'une méthode pour en effectuer la vérification;

ii. la conception d'un algorithme de calcul de certificat permettant de certifier des classes existantes (déjà compilées) sans avoir à en modifier le code; la compatibilité et la portabilité du code est donc préservée. Un certificat est ajouté à une classe sous la forme d'attributs de méthode et de classe;

iii. la conception d'un algorithme complet de vérification de classe utilisant ce format de certificat.

- L'implémentation d'un compilateur certifiant pour des classes existantes (déjà compilées).

- L'implémentation d'un vérificateur de code-octet pouvant utiliser un algorithme d'analyse de flot de données (processus en une phase) ou celui de code-certifié (deux phases). Ce vérificateur est une application Java qui est intégrée dans la machine virtuelle SableVM [Gagn02] et qui peut aussi être utilisée de manière autonome avec

interface en ligne de commande. Le vérificateur choisit le mode de vérification automatiquement en détectant la présence de certificat dans une classe. Il permet aussi, de façon optionnelle, de vérifier statiquement les contraintes de sous-types (telles que spécifiées par Sun [LinY99] à la section 2.6.7) ou alors de produire l'ensemble de ces contraintes qui devraient être satisfaites afin que la classe soit vérifiable.

- L'évaluation par expérimentation de l'impact relatif sur la taille des classes de l'utilisation de tels certificats. Ces expérimentations ont permis de raffiner la technique de certification afin d'atteindre un ratio de taille acceptable pour des applications du « monde réel ».

- La présentation des principes directeurs de l'algorithme de vérification du code-octet (en une phase) basé sur une analyse de flot de données ainsi que de ses fondements mathématiques tels que présentés dans [SaVM06].

1.5 Structure du mémoire

Ce mémoire présentera d'abord certaines notions préliminaires relatives au code-octet, au format de fichier *class* et à la vérification dans le chapitre 2. Le chapitre 3 présentera la formalisation de l'analyse de flot de données et de l'algorithme de vérification en un temps. Au chapitre 4, la vérification en deux temps sera présentée. Seront introduits, plus particulièrement : un format de certificat, un algorithme de calcul de certificat expliquant comment l'intégrer au format de fichier *class*, un algorithme de vérification en deux temps utilisant le certificat et pour terminer une justification de l'utilisation de cette technique et de sa validité. Le chapitre 5 décrira sommairement l'implémentation du vérificateur et le cadre d'expérimentation, le choix des données utilisées et les résultats de mesures portant sur l'impact des certificats sur la taille de fichiers *class* et une discussion concernant les

observations. Les principaux travaux reliés seront décrits, commentés et comparés au présent ouvrage dans le chapitre 6. La conclusion au chapitre 7 permet de souligner les éléments et les résultats les plus importants de nos travaux et d'indiquer des pistes pour des travaux qui pourraient être menés dans le domaine de la vérification du code-octet par code-certifié.

CHAPITRE II

NOTIONS PRÉLIMINAIRES

Avant d'aborder le thème central de ce mémoire, certaines notions sont nécessaires en guise de mise en contexte. Ainsi, qui pense vérification du code-octet pense aussi format de fichier *class*, compilation, communication réseau, machine virtuelle et chargement de classe. Ce chapitre présente donc une introduction succincte de ces notions.

2.1 Java vs. code-octet

Le code-octet est un langage de programmation à pile de bas niveau. À l'origine, il a été conçu comme une représentation intermédiaire, normalement générée par un compilateur, entre du code source de haut niveau (principalement en langage Java) et un langage machine assembleur. Ce code est normalement destiné à être exécuté par une machine virtuelle et est distribué dans le format *class* sous forme de fichier ou de flux de données dans un réseau. Un fichier au format *class* représente une classe ou une interface en langage de haut niveau, orienté-objet. L'utilisation d'un format intermédiaire favorise la portabilité des programmes entre différentes architectures systèmes. La spécification de la machine virtuelle Java [LinY99] est une référence décrivant ce format de fichier. Certains des principaux éléments le constituant sont présentés dans cette section, mais l'intention n'est pas de faire un résumé complet de la spécification.

2.2 Format de fichier *class*

Un fichier *class* est encodé en une séquence de codes-octets. Un code-octet est représenté sur 8 bits. Les structures constituant le fichier *class* sont représentées avec des séquences de 1, 2 ou 4 octets. Les notations utilisées pour les représenter sont `u1`, `u2`, `u4`.

Un fichier *class* contient des informations comme la version du format de fichier utilisée, une table de constantes (`constant_pool`), la super-classe, les interfaces implémentées, les champs, les méthodes et les attributs de la classe ou de l'interface. Toute référence à une classe, une interface, une méthode ou un champ faite dans le fichier (par exemple la super-classe ou, dans le code d'une méthode, une instruction d'invocation de méthode) est exprimée comme une référence symbolique à une constante du `constant_pool`. Cette constante exprimera sous forme de chaîne de caractères au format *utf-8* [ISO03] le nom ou la description de l'item référé. Par exemple, l'item `super_class` d'une classe contient l'indice qui doit désigner une constante de type `CONSTANT_Class_info` dans la `constant_pool`. Cette constante indique le nom complet de la classe; `java/lang/String` par exemple.

Les méthodes d'une classe sont encodées dans des structures `method_info`. Cela indique les propriétés de contrôle d'accès à la méthode (publique, privée, ...), son nom, son descripteur et ses attributs. Le descripteur est représenté par une chaîne de caractères exprimant les paramètres entre parenthèses suivi du type de retour. Par exemple, le descripteur de la méthode « `int test(boolean b, Object o)` » est noté `(ZLjava/lang/Object;)I`. Nous distinguons le `Z` pour `boolean`, `Ljava/lang/Object;` pour la référence à la classe `Object` et `I` pour le type de retour `int`.

Le code d'une méthode est représenté par une séquence de codes-octets dans l'attribut `code` de la méthode.

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Figure 2.1 : Schéma du format de fichier class

Source: Lindholm, Tim, et Yellin, Frank. *The Java Virtual Machine Specification, deuxième édition*. http://java.sun.com/docs/books/vmspec, 1999, section 4.1.

2.3 Code-octet et instructions d'une méthode

Le format de chaque instruction du tableau de code est présenté au chapitre 6 de la spécification de la machine virtuelle. Nous en présentons ici les principes directeurs.

Une instruction est identifiée par un code d'opération. Elle a possiblement des opérandes encodés dans le tableau de code. Par exemple, l'instruction `iload_1` qui charge le contenu de la variable locale `1` sur la pile est représentée par le code-octet `27` et

l'instruction `iload 10`, qui charge le contenu de la variable locale `10` sur la pile, est représentée par les codes-octets `21 10`.

Certaines instructions peuvent aussi récupérer dynamiquement (lors de l'exécution) leurs opérandes sur la pile d'opérandes de la méthode courante. Par exemple, `iadd` reçoit deux valeurs de type `int` sur la pile et empile ensuite une valeur de type `int`.

Cela introduit un concept nouveau : le fait que le code-octet est interprété par une machine à pile. L'exécution de chaque méthode est effectuée dans un espace mémoire restreint comprenant, principalement, les variables locales à la méthode ainsi qu'une pile d'opérandes servant à effectuer des calculs. Les valeurs temporaires sont donc empilées et dépilées par différentes instructions afin de calculer les opérations de la méthode.

L'exemple de la figure 2.2 montre le calcul de l'équation `a = b + c` en utilisant une pile `P` et avec `b = 1` et `c = 2`. Les opérandes `b` et `c` doivent d'abord être empilés. L'opération d'addition les dépile et empile le résultat. Cette valeur est ensuite dépilée pour être affectée à la variable locale `a`.

Instruction		Pile		Variables locales		
		0	1	a	b	c
empiler(P, b)	avant:	-	-	-	1	2
	après:	1	-	-	1	2
empiler(P, c)	avant:	1	-	-	1	2
	après:	1	2	-	1	2
additionner	avant:	1	2	-	1	2
	après:	3	-	-	1	2
affecter(a, dépiler(P))	avant:	3	-	-	1	2
	après:	-	-	3	1	2

Figure 2.2 : Additionner avec une machine à pile

2.4 Des instructions typées

Les instructions du code sont typées, c'est-à-dire qu'elles sont spécialisées pour opérer sur un type (ou une famille de types) de données précis. Les types de données reconnus par la machine virtuelle sont répartis en **types primitifs** et en **types références**.

Les types primitifs désignent premièrement des nombres entiers représentés en complément à deux, ou des caractères : `byte` (8 bits signés), `short` (16 bits signés), `int` (32 bits signés), `long` (64 bits signés) et `char` (16 bits non-signés représentant des caractères Unicode). Le type `boolean` est aussi représenté sur 32 bits. Cela dit, les structures de la machine virtuelle étant en 32 bits, le type `int` généralise les types `byte`, `short`, `char` et `boolean`. Il y a aussi les nombres à virgule flottante : `float` (32 bits signés, simple précision) et `double` (64 bits signés, double précision). Enfin, le type `returnAddress` (32 bits) représente l'adresse relative d'une instruction dans le tableau de code; ce type ne correspond cependant pas à un type de données de haut niveau du langage Java mais est généré par un compilateur pour supporter la construction de sous-routines. Nous reviendrons à ce concept plus en détail à la section 3.4.

Les types références représentent des structures dynamiques. Ce sont les classes, les interfaces et les tableaux. La valeur `null` est compatible avec tous les types de référence.

Le traitement des différents types de données est donc effectué par des familles d'instructions dont les membres sont spécialisés pour un type de donnée. Par exemple, en Java, une addition peut avoir comme opérande des valeurs de types primitifs numériques. Nous distinguons donc quatre instructions d'addition : `iadd`, `ladd`, `fadd` et `dadd`, qui opèrent respectivement sur des données de types entier (`int`), entier (`long`), virgule flottante simple précision (`float`) et virgule flottante double précision (`double`).

Reprenons l'exemple de la figure 2.2 pour implémenter l'équation `a = b + c` en utilisant les instructions du code-octet. Nous représentons les variables locales `a`, `b` et `c` par les indices 0, 1 et 2. Nous remplaçons l'opération `empiler(P, b)` par l'instruction `iload_1`, l'opération `empiler(P, c)` par l'instruction `iload_2`, l'opération `additionner` par l'instruction `iadd` et l'opération `affecter(a, dépiler(P))` par l'instruction `istore_0`. La figure 2.3 montre cet exemple.

Instruction		Pile		Variables locales		
		0	1	0	1	2
iload_1	avant:	-	-	-	1	2
	après:	1	-	-	1	2
iload_2	avant:	1	-	-	1	2
	après:	1	2	-	1	2
iadd	avant:	1	2	-	1	2
	après:	3	-	-	1	2
istore_0	avant:	3	-	-	1	2
	après:	-	-	3	1	2

Figure 2.3 : Additionner un deux entiers avec le code-octet

2.5 Contraindre les instructions

Certaines règles régissent l'agencement des instructions dans le tableau de code. Il s'agit des contraintes statiques et dynamiques présentées à la section 4.8 de [LinY99].

Les contraintes statiques concernent la disposition des codes-octets et aussi les relations entre les instructions et les opérandes. Par exemple, la première instruction doit être inscrite à l'adresse relative 0 du tableau de code; ou encore, l'adresse relative en opérande d'une instruction effectuant un branchement doit désigner une instruction valide de la même méthode.

Les contraintes dynamiques, aussi nommées contraintes structurelles, concernent l'interaction des instructions entre elles. Par exemple, à aucun moment de l'exécution d'un programme une instruction ne doit causer un débordement dc la pile d'opérandes de la machine virtuelle; aussi, les instructions d'invocation de méthode doivent être exécutées avec des arguments appropriés (en nombre et en types) comme opérandes sur la pile.

2.6 Code-octet et sécurité

Bien que les compilateurs respectent en général ces contraintes, le processus de vérification est une mesure de sécurité pour palier à toute éventualité de corruption des données, que ce soit lors du stockage ou de la transmission sur un réseau, ou alors pour détecter d'éventuelles attaques envers la machine virtuelle avec des programmes intentionnellement mal formés. En effet, une classe peut être distribuée sur un réseau ouvert sans que l'on puisse s'assurer de l'authenticité du distributeur ou de l'intégrité des données.

La figure 2.4 illustre à haut niveau le processus de distribution d'une classe et l'interaction entre la machine virtuelle et la vérification.

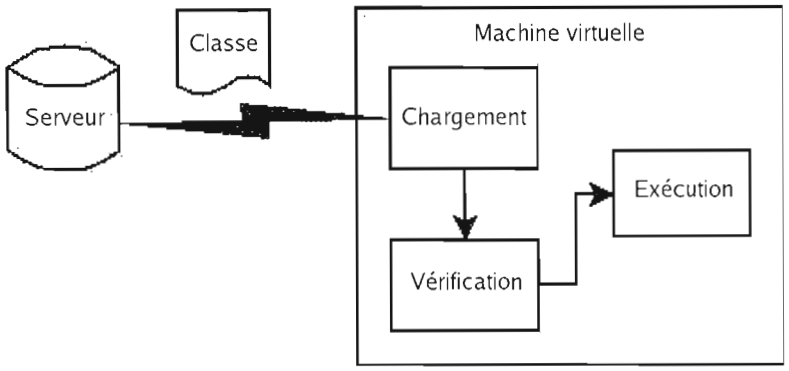

Figure 2.4 : Schéma sommaire de distribution d'une classe

La vérification d'une classe a lieu une seule fois, après le chargement et lors de l'édition de liens de la classe. Les méthodes de la classe peuvent ensuite être exécutées de manière « agressive », c'est-à-dire sans se soucier de la disposition des données dans le tableau de code.

Le processus de vérification a donc pour but premier d'assurer que le code des méthodes d'une classe respecte les contraintes statiques et (surtout) dynamiques du code-octet. Nous verrons aux chapitres 3 et 4 deux techniques pour effectuer la vérification du code-octet.

CHAPITRE III

VÉRIFICATION EN UN TEMPS

Un programme compilé au format *class* est encodé en une séquence de codes-octets. Celle-ci ne comportant pas suffisamment d'information concernant le comportement du programme lors de son exécution, certaines analyses sont nécessaires afin de l'estimer et de vérifier qu'il n'effectue pas d'opération compromettante pour la machine virtuelle.

Nous présenterons dans ce chapitre une approche de vérification du code-octet utilisant une analyse de flot de données intra-procédurale et polyvariante. Plus précisément, nous présenterons dans un premier temps la notion de graphe de contrôle, son utilité et un algorithme pour le construire. Nous présenterons ensuite formellement l'analyse de flot de données : les types, les structures de données, le treillis, les fonctions de transition et l'algorithme de calcul. Les difficultés reliées à l'analyse de flot de données en présence des instructions `jsr` et `ret` et des gestionnaires d'exception seront exposées et nous montrerons comment les analyser. Enfin, un algorithme de vérification en un temps sera présenté.

3.1 Graphe de flot de contrôle

Une analyse de flot de contrôle sert à créer une abstraction de l'ordonnancement des instructions d'une méthode. L'analyse portera sur les méthodes non abstraites et non

natives[1] d'une classe. Cette analyse utilise un graphe orienté pour représenter tous les chemins d'exécution d'une méthode. Nous présentons dans ce qui suit un exemple de construction de graphe de flot de contrôle suivi d'une définition formelle de la structure de données de graphe utilisée et, pour terminer, un algorithme de construction de graphe.

Exemple

Afin d'acquérir une intuition de la notion de graphe de contrôle en relation avec le code-octet et le code Java, observons les figures 3.1 et 3.2. Il s'agit d'un programme simple retournant une valeur pseudo-aléatoire négative. La méthode `Random.nextInt()` est invoquée, itérativement, jusqu'à ce que le résultat soit négatif. La variable locale 0 est utilisée pour mémoriser le résultat, tester s'il est plus grand que 0 et pour retourner la valeur.

3.1.1 Modèle et structures de données

Nous notons $G = \langle N, E \rangle$, un graphe orienté avec l'ensemble de noeuds N et l'ensemble d'arcs $E \subseteq N \times N$. Un arc est une relation entre deux noeuds $n, p \in N$ et est noté (n, p). À un noeud n sont associés une instruction avec son adresse relative dans le tableau de code et deux ensembles de noeuds désignant les prédécesseurs et les successeurs. Ces ensembles sont notés ainsi : soit l'arc $e \in E$,

$$pred(n) = \{p \in N \mid \exists e = (p, n)\} \text{ et } succ(n) = \{p \in N \mid \exists e = (n, p)\}.$$

Notons aussi les noeuds spéciaux *entrée* et *sortie* que nous décrirons avec l'algorithme de construction du graphe.

---

1 Les méthodes abstraites n'ont pas de code et les méthodes natives ont du code dans un langage autre que le code-octet et n'est pas encodé dans le fichier *class*.

```
public static int ex3_1()
{
   int x;

   do
   {
     x = random.nextInt();
   }
   while(x > 0);

   return x;
}
```

```
 0: getstatic #2;      //Champs random:Ljava/util/Random;
 3: invokevirtual #3; //Méthode java/util/Random.nextInt:()I
 6: istore_0
 7: iload_0
 8: ifgt 0
11: iload_0
12: ireturn
```

Figure 3.1 : Exemple de programme Java avec sa traduction en codes-octets

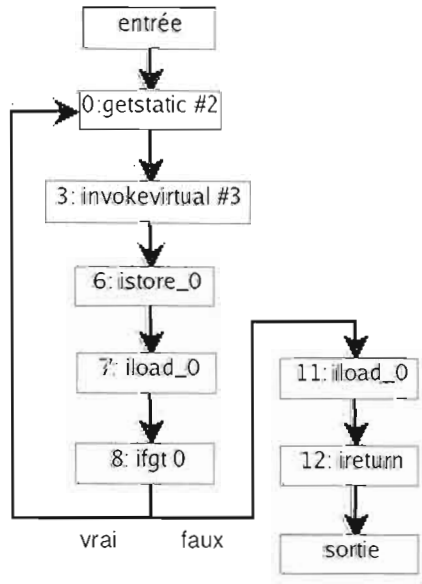

Figure 3.2 : Exemple de graphe de contrôle pour le programme 3.1

3.1.2 Algorithme de construction du graphe de contrôle

Dans cette section, nous présentons un algorithme de construction de graphe de contrôle pour des méthodes ne comportant pas d'instruction `jsr` ou `ret` ni de mécanismes de gestion d'exception.

Il s'agit d'un algorithme itératif avec *worklist* adapté de Kildall [Kild73]. Une *worklist* est un ensemble ordonné servant à mémoriser les noeuds à traiter. L'algorithme l'utilise en tant que file. L'algorithme commence par créer un noeud *entrée* correspondant à la première instruction du tableau de code de la méthode à analyser. Ses ensembles *prédécesseurs* et *successeurs* sont initialement vides. Ce noeud est ensuite inséré dans le graphe et dans la *worklist*. L'algorithme procède ainsi : tant que la *worklist* n'est pas vide : récupérer un noeud n de la *worklist*, identifier ses successeurs, créer un noeud pour chacun de ceux-ci si le graphe ne les contient pas encore, créer un arc entre le noeud n et chacun de ses successeurs et les insérer dans la *worklist* s'ils ne sont pas marqués comme étant déjà visités. La *worklist* étant un ensemble, on n'y insérera pas un noeud s'il s'y trouve déjà. Enfin, on marque le noeud n comme étant visité et on recommence le processus.

Chaque instruction de la méthode sera visitée au plus une fois. Le code mort, instructions ne faisant partie d'aucun chemin partant de la première instruction de la méthode, ne sera pas inclus dans le graphe qui est construit progressivement en considérant les chemins possibles à partir d'une instruction « vivante ».

3.1.3 Identification des successeurs

Plusieurs cas de figure existent en ce qui a trait aux chemins de contrôle. Nous avons identifié six types de branchement pouvant qualifier le comportement d'une instruction.

Nous donnons ici une description de chacun.

La première famille est la séquence. Une instruction de ce type n'a qu'un successeur, soit l'instruction suivante dans le tableau de code. Des codes-octets comme `iadd`, `astore` ou `putfield`, servant à manipuler des opérandes sur la pile ou les variables locales, sont des exemples d'instructions séquentielles.

La deuxième famille est celle des branchements inconditionnels. Ces instructions ont un successeur. Cela se résume à l'instruction `goto t` (avec son extension `goto_w t`) dont l'adresse relative du successeur est spécifiée en opérande de l'instruction dans le tableau de code.

La troisième famille est formée des branchements conditionnels. Il s'agit par exemple de codes-octets tels `ifeq`, `lcmp`, `if_acmpeq`. Ces instructions évaluent un opérande sur la pile et décident du branchement à effectuer en conséquence, le successeur pouvant être l'instruction suivant séquentiellement l'instruction courante ou alors celle à l'adresse relative spécifiée en opérande de l'instruction dans le tableau de code. Bien que lors d'une exécution réelle du programme un seul chemin sera emprunté, nous sommes incapables de décider lequel durant l'analyse car il est déterminé par la valeur d'une expression sur la pile[1]. Nous identifierons donc comme successeurs les deux possibilités de branchement.

La quatrième famille comprend les instructions `tableswitch` et `lookupswitch`. Dans ce cas, on estime que toutes les possibilités de branchement, telles qu'encodées en opérande de l'instruction dans le tableau de code sont des successeurs de l'instruction.

---

1 La valeur de cette expression est calculée dynamiquement et même une analyse de propagation de constantes ne serait pas suffisante pour pouvoir l'évaluer en tout temps car elle peut dépendre d'une entrée dont la source est extérieure au programme.

La cinquième famille de branchement, relative à la gestion des exceptions d'une méthode, ainsi que la sixième famille qui consiste en la paire d'instructions `jsr - ret` nécessitent un traitement particulier qui sera présenté dans les sections 3.3 et 3.4.

Les noeuds terminaux constituent l'ensemble des noeuds de sortie du graphe et ont comme successeur le noeud spécial *sortie* . Pensons ici au code-octet `return` (et à toutes ses variantes `ireturn`, `areturn`, ...) qui signifie de retourner l'exécution à l'appelant de la méthode et qui n'a aucun successeur. Il y a aussi `athrow`, qui lance une exception; l'exception pourrait être attrapée par un gestionnaire d'exception (`exception_handler`) lors de l'exécution, mais ne sachant l'estimer lors de l'analyse, nous supposons que cela peut ne pas être le cas. Ses successeurs seront donc les `exception_handlers` s'il y en a, sinon il n'en aura pas d'autre.

3.2 Introduction à l'analyse de flot de données

Ayant estimé de manière abstraite dans quel ordre les instructions d'un programme seront exécutées, nous allons en tirer parti afin d'estimer ce qu'elles peuvent faire. Ceci peut être calculé par une analyse de flot de données qui examine le traitement de données effectué par un programme afin d'en extraire de l'information et des propriétés utiles dans un but précis. Par exemple, cette technique est couramment utilisée dans le domaine des compilateurs optimisants. Dans le cadre de la vérification, l'information que nous recherchons est le type de chaque variable pouvant atteindre toute instruction d'une méthode. Le type d'analyse que nous utiliserons peut être effectué de manière semblable à une interprétation abstraite sur des types en associant une équation de flot de données à l'instruction de chaque noeud du graphe de flot de contrôle et en les évaluant répétitivement.

Nous nous concentrons sur un type d'analyse dite intra-procédurale, globale et polyvariante. Intra-procédurale parce qu'elle porte sur le code d'une seule méthode à la fois, contrairement à interprocédurale qui analyse les relations entre plusieurs méthodes[1]. Globale car elle traite l'ensemble de toutes les instructions d'une méthode. Polyvariante car elle permet d'évaluer plusieurs états des données possibles pour une même instruction. Nous reviendrons sur cette notion.

Dans cette section seront présentés les fondements mathématiques de l'analyse de flot de données : les types de données, l'abstraction des structures de données, le treillis hiérarchisant les données, les équations de flot de données, les fonctions de transition et l'algorithme de calcul de l'analyse. Nous montrerons aussi comment structurer cette information en vue d'effectuer la vérification du code. Il s'agit d'une présentation préliminaire qui sert à l'analyse du code de méthodes sans instructions `jsr` et `ret` et sans gestionnaire d'exception.

3.2.1 Les types

Les données à analyser dans le but d'effectuer la vérification du code-octet sont les types des variables. Il y a les types primitifs, les références et d'autres types supplémentaires.

Les types primitifs sont : `int`, `byte`, `short`, `char`, `boolean`, `float`, `llong`, `hlong`, `ldouble`, `hdouble`. Les types `long` et `double` sont dits de « catégorie 2 », les autres étant de « catégorie 1 ». La représentation interne des types de catégorie 1 est faite sur quatre octets (32 bits) et celle des types de catégorie 2 est faite sur huit octets (64 bits). Un emplacement de la pile ou des variables locales d'une méthode comportant quatre

1 L'approche interprocédurale peut donner des résultats plus précis pour certaines analyses (en vue de l'allocation de registres par exemple) mais est plus complexe et n'est pas nécessaire pour effectuer la vérification.

octets, ces derniers types sont subdivisés en parties basse (`llong, ldouble`) et haute (`hlong, hdouble`). Nous y reviendrons dans la section suivante.

Les types référence comprennent : `initReference(C), uninitReference(C, offset), uninitThis, array(dim,type), null, reference(r).`

Une référence initialisée *C* (`initReference(C)`) représente une classe ou une interface dont le nom se retrouve dans le *constant_pool* de la classe courante. L'analyse utilise une référence symbolique consistant aux caractères délimiteurs « `L` » et « `;` » avec le nom de la référence. Par exemple, une référence à la classe *C* est notée `LC;` et une référence à la classe `java.lang.String` est notée `Ljava/lang/String;`.

Une référence non initialisée (`uninitReference(C,offset)`) est identifiée par le nom de la référence et l'adresse relative de son instanciation (instruction `new C`) afin de pouvoir distinguer plusieurs instances non initialisées d'une même référence. Voir la spécification de la machine virtuelle Java à la section 4.9.4 [LinY99] pour plus de détails.

Une référence non initialisée à la classe courante (`uninitThis`) est utilisée dans les constructeurs. Les constructeurs reçoivent une instance non initialisée dans la variable locale 0 afin de l'initialiser.

Les tableaux (`array(dim,type)`) sont représentés avec leur dimension et leur type de base. Le type de base peut être un type primitif ou une référence. Les instructions `anewarray` et `newarray` créent des tableaux d'une dimension et l'instruction `multianewarray` crée des tableaux à plusieurs dimensions. L'instruction `aaload` a pour fonction d'extraire un élément d'un tableau de références. S'il s'agit d'un tableau multidimensionnel, l'élément extrait sera un nouveau tableau dont la dimension sera égale à celle du tableau d'origine moins 1 et dont le type de base sera le même.

Le type `null` est compatible avec toutes les références. Nous notons aussi `reference(r)` une référence générique où `r` peut être tout type de référence.

Parmi les types supplémentaires, nous ne considérons pour le moment que `uninitialized`. Une variable dont le type n'est pas connu ou une variable qui a été invalidée est marquée comme étant non initialisée (`uninitialized`). Les instructions `jsr` et `ret` requièrent des types spéciaux qui seront présentés dans la section 3.4.

3.2.2 Abstraction

Les noeuds composant le graphe de contrôle doivent être enrichis de structures de données pour représenter les types des variables avant (*in*) et après (*out*) l'exécution de l'instruction d'un noeud. Les variables correspondent aux emplacements de la pile et des variables locales d'un bloc d'activation (*frame*) de la machine virtuelle. Chaque emplacement dans ce bloc d'activation est un ensemble de types (d'où l'aspect polyvariant mentionné en introduction, car tous les types pouvant atteindre un noeud sont mémorisés).

Soit k, la hauteur maximale de la pile et l, le nombre maximal de variables locales, tel que spécifié dans l'attribut `Code` de la méthode. Nous notons $S = (s_0, s_1, \ldots, s_{k-1})$ la pile et $R = (r_0, r_1, \ldots, r_{l-1})$ les variables locales (ou registres) comme des séquences d'ensembles de types. Un bloc d'activation peut être représenté comme un couple $F = (S, R)$.

Nous noterons donc un noeud $n = (\mathit{offset}, \mathit{instruction}, \mathit{pred}, \mathit{succ}, \mathit{in}, \mathit{out})$, où *offset* est l'adresse relative de l'instruction dans le tableau de code tel que $0 \leq \mathit{offset} < \mathit{code\_lentgh}$ (la longueur du tableau de code), *pred* et *succ* sont des ensembles de noeuds et *in* et *out* désignent les blocs d'activation avant et après la visite du noeud.

3.2.3 Treillis

Un treillis est une structure algébrique représentant un ensemble dont les éléments sont partiellement ordonnés et dont toute paire d'éléments est bornée inférieurement et supérieurement. Nous utilisons un treillis (*lattice*), dénoté $(\boldsymbol{L}, \cap, \cup, \subseteq)$, dont les éléments sont des ensembles de types. Les bornes inférieure et supérieure sont définies par les opérateurs ensemblistes $\cap$ et $\cup$. La relation d'ordre est définie par l'opérateur d'inclusion ensembliste $\subseteq$. L'élément inférieur du treillis est l'ensemble vide ($\perp$ = *bottom*) et l'élément supérieur est l'ensemble de tous les types potentiellement utilisés par le code d'une méthode ($\top$ = *top*).

Par exemple, le code de la figure 3.3 implique les types `int`, `uninitialized` et `boolean` et est représenté par le treillis de la figure 3.4 qui représente toutes les combinaisons possibles d'ensembles composés de ces types.

```
 0:  iload_0
 1:  ifeq 9
 4:  iconst_1
 5:  istore_1
 6:  goto 11
 9:  iconst_0
10:  istore_1
11:  iload_1
12:  ireturn
```

Figure 3.3 : Exemple de programme

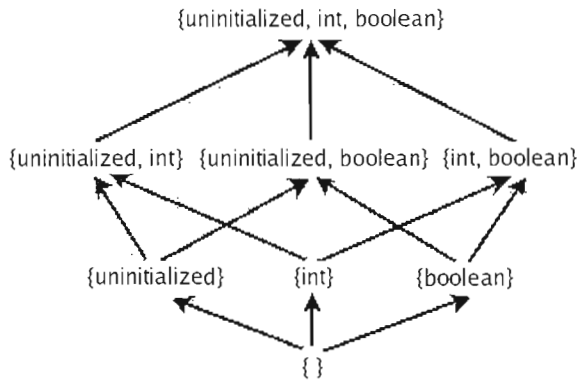

Figure 3.4 : Treillis d'ensembles de types pour le programme de la figure 3.3

3.2.4 Fonctions de transition

Afin d'évaluer le comportement d'un programme, l'effet de chaque instruction sur les variables doit être calculé. Rappelons que l'information que nous recherchons est le type des variables préalablement à l'exécution des instructions.

Pour le calculer, nous utilisons des familles de fonctions sur le treillis ***L***. Par évaluation répétitive des fonctions de transition associées aux instructions des noeuds du graphe, les données calculées vont se stabiliser et atteindre un *point-fixe* si les fonctions sont monotones (croissantes). L'algorithme utilisé pour calculer l'analyse de flot de données se terminera donc et produira comme résultat un estimé conservateur du comportement du programme lors d'une exécution réelle. Cet estimé doit être suffisamment précis car s'il est trop grossier, il ne donnera pas d'information utile. Kildall [Kild73] et Muchnick [Much97] ont démontré que le résultat d'un tel algorithme est le *point-fixe* minimal si les fonctions sont aussi distributives. Une fonction $f : L \rightarrow L$ est :

- monotone si $\forall x, y \;\; x \subseteq y \rightarrow f(x) \subseteq f(y)$ et

- distributive si $\forall x, y \;\; f(x \cap y) = f(x) \cap f(y)$.

Nous montrons maintenant quelques exemples de fonctions de transition adaptées de [SaVM06].

Rappelons qu'un bloc d'activation est noté $F = (S, R)$ avec S et R des séquences d'ensembles de types. Dans le but de simplifier la notation du formalisme, nous considérons un bloc d'activation comme étant une seule séquence d'ensembles de types résultant de la concaténation de la pile et des variables locales.

Soit des blocs d'activation x et y, $n = |S| + |R|$ (hauteur maximale de la pile + nombre maximal de variables locales) le nombre d'éléments du bloc d'activation, i tel que $0 \leq i < n$, h la hauteur de la pile avant l'exécution de l'instruction tel que $0 \leq h \leq |S|$, l un indice de variable locale tel que $0 \leq l < |R|$ et $l' = |S| + l$, nous avons que,

si $x = (x_0, ..., x_{n-1})$ alors

- $iconst_h(x) = (y_0, ..., y_{n-1})$ où

$$y_i = \begin{cases} int & \text{si } i = h \\ x_i & \text{sinon} \end{cases}$$

- $istore_{hl}(x) = (y_0, ..., y_{n-1})$ où

$$y_i = \begin{cases} x_{h-1} & \text{si} \quad i = l' \\ \{u\} & \text{si} \quad i = h-1 \\ x_i & \text{sinon} \end{cases}$$

- $aaload_h(x) = (y_0, ..., y_{n-1})$ où

  $x_{h-1} = \{array(d_1, ref(r_1)), ..., array(d_2, ref(r_2))\} \cup C$

  avec d_k la dimension du tableau de références,

  C l'ensemble des autres types pouvant appartenir à x_{h-1}, et

$$y_i = \begin{cases} \{u\} & \text{si } i = h-1 \\ \{r_1, ..., r_n\} & \text{si } i = h-2 \ \wedge \ C = \emptyset \wedge d_k = 1 \\ \{array(d_1 - 1, ref(r_1)), ..., & \\ \ array(d_2 - 1, ref(r_2))\} & \text{si } i = h-2 \ \wedge \ C = \emptyset \wedge d_k > 1 \\ \{u\} & \text{si } i = h-2 \ \wedge \ C \neq \emptyset \\ x_i & \text{sinon} \end{cases}$$

- $new_{(C,o),h}(x) = (y_0, ..., y_{n-1})$ où

  o est l'adresse relative de l'instruction **new** dans la méthode et

$$y_i = \begin{cases} \{uninitReference(C, o)\} & \text{si } i = h \\ x_i & \text{sinon} \end{cases}$$

Brièvement, voici une description du comportement des instructions associées à ces fonctions de transition. `Iconst i` empile un ensemble dont l'élément est le type `int`. `Istore 1` dépile un ensemble et l'affecte à la variable locale *l*. `Aaload` dépile premièrement un type `int` et ensuite un ensemble devant comporter exclusivement des tableaux de références. L'instruction empile ensuite un autre ensemble comprenant les types des éléments de ces tableaux. Si une référence est un tableau multidimensionnel, le nouveau type sera un tableau dont la dimension sera égale à celle du tableau d'origine moins 1 et dont le type de base sera le même. Si l'ensemble en opérande comporte d'autres types que des tableaux de références, l'ensemble empilé aura comme seul élément le type `uninitialized` signifiant que la transition n'a pu être effectuée correctement. `New(C,o)` empile un ensemble dont l'élément est `uninitReference(C, o)`.

3.2.5 Algorithme

Dans la section 3.1.2 nous avons exposé un algorithme de construction de graphe de contrôle. Nous présenterons dans cette section comment adapter cet algorithme afin de calculer concurremment le graphe de contrôle et le *point-fixe* minimal de l'analyse de flot de données pour une méthode sans les instructions `jsr` et `ret` et sans gestionnaire d'exception.

Initialement, créer le noeud d'*entrée* du graphe. Ce noeud a comme unique successeur le noeud correspondant à la première instruction de la méthode. Les types des variables qui atteignent ce noeud (*entrée.in*) sont déterminés par le descripteur de la méthode. La pile est vide. Pour les variables locales, il y a trois cas. Si la méthode est statique, le type de chaque paramètre du descripteur de la méthode est affecté aux variables locales de façon séquentielle à partir de l'indice 0. Les types de « catégorie 2 » (`long` et `double`) utilisent deux emplacements chacun. Si la méthode est un constructeur (nommée `<init>`), l'emplacement 0 des variables locales sera initialisé avec le type `uninitThis` et les types des paramètres suivent séquentiellement. Pour toute autre méthode, la variable locale 0 contient une référence à la classe courante, déclarant la méthode, et les types des paramètres suivent séquentiellement. Les autres variables locales dont le type est inconnu sont initialisées avec le type `uninitialized`. Le noeud *entrée* est ensuite inséré dans le graphe et dans la *worklist*.

Pour effectuer l'analyse, répéter itérativement le processus suivant tant que la *worklist* n'est pas vide. Récupérer un noeud *n* de la *worklist*. Calculer la fonction de transition associée à l'instruction de ce noeud, prenant *n.in* en entrée et affectant le résultat à *n.out*. L'ensemble des types pour chaque emplacement en entrée d'un noeud est l'union des types des emplacements correspondant en sortie des prédécesseurs de *n*. Si la fonction a modifié l'état des types en sortie, insérer les successeurs de *n* dans la *worklist*. Les successeurs sont

identifiés seulement lors de la première visite de chaque noeud en suivant les règles mentionnées dans la section 3.1.3 et ils sont alors insérés au graphe. La *worklist* étant un ensemble, on n'y insérera pas un noeud s'il s'y trouve déjà. Enfin, s'il s'agit de la première visite du noeud, marquer le noeud *n* comme étant visité et recommencer le processus.

Afin d'illustrer l'algorithme, nous en effectuons la trace en reprenant l'exemple de la figure 3.1. La figure 3.5 montre une trace simplifiée de l'algorithme n'utilisant pas de *worklist* pour déterminer les noeuds restant à traiter mais itérant sur tous les noeuds à chaque passe.

Notons d'abord que la méthode est statique et ne comporte qu'une variable locale (`int x`). De plus, la hauteur maximale de la pile est déterminée à 1. Les séquences des blocs d'activation représentant la pile (S) et les variables locales (R) comporteront donc un élément chacune. Au départ, la pile est vide et cela est dénoté par le type {U}. De plus, le type de la variable locale est inconnu car elle n'a pas été initialisée et le compilateur n'a pas transmis d'information à ce sujet dans le code-octet.

Nous voyons ainsi que la variable locale comporte d'abord le type {U} lors de la première itération jusqu'à l'instruction 6 (`istore_0`) qui lui affecte le type {I}. L'instruction de branchement à l'adresse 8 propagera ce type à l'adresse 0 lors de la deuxième itération, ce qui produira l'ensemble {I, U}. Constatant que la troisième itération ne modifie aucun état des types, l'algorithme se termine et produit le *point-fixe*.

Instruction		Itération 1		Itération 2		Itération 3	
		S	R	S	R	S	R
		0	0	0	0	0	0
0: getstatic 2	avant:	{U}	{U}	{U}	{I, U}	{U}	{I, U}
	après:	{LRandom;}	{U}	{LRandom;}	{I, U}	{LRandom;}	{I, U}
3: invokevirtual 3	avant:	{LRandom;}	{U}	{LRandom;}	{I, U}	{LRandom;}	{I, U}
	après:	{I}	{U}	{I}	{I, U}	{I}	{I, U}
6: istore_0	avant:	{I}	{U}	{I}	{I, U}	{I}	{I, U}
	après:	{U}	{I}	{U}	{I}	{U}	{I}
7: iload_0	avant:	{U}	{I}	{U}	{I}	{U}	{I}
	après:	{I}	{I}	{I}	{I}	{I}	{I}
8: ifgt 0	avant:	{I}	{I}	{I}	{I}	{I}	{I}
	après:	{U}	{I}	{U}	{I}	{U}	{I}
11: iload_0	avant:	{U}	{I}	{U}	{I}	{U}	{I}
	après:	{I}	{I}	{I}	{I}	{I}	{I}
12: ireturn	avant:	{I}	{I}	{I}	{I}	{I}	{I}
	après:	{U}	{I}	{U}	{I}	{U}	{I}

Figure 3.5 : Trace d'analyse de flot de données

3.3 Analyse de gestionnaire d'exception

Le mécanisme de gestion des exceptions d'une méthode pose une particularité quant à la détermination du flot de contrôle ainsi qu'à la propagation des types lors de l'analyse de flot de données.

Toute instruction du code d'une méthode peut être couverte par un gestionnaire d'exception. La table d'exception (`exception_table`) associée à l'attribut `Code` d'une méthode identifie les séquences d'instructions qui sont couvertes par un ou plusieurs `exception_handler` en y spécifiant l'adresse relative (`handler_pc`) et la classe d'exception associée (`exception_type`).

Lors de l'exécution du programme, une exception lancée par une instruction sera attrapée par le premier `exception_handler` dont le type d'exception est compatible et seul ce code sera exécuté. À fin d'analyse, une instruction couverte par un ou plusieurs `exception_handlers` est considérée comme pouvant effectuer un branchement à tous ces

`exception_handlers`. Aux successeurs d'un tel noeud devraient donc s'ajouter les premières instructions de ceux-ci, identifiées par `handler_pc` dans la table d'exception. Mais considérant que l'exécution de l'instruction en question n'a pu se compléter, les types propagés sont ceux en entrée (*n.in*) et non *n.out*. Pour traiter cette particularité, un noeud intermédiaire est donc ajouté entre l'instruction et l'`exception_handler`. Ce noeud a comme clé dans le graphe la valeur négative de l'adresse relative du gestionnaire d'exception (`handler_pc` * -1). Les types en entrée de ce noeud sont l'union de tous les types en entrée des instructions couvertes par le gestionnaire d'exception. La fonction de transition associée à ce noeud spécial va vider la pile d'opérandes du bloc d'activation et empiler une référence au type d'exception (`exception_type`) du gestionnaire d'exception associé.

Exemple

Voyons une méthode comportant des gestionnaires d'exception avec le graphe de contrôle et l'analyse de flot de données associés, montrés dans les figures 3.6, 3.7 et 3.8.

Nous voyons qu'il y a deux `exception_handlers` qui couvrent les instructions 0 et 1. Ils sont associés aux noeuds -5 et -17. Nous pouvons constater que les fonctions de transitions pour les exceptions vont vider la pile et empiler les références `LClassCastException;` (noté `LCastException;` dans la figure 3.8) et `LException;`. L'instruction à l'adresse 20 vient après les gestionnaires d'exception. Nous voyons que les emplacements de la pile et des variables locales contiennent l'ensemble de tous les types pouvant les atteindre : `{NULL, LA;, LB;}` pour la variable locale 1 et `{LClassCastException;, LException;}` pour la variable locale 2.

```
public static A test(B b) {
    A a;
    try {a = (A)b;}
    catch(ClassCastException cce) {a = new A();}
    catch(Exception e) {a = null;}
    return a;
}
```

```
 0: aload_0
 1: astore_1
 2: goto 20
 5: astore_2
 6: new A
 9: dup
10: invokespecial A.<init>()V
13: astore_1
14: goto 20
17: astore_2
18: aconst_null
19: astore_1
20: aload_1
21: areturn
  Exception table:
   from   to  target type
     0     2     5   java/lang/ClassCastException
     0     2    17   java/lang/Exception
```

Figure 3.6 : Gestion des exceptions (code)

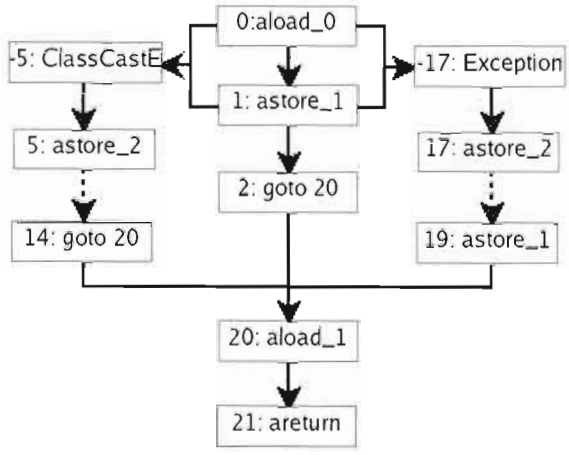

Figure 3.7 : Gestion des exceptions (graphe)

Instruction		Pile		Variables locales		
		0	1	0	1	2
0: aload_0	in:	{U}	{U}	{LB;}	{U}	{U}
	out:	{LB;}	{U}	{LB;}	{U}	{U}
1: astore_1	in:	{LB;}	{U}	{LB;}	{U}	{U}
	out:	{U}	{U}	{LB;}	{LB;}	{U}
2: goto 20	in:	{U}	{U}	{LB;}	{LB;}	{U}
	out:	{U}	{U}	{LB;}	{LB;}	{U}
-5: *ClassCast*	in:	{U}	{U}	{LB;}	{U}	{U}
	out:	**{LCastExcep;}**	{U}	{LB;}	{U}	{U}
5: astore_2	in:	**{LCastExcep;}**	{U}	{LB;}	{U}	{U}
	out:	{U}	{U}	{LB;}	{U}	**{LCastExcep;}**
...						
14:goto 20	in:	{U}	{U}	{LB;}	{LA;}	{LCastExcep;}
	out:	{U}	{U}	{LB;}	{LA;}	{LCastExcep;}
-17:*Exception*	in:	{U}	{U}	{LB;}	{U}	{U}
	out:	**{LExcep;}**	{U}	{LB;}	{U}	{U}
17:astore_2	in:	**{LExcep;}**	{U}	{LB;}	{U}	{U}
	out:	{U}	{U}	{LB;}	{U}	**{LExcep;}**
...						
19:astore_1	in:	{NULL}	{U}	{LB;}	{U}	{LExcep;}
	out:	{U}	{U}	{LB;}	{NULL}	{LExcep;}
20:aload_1	in:	{U}	{U}	{LB;}	{NULL, LA;, LB;}	**{LCastExcep;,LExcep;}**
	out:	{NULL,LA;,LB;}	{U}	{LB;}	{NULL, LA;, LB;}	{LCastExcep;,LExcep;}
12:areturn	in:	{NULL,LA;,LB;}	{U}	{LB;}	{NULL, LA;, LB;}	{LCastExcep;,LExcep;}
	out:	{U}	{U}	{LB;}	{NULL, LA;, LB;}	{LCastExcep;,LExcep;}

Figure 3.8 : Gestion des exceptions (analyse de flot de données)

3.4 Le cas des sous-routines

Nous avons précédemment mentionné les instructions `jsr` et `ret`, sans montrer comment les intégrer à l'algorithme d'analyse que nous avons présenté. Nous verrons ici que cela cause des problèmes majeurs, dont la considération nous oblige à remettre en question et à raffiner les modélisations et l'algorithme d'analyse de flot de données.

Il y a deux aspects à considérer pour en comprendre la portée : l'introduction d'un élément dynamique dans la détermination du flot de contrôle (la structure syntaxique) et le

polymorphisme (élément sémantique de l'analyse). Cette section en présente d'abord une description et ensuite des moyens appropriés afin de calculer un résultat précis lors de l'analyse.

3.4.1 Introduction à `jsr` et `ret`

Rappelons d'abord ce que sont les instructions `jsr` et `ret`. Dans la spécification officielle de la machine virtuelle [LinY99], ces instructions sont introduites pour compiler en codes-octets la construction du langage Java pour la gestion des exceptions : `try{...} - finally{...}`. La clause `finally{...}` est alors associée au concept de sous-routine. Mais le code-octet n'étant pas nécessairement produit par un compilateur Java standard, un programme peut faire une utilisation variée de ce puissant concept.

Il y a quatre règles structurelles que nous utilisons pour les instructions `jsr` et `ret`. Ces règles respectent les contraintes structurelles énoncées dans la spécification de la machine virtuelle [LinY99] au chapitre 4.8.2 et permettent aussi une flexibilité dans l'utilisation de ces instructions.

Nous désignons comme sous-routine une paire d'instructions composée de la cible d'un `jsr` et d'un `ret` et ayant les propriétés suivantes. Une sous-routine peut-être invoquée par une ou plusieurs instructions `jsr`. Une sous-routine doit avoir exactement une instruction `ret` associée: l'absence de `ret` fait que ce n'est pas une sous-routine (le `jsr` peut être considéré comme un simple branchement inconditionnel) et il ne peut y avoir plusieurs points de sortie explicites (`ret`) d'une sous-routine. Une instruction `ret` peut être associée à une seule sous-routine. Enfin, les sous-routines récursives ne sont pas permises.

Exemple

La figure 3.9 illustre le comportement de ces instructions, tel que décrit dans la spécification de la machine virtuelle [LinY99]. L'instruction `jsr 9` empile la valeur 4 qui est l'adresse relative de l'instruction suivante dans le tableau de code et effectue un branchement à l'adresse `9`. L'instruction `ret 0` doit effectuer un branchement en récupérant l'adresse dans la variable locale `0`. Cette adresse aura été copiée au préalable par l'instruction `astore 0`.

Instruction		Pile	Variable locale
		0	0
...			
1: jsr 9	in:	{U}	{U}
	out:	{4}	{U}
4: return	in:	{U}	{4}
	out:	{U}	{4}
...			
9: astore_0	in:	{4}	{U}
	out:	{U}	{4}
...			
15: ret 0	in:	{U}	{4}
	out:	{U}	{4}

Figure 3.9 : Trace d'analyse naïve de flot de données de sous-routine

3.4.2 Problématique

Voyons maintenant comment les sous-routines complexifient l'analyse de flot de données de programmes. Nous présenterons d'abord le fait que la structure syntaxique des sous-routines introduit un élément dynamique dans le flot de contrôle du programme et ensuite comment le polymorphisme en entrée de sous-routine peut rendre la sémantique de l'algorithme d'analyse présenté dans la section précédente trop imprécise.

3.4.2.1 Structure syntaxique

Nous avons mentionné que les codes-octets `jsr t` et `ret 1` font partie de la famille des instructions de branchement inconditionnel. Cela est évident pour `jsr` dont l'opérande encodé statiquement dans le tableau de code indique l'adresse relative `t` de branchement, qui est le début de la sous-routine. Mais les successeurs de l'instruction `ret` dépendent de la valeur de l'opérande stockée dans la variable locale `1`. La valeur de cet opérande sera calculée dynamiquement par la machine virtuelle lors de l'exécution réelle de l'instruction `jsr` et correspondra à l'adresse de la prochaine instruction suivant séquentiellement ce `jsr`. Cependant, lors d'une analyse qui nécessite l'évaluation de tous les chemins possibles dans le code d'une méthode, il nous faut identifier tous les points de retour éventuels d'une sous-routine. Ces points seront aussi nombreux qu'il y a de `jsr` se branchant à la sous-routine.

De plus, parce que les instructions d'une sous-routine ne forment pas nécessairement une séquence et que des branchements de toutes sortes peuvent survenir entre le moment où l'on entre dans une sous-routine et où on en sort, il est très difficile de déterminer si une instruction appartient à telle ou telle sous-routine, quelle est l'historique d'appels de sous-routines qui y a conduit et enfin de déterminer si une instruction `ret 1` cause le retour de plus d'une sous-routine imbriquée.

Afin de pouvoir traiter l'association entre le début d'une sous-routine et sa fin ainsi que les `jsr` invoquant cette sous-routine, de nouveaux types, de nouvelles structures de données et un algorithme adapté seront introduits un peu plus loin dans cette section.

Exemple

Les figures 3.10 et 3.11 illustrent la problématique du flot de contrôle dynamique. L'instruction « `0:jsr 4` » invoque la sous-routine 4 et l'instruction « `11:jsr 22` » invoque

la sous-routine 22. Plus loin, nous trouvons que l'instruction « `32:ret 2` » retourne de la sous-routine 22. L'instruction « `20:ret 1` » est associée à la sous-routine 4, mais trois chemins permettent de l'atteindre et cela rend impossible d'identifier de façon certaine l'historique d'appels de sous-routines à ce point du programme : est-ce <4> ou <4,22>?

```
 0: jsr 4      //appel sous-routine 4
 3: return
 4: astore 1   //sous-routine 4
 6: iload_0
 8: ifeq 17
11: jsr 22     //appel sous-routine 22
14: goto 20
17: goto 20
20: ret 1      //fin sous-routine 4 ou 22?
22: astore 2   //sous-routine 22
24: iload_0
26: ifeq 32
29: goto 20
32: ret 2      //fin sous-routine 22
```

Figure 3.10 : Sous-routines et flot de contrôle dynamique (code)

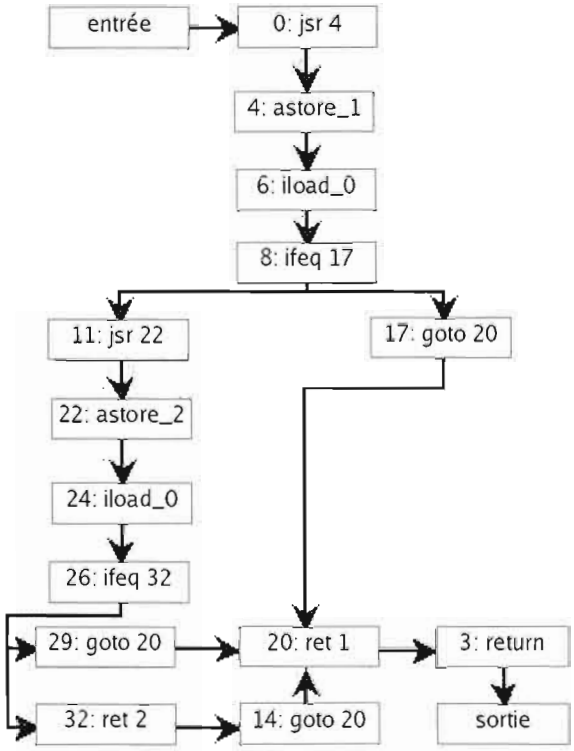

Figure 3.11 : Sous-routines et flot de contrôle dynamique (graphe)

3.4.2.2 Polymorphisme

L'autre aspect problématique avec les sous-routines est le polymorphisme et a trait à la sémantique de l'analyse de flot de données. Nous avons vu que les types en entrée d'un noeud sont égaux à l'union des types en sortie de ses prédécesseurs (section 3.2.5). Dans le cas d'une sous-routine `t`, les prédécesseurs sont tous les `jsr t` du graphe. L'instruction `ret l` retournant de cette sous-routine va propager ces types à tous ses successeurs. Il faudrait cependant faire attention de ne pas propager des types provenant d'un appelant p_1 au successeur d'un autre appelant p_2. Cela serait une évaluation trop grossière du comportement du programme car jamais lors de l'exécution le chemin de contrôle ne partira d'un point d'invocation pour ensuite retourner ailleurs dans le code. Une telle analyse peut dans

certains cas rendre un programme valide non vérifiable. Ce problème a été documenté par Stärk [StaS03].

Exemple

Les figures 3.12 et montrent un exemple de polymorphisme. Ce code comporte une sous-routine dont le début est à l'adresse 21 et la fin à l'adresse 28. Il y a deux `jsr 21`, dont les types en sortie pour la variable locale 1 sont respectivement `int` et `uninitialized`. L'union de ces types est propagée en entrée de sous-routine jusqu'au `ret` et à ses successeurs, les instructions 9 et 18. Nous nous retrouvons donc à l'instruction « `9:iinc 1,1` », indiquant d'incrémenter de 1 une valeur de type `int` dans la variable locale 1, à recevoir les types `{I,U}` en entrée. Cela nous porte à conclure que le programme est erroné, mais c'est l'analyse qui l'est car lors de l'exécution réelle du programme, seuls les chemins de 6 à 9 et de 15 à 18 sont possibles, pas les chemins 6 à 18 et 15 à 9. Ainsi, seul le type `int` atteint réellement la variable locale 1 pour l'instruction « `9:iinc 1,1` ».

Instruction		Pile	Variables locales		
		0	0	1	2
...					
6: jsr 21	in:	{U}	{I}	{I}	{U}
	out:	**{9}**	{I}	**{I}**	{U}
9: iinc 1, 1	in:	{U}	{I}	**{I, U}**	**{9, 18}**
	out:	{U}	{I}	{I}	{9, 18}
...					
15: jsr 21	in:	{U}	{I}	{U}	{U}
	out:	**{18}**	{I}	**{U}**	{U}
18: goto 30	in:	{U}	{I}	**{I, U}**	**{9, 18}**
	out:	{U}	{I}	{I, U}	{9, 18}
21: astore_2	in:	**{9, 18}**	{I}	**{I, U}**	{U}
	out:	{U}	{I}	{I, U}	{9, 18}
...					
28: ret 2	in:	{U}	{I}	{I, U}	{9, 18}
	out:	{U}	{I}	**{I, U}**	**{9, 18}**

Figure 3.12 : Sous-routines et polymorphisme

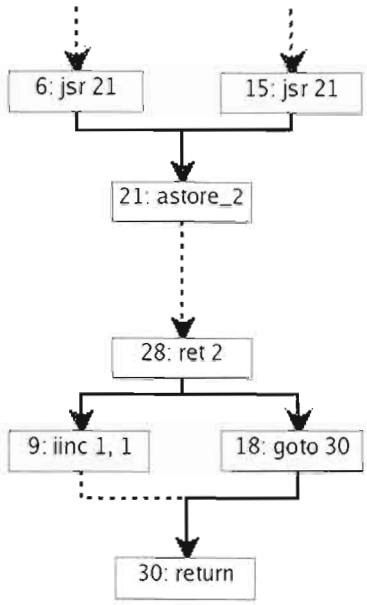

Figure 3.13 : Sous-routines (graphe associé à la figure 3.12)

3.4.3 Extension des types

Des types supplémentaires seront introduits pour effectuer l'analyse de flot de données de code comportant des sous-routines: `subroutineAddress(t)`, `contextualStackInfo(i)`, `contextualLocalInfo(i)` et `contextualInfo(i)`. Une adresse de sous-routine (`subroutineAddress(t)`) est générée par l'instruction `jsr t` qui l'empile sur la pile d'opérandes. Notons que `t` correspond à l'adresse du début de la sous-routine et non pas à l'adresse de l'instruction suivante dans le tableau de code. Les termes « information contextuelle » (« `contextual...Info(i)` ») désignent le contenu de la pile ou des variables locales, à l'indice `i`, dans le contexte de l'invocation de la sous-routine (instruction `jsr t`).

3.4.4 Extension de l'abstraction

Les blocs d'activation, notés $F = (S, R)$ ont été présentés à la section 3.2.2 comme une abstraction de l'environnement d'exécution de la machine virtuelle. Dans le cadre de l'analyse de flot de données, ils sont utilisés pour propager dans le graphe les ensembles de types pour les variables. Maintenant, à chaque bloc d'activation de base nous ajoutons un bloc d'activation pour chaque sous-routine de la méthode afin de propager les types contextuels à chaque sous-routine. Nous introduisons donc la notion d'*environnement*. Soit r, le nombre maximal de sous-routines de la méthode. Un *environnement* $E = (F_0, F_1, \ldots, F_r)$ est une séquence de blocs d'activation dont F_0 est le bloc d'activation de base et les blocs subséquents sont associés, séquentiellement, aux sous-routines. Tous les blocs d'activation ont le même nombre d'éléments. Les éléments *in* et *out* d'un noeud désigneront désormais des environnements :

$$n = (offset, instruction, pred, succ, in, out).$$

3.4.5 Extension du treillis

Les nouveaux types s'intègrent directement dans le treillis tel que présenté à la section 3.2.3.

3.4.6 Extension des fonctions de transition

Les fonctions de transition associées aux sous-routines sont *jsr, ret* et *jsrBis*. Nous les présentons ici, ainsi que la fonction associée à l'instruction `iconst i` comme exemple montrant comment les fonctions de transition présentées à la section 3.2.4 peuvent être directement étendues à la nouvelle abstraction. Ces fonctions sont inspirées de [SaVM06].

Rappelons qu'un bloc d'activation est noté $F = (S, R)$ avec S et R des séquences d'ensembles de types et qu'un environnement est noté $E = (F_0, F_1, \ldots, F_r)$, une séquence de blocs d'activation, et $|F_i| = |F_j| \; \forall \, i, j \in E$. Dans le but de simplifier la notation du formalisme, nous considérons un bloc d'activation comme étant une seule séquence d'ensembles de types résultant de la concaténation de la pile et des variables locales. De plus, nous considérons un environnement comme étant une seule séquence d'ensembles de types résultant de la concaténation des blocs d'activation qu'il contient.

Soient

$x = (x_0, ..., x_{n-1})$, $y = (y_0, ..., y_{n-1})$, des environnements,

I, l'ensemble des instructions du code d'une méthode,

$A = \{a \mid i \in I \wedge i = jsra\}$, l'ensemble des adresses de début de sous-routines dans une méthode,

$maxStack = |S|$, la hauteur maximale de la pile,

$maxLocals = |F|$, le nombre maximal de variables locales,

$maxSR = |A|$, le nombre maximal de sous-routines,

$m = maxStack + maxLocals$, le nombre d'emplacements dans un bloc d'activation,

$n = m \times (maxSR+1)$, le nombre d'emplacements dans un environnement,

i, un indice d'emplacement dans un environnement tel que $0 \leq i < n$,

h, la hauteur de la pile préalablement à l'évaluation d'une l'instruction,

l, l'indice d'une variable locale en opérande d'une instruction,

j et v, des indices d'emplacements dans un bloc d'activation tel que $0 \leq j, v < m$,

k, un indice de bloc d'activation tel que $0 \leq k \leq maxSR$,

p, l'indice d'un bloc d'activation associé à une sous-routine tel que $a_p \in A$,

A_p, l'ensemble des adresses de début de sous-routines présentes dans un bloc d'activation associé à une sous-routine, tel que $A_p \subseteq A$,

alors

- $iconst_h(x) = (y_0, ..., y_{n-1})$ où

$$y_i = \begin{cases} \{int\} & \text{si } i = h + km \ \wedge \ k \in \{0, ..., maxSR\} \\ x_i & \text{sinon} \end{cases}$$

- $jsr_{ha_p}(x) = (y_0, ..., y_{n-1})$ où

$$y_i = \begin{cases} \{a_p\} & \text{si } i = h + km \ \wedge \ k \in \{0, ..., maxSR\} \\ \{contextualInfo(v)\} & \text{si } i \in \{pm, ..., pm + (m-1)\} \wedge \\ & \quad i \neq h + pm \ \wedge \ v = i - pm \\ x_i & \text{sinon} \end{cases}$$

- $ret_{lA_p}(x) = (y_0, ..., y_{n-1})$ où

$$y_i = \begin{cases} x_i \cup \{u\} & \text{si } i = l + km \ \wedge \ k \in \{0, ..., maxSR\} \ \wedge \ k \neq p \\ x_i \cup \{u\} & \text{si } x_i \cap A_p \neq \emptyset \wedge k = p \\ x_i & \text{sinon} \end{cases}$$

- $jsrBis_{a_p}(x, y) = (z_0, ..., z_{n-1})$ où

pour tout $j \in \{0, ..., m-1\}$, $k \in \{0, ..., maxSR\}$,

$T_{aj} = \{v \mid contextualInfo(v) \in y_{j+pm}\}$, l'ensemble des indices désignés par des informations contextuelles à l'emplacement y_{j+pm},

$T_{bj} = \{t \mid t \neq contextualInfo(v), t \in y_{j+pm}\}$, l'ensemble des types autres que les informations contextuelles à l'emplacement y_{j+pm},

nous avons que

$$z_{j+km} = \begin{cases} T_{bj} \cup (\bigcup_{v \in T_{aj}} x_{v+km}) \cup \{u\} & \text{si } k = p \\ T_{bj} \cup (\bigcup_{v \in T_{aj}} x_{v+km}) & \text{si } k \neq p \end{cases}$$

La fonction associée à l'instruction `iconst i` empile un ensemble dont l'élément est le type `int` sur toutes les piles de l'environnement (pour tous ses blocs d'activation).

La fonction associée à l'instruction `jsr t` (a_p = `t`) initialise tous les emplacements du bloc d'activation à l'indice p de l'environnement, qui est associé à la sous-routine invoquée, avec le type `contextualInfo(v)`, où v est l'indice de l'emplacement dans le bloc d'activation, et empile l'adresse du début de cette sous-routine sur toutes les piles de l'environnement.

La fonction associée à l'instruction `ret 1` ajoute le type `uninitialized` dans la variable locale `1` pour tous les blocs d'activation de l'environnement. Aussi, une instruction *ret* pouvant causer le retour de sous-routines imbriquées, nous les identifions par l'ensemble des adresses de sous-routine présentes dans le bloc d'activation p de la sous-routine. Le type `uninitialized` sera donc aussi ajouté dans tout emplacement de l'environnement ayant comme élément une de ces adresses de sous-routines.

La fonction $jsrBis_{ap}(x,y)$ correspond à la transition entre un `ret` (variable y) et un de ses successeurs (variable z), soit l'instruction suivant un `jsr` invoquant cette sous-routine (variable x). Pour palier au polymorphisme, cette fonction propage les types de façon sélective. Tout emplacement z_i de l'environnement en sortie de la fonction sera calculé d'après le contenu de l'emplacement y_j correspondant dans le bloc d'activation p en sortie du `ret`. Si y_j comprend des types autres que `contextualInfo(v)`, ils seront propagés dans z_i. Si y_j comprend des types `contextualInfo(v)`, ce qui sera propagé est le contenu de l'emplacement x_v en entrée du `jsr`. Le type `uninitialized` sera ensuite ajouté à tous les emplacements du bloc d'activation p pour signifier le retour de la sous-routine.

Exemple

La figure 3.14 illustre le fonctionnement de ces fonctions. *F-0* et *F-1* désignent les deux blocs d'activation d'un environnement, celui de base et celui de la sous-routine 9. *S* et *R* désignent respectivement la pile et les variables locales.

L'instruction `jsr 9` empile l'adresse relative du début de la sous-routine, initialise les emplacements du bloc d'activation de la sous-routine et effectue un branchement à l'adresse `9`. L'instruction `ret 0` invalide l'adresse `9` dans tout l'environnement. La fonction *jsrBis* calcule les types en entrée du noeud « `4:return` » propageant le type `int` dans la

variable locale 1, type qui provient du contexte appelant de la sous-routine. Cette valeur a été récupérée car la fonction *jsrBis* observe la valeur `r1 (contextualLocalInfo(1))` dans la variable locale 1 du bloc d'activation contextuel à la sous-routine. Rappelons-nous que l'information contextuelle est semblable à un pointeur à un emplacement dans le bloc d'activation en entrée de l'instruction `jsr` invoquant cette sous-routine. Ensuite, le bloc d'activation de la sous-routine est invalidé par injection du type `uninitilized` à chaque emplacement.

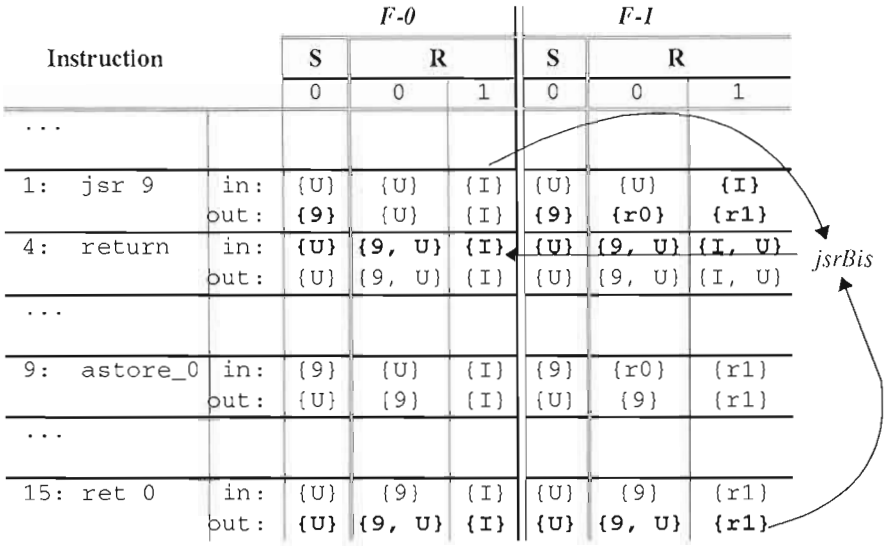

Figure 3.14 : Trace d'analyse de flot de données avec sous-routine (2)

3.4.7 Extension de l'algorithme

Voyons comment compléter l'algorithme de la section 3.2.5 en utilisant les nouveaux types, abstractions et fonctions de transition exposés précédemment afin de réaliser l'analyse de flot de données de méthodes avec des sous-routines.

L'emphase est mise sur ces nouveaux aspects : le calcul des fonctions de transition se fait désormais sur des *environnements*, donc les types sont propagés parallèlement dans tous les blocs d'activation d'un noeud, et le calcul du flot de contrôle dû aux instructions `jsr` et `ret` nécessite une attention particulière.

Initialement, le nombre de blocs d'activation des environnements correspond au nombre de sous-routines dans la méthode plus un. Cela peut être calculé en itérant une fois sur le tableau de code et en comptant les cibles distinctes de branchement des instructions `jsr t`.

Ensuite, les types des variables atteignant le noeud d'*entrée* du graphe (*entrée.in*) pour le bloc d'activation de base sont déterminés par le descripteur de la méthode, tel que présenté à la section 3.2.5. Les autres blocs d'activation sont initialisés avec le type `uninitialized`. Le noeud *entrée* est ensuite inséré dans le graphe et dans la *worklist*.

Pour effectuer le calcul de l'analyse, répéter itérativement le processus suivant tant que la *worklist* n'est pas vide. Récupérer un noeud *n* de la *worklist*. Calculer la fonction de transition associée à l'instruction de ce noeud, prenant *n.in* en entrée et affectant le résultat à *n.out* (sauf pour *jsrBis*). Rappelons que les types sont propagés à tous les blocs d'activation de l'environnement. Si la fonction a modifié l'état des types en sortie, insérer les successeurs de *n* dans la *worklist*. Les successeurs sont identifiés seulement lors de la première visite de chaque noeud (sauf pour `jsr` et `ret`) en suivant les règles mentionnées dans la section 3.1.3 et sont alors insérés dans le graphe. La *worklist* étant un ensemble, on n'y insérera pas un noeud s'il s'y trouve déjà. Enfin, s'il s'agit de la première visite du noeud, marquer le noeud *n* comme étant visité et recommencer le processus.

Concernant les instructions `jsr` et `ret`, leurs fonctions de transition sont déjà connues et il reste à calculer les successeurs. À cette fin, une table des sous-routines est utilisée pour

mémoriser des informations concernant chaque sous-routine (voir la figure 3.15). On y inscrit l'adresse de début d'une sous-routine, la liste des adresses des `jsr` l'invoquant et une référence au noeud de l'instruction `ret` qui en retourne.

Lors de la visite d'un noeud `jsr t`, son successeur immédiat est l'instruction à l'adresse `t`. Si la table des sous-routines ne contient pas d'enregistrement associé à la sous-routine `t`, ajouter une entrée dans la table avec l'adresse de début `t` et l'adresse du `jsr` dans la liste des `jsr` se branchant à cette sous-routine. Le noeud `ret` associé est encore inconnu et la visite est terminée.

Si la table contient déjà un enregistrement pour la sous-routine `t`, on ajoute l'adresse du `jsr` dans la liste des `jsr`. On vérifie si le noeud `ret` a déjà été identifié. Si ce n'est pas le cas, l'opération est terminée.

Si le `ret` est connu, l'instruction suivant le `jsr t` dans le tableau de code sera visité en tant que noeud successeur spécial (un point de retour de la sous-routine) du `jsr t`. On l'insère donc dans le graphe et dans la *worklist* et on crée des arcs dans le graphe entre le `ret` et le `jsr` et entre le `jsr` et l'instruction suivante.

Lors de la visite d'un noeud représentant une instruction `ret l`, on examine la variable locale `l` dans le bloc d'activation de base. Si elle contient une seule adresse, elle désigne la sous-routine retournée. On inscrira alors le noeud du `ret` à l'entrée correspondante dans la table des sous-routines. Les successeurs du noeud `ret` sont toutes les instructions `jsr t` faisant partie de la liste des `jsr` invoquant cette sous-routine. Ils sont reliés dans le graphe de la façon expliquée au paragraphe précédent. On examine ensuite la variable locale `l` dans le bloc d'activation associé à cette sous-routine. S'il y a un autre type

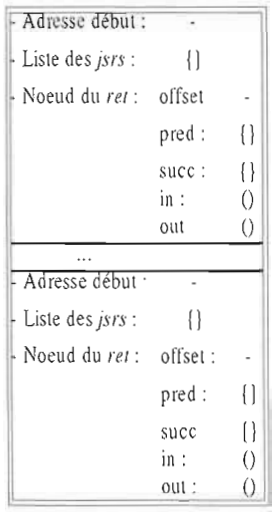

Figure 3.15 : Table des sous-routines

que l'adresse de retour, il y a erreur de vérification car cela ne respecterait pas les règles syntaxiques d'une sous-routine.

L'arrêt de l'algorithme d'analyse de flot de données pour cause d'erreur de vérification est justifiable d'une part par le fait qu'il s'agit d'une erreur structurelle (le graphe de contrôle ne peut être complètement construit). D'autre part, les fonctions de transitions étant monotones, si une donnée erronée est trouvée en cours d'analyse, elle sera préservée jusqu'à la fin de l'algorithme et l'erreur serait donc aussi détectée plus tard.

Les divers cas possibles sont :

Contenu de variable locale 1	Explication
{5}	Retour de la sous-routine 5.
{U}	Erreur : sous-routine non-identifiable.
{5, U}	Erreur : soit c'est un retour de la sous-routine 5, soit on ne peut identifier de sous-routine.
{5, 10}	Erreur : retour de deux sous-routines différentes.
{5, contextualStackInfo(2)}	Erreur : Retour de la sous-routine 5 mais on ne sait ce que désigne le type contextuel.
{contextualLocalInfo(1)}	Erreur : on ne sait ce que désigne le type contextuel.

Lorsqu'un type autre qu'une adresse de sous-routine est propagé au `ret 1` dans la variable locale `1`, c'est qu'il existe un chemin qui atteint le `ret` sans passer par un `jsr` ou qu'une instruction entre le début et la fin de la sous-routine a généré ce type. Aussi, dans le bloc d'activation associé à la sous-routine, lorsqu'un type `contextualInfo` atteint le `ret` nous savons qu'il désigne un ensemble de types dans un contexte préalablement à l'appel de la sous-routine. Même si cet ensemble contient seulement l'adresse de cette même sous-routine comme élément, elle aurait été générée par un appel antérieur et la récursion sur les sous-routines n'est pas permise.

Exemple

Reprenons l'exemple de la figure 3.12 pour illustrer cet algorithme. La figure 3.16 montre le résultat de l'analyse de flot de données de ce programme qui montrait le problème du polymorphisme dans les sous-routines.

Les instructions `jsr 21` aux adresses 6 et 15 initialisent les blocs d'activation de la sous-routine 21 dans leurs environnements en sortie, y empilent l'adresse de la sous-routine (21) et mettent à jour la table de sous-routine. L'instruction « `21:astore_2` » copie l'adresse de la sous-routine de la pile à la variable locale 2. L'instruction « `28:ret 2` » récupère l'adresse de la sous-routine 21 dans la variable locale 2, met à jour la table de sous-routine et calcule la fonction de transition *jsrBis* pour propager les types aux instructions aux adresses 9 et 18. Le contenu de la variable locale 1 du bloc d'activation de la sous-routine 21 est `{r1,I}` et indique de propager le type `int` en union avec le contenu de la variable locale 1 dans le contexte respectif des `jsr` appelant. Pour le `jsr` à l'adresse 6 cet ensemble est `{I}` et pour le `jsr` à l'adresse 15 c'est `{U}`.

Instruction		*F*-0				*F*-1			
		S	R			S	R		
		0	0	1	2	0	0	1	2
...									
6: jsr 21	in:	{U}	{I}	{I}	{U}	{U}	{I}	{I}	{U}
	out:	**{21}**	{I}	**{I}**	{U}	**{21}**	**{r0}**	**{r1}**	**{r2}**
9: iinc 1, 1	in:	{U}	{I}	**{I}**	**{21, U}**	**{U}**	**{I, U}**	**{I, U}**	**{21, U}**
	out:	{U}	{I}	{I}	{21, U}	{U}	{I, U}	{I, U}	{21, U}
...									
15: jsr 21	in:	{U}	{I}	{U}	{U}	{U}	{I}	{U}	{U}
	out:	**{21}**	{I}	**{U}**	{U}	**{21}**	**{r0}**	**{r1}**	**{r2}**
18: goto 30	in:	{U}	{I}	**{I, U}**	**{21, U}**	**{U}**	**{I, U}**	**{I, U}**	**{21, U}**
	out:	**{U}**	{I}	{I, U}	{21, U}	{U}	{I, U}	{I, U}	{21, U}
21: astore_2	in:	**{21}**	{I}	**{I, U}**	{U}	**{21}**	{r0}	**{r1}**	{r2}
	out:	{U}	{I}	{I, U}	{21}	{U}	{r0}	{r1}	**{21}**
...									
28: ret 2	in:	{U}	{I}	{I, U}	{21}	{U}	{r0}	{r1, I}	{21}
	out:	{U}	{I}	**{I, U}**	**{21, U}**	{U}	{r0}	**{r1, I}**	**{21, U}**
30: return	in:	{U}	{I}	{I, U}	{21, U}	{U}	{I, U}	{I, U}	{21, U}
	out:	{U}	{I}	{I, U}	{21, U}	{U}	{I, U}	{I, U}	{21, U}

Figure 3.16 : Sous-routines et polymorphisme (2)

Adresse début :	21
Liste des *jsr* :	{6, 15}
Noeud du *ret* :	28

Figure 3.17 : Table de sous-routine pour la figure 3.16

3.5 Vérification

Après avoir calculé l'analyse de flot de données, nous avons de l'information précise sur l'état de la pile et des variables locales préalablement à l'exécution de chaque instruction du code d'une méthode. Nous présentons dans cette section un algorithme vérifiant que le code respecte les contraintes structurelles (voir la spécification de la machine virtuelle [LinY99] à la section 4.8.2).

3.5.1 Algorithme général

L'algorithme reçoit en entrée le graphe de contrôle dont les noeuds comportent l'état des types dans les blocs d'activation en entrée et en sortie ainsi que la table des sous-routines. Le calcul effectué est une variante de l'algorithme de la section 3.4.7. Utilisant une *worklist*, chaque noeud du graphe sera itérativement visité une seule fois. La fonction de transition associée à l'instruction du noeud sera calculée en prenant pour entrée l'état des types de l'environnement tel que calculé par l'analyse de flot de données et mémorisé dans *n.in*. Mais ce calcul sera effectué comme une simulation défensive de l'instruction en validant que seuls les types permis soient manipulés.

À titre d'exemple, la figure 3.18 montre la vérification de l'information calculée par l'analyse de flot de données pour l'instruction `astore_0`. Cette instruction doit dépiler une référence et la copier dans la variable locale 0. L'analyse de flot de données a modélisé ceci

sans examiner les types. L'algorithme de vérification constate que le type en entrée sur la pile est `int` et lance une erreur de vérification.

La vérification procède de façon assez intuitive en calculant une exécution abstraite des instructions tout en validant que la manipulation des types soit légale (voir la spécification de la machine virtuelle [LinY99] au chapitre 6 pour une description complète).

Instruction		Pile		Variables locales	
		0	1	0	1
...					
10: aload 0	in:	{I}	{I}	{U}	{U}
	out:	{I}	{U}	{U}	{I}
...					

Figure 3.18 : Vérification de astore_0 basée sur l'analyse de flot de données

3.5.2 Compléments

Certains aspects de vérification des contraintes structurelles ne sont pas modélisés par les fonctions de transition. Nous présentons ici les aspects les plus problématiques : les contraintes de sous-type, certains cas de branchements en présence du type d'instance non initialisée et les constructeurs.

Les contraintes de sous-type concernent la vérification de la compatibilité des opérandes de type référence de certaines instructions. Rappelons qu'une référence désigne une classe, une interface, un tableau ou la référence `null`. Deux références A et B sont compatibles si $B \leq A$, c'est-à-dire si B est le même type ou un sous-type de A. Si A et B sont des classes, $B \leq A$ implique que B est égal à A ou que B hérite de A. Si ce sont des tableaux, B doit être de dimension plus petite ou égale à A et le type de ses éléments être compatible

avec ceux de *A*. L'algorithme complet d'évaluation de la compatibilité entre deux types est présenté dans la spécification de la machine virtuelle [LinY99] à la section 2.6.7. Voici une description des instructions générant des contraintes de sous-type.

L'instruction `areturn` termine la méthode en retournant une référence. Elle doit avoir en opérande sur la pile une référence compatible avec le type de retour de la méthode courante tel qu'indiqué dans son descripteur. Par exemple, l'instruction `areturn` `LB;` dans la méthode `test()LA;` implique que $B \leq A$.

L'instruction `athrow` doit avoir en opérande sur la pile une instance initialisée compatible avec la classe `java.lang.Throwable`. La même chose pour le type (`catch_type`) des gestionnaires d'exception (`exception_handlers`).

L'instruction `getfield index` a pour fonction de récupérer un champ d'un objet. L'`index` doit faire référence à une `constant_fieldref_info` dans le `constant_pool` de la classe courante qui donne le nom et le descripteur du champ ainsi que le nom de l'objet (*C*) le déclarant. L'opérande sur la pile d'un `getfield` doit être une référence compatible avec la référence *C* désignée dans le `constant_pool`.

De la même manière, les instructions `putfield index` et `putstatic index` doivent avoir en opérande sur la pile une référence compatible avec le type du champ *f* indiqué dans le `constant_pool`. Aussi, pour `putfield`, l'opérande de l'instruction sur la pile doit être une référence compatible avec la référence *C* déclarée dans le `constant_pool`.

Les instructions d'invocation de méthode sont `invokeinterface`, `invokespecial`, `invokestatic` et `invokevirtual`. Elles ont un comportement similaire entre elles. Elles ont en opérande dans le tableau de code une référence à une

entrée du `constant_pool` de la classe courante donnant le descripteur de la méthode à invoquer et la classe la déclarant. Les opérandes sur la pile doivent être compatibles avec les arguments de la méthode invoquée, tel que défini dans le descripteur. De plus, l'instance de la classe en opérande sur la pile (de laquelle la méthode doit dynamiquement être invoquée) doit être compatible avec le type spécifié dans le `constant_pool`.

La figure 3.19 illustre l'instruction `invokevirtual 22`. Le premier opérande sur la pile correspond au dernier paramètre indiqué par le descripteur dans le `constant_pool`. La référence *B* doit être compatible avec le deuxième paramètre de la méthode, la référence *A*. Le premier paramètre de la méthode est le type `int`. La référence *D*, de laquelle la méthode sera invoquée, doit être compatible avec la référence *C* déclarant la méthode.

Tout branchement vers l'arrière dans le tableau de code offre comme possibilité de réutiliser des types produits lors d'une visite antérieure des instructions, tel dans le corps d'une boucle. Une instance non initialisée produite lors d'une itération pourrait ainsi plus tard être considérée comme ayant été initialisée par une invocation de constructeur sur une autre instance non initialisée du même type. Ainsi, lors des branchements arrière, des invocations et retours de sous-routines et lors d'une entrée dans un `exception_handler`, nous devons vérifier qu'aucune référence non initialisée ne se trouve parmi les types en entrée du noeud (*n.in*).

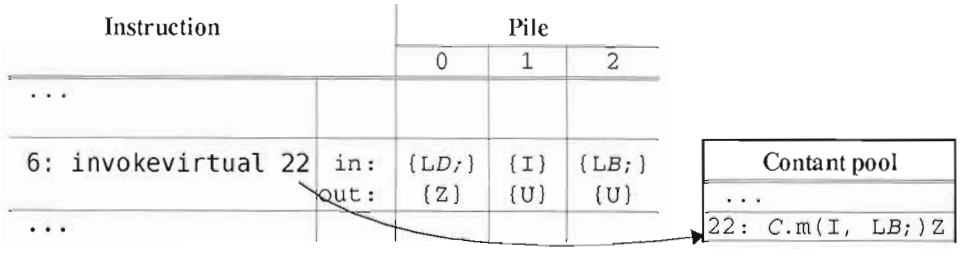

Figure 3.19 : Vérification de compatibilité des sous-types (invokevirtual)

Les constructeurs doivent initialiser une instance de classe. Une instance est initialisée par l'invocation d'un autre constructeur de la classe courante ou de la super-classe (sauf dans le cas de la classe `java.lang.Object` qui n'a pas de super-classe). Lors de la vérification d'un constructeur (méthode nommée `<init>`) nous devrons donc initialiser la variable locale 0 par le type spécial `uninitThis`. Nous devrons vérifier lors de la visite de chaque noeud du graphe que si le type `uninitThis` est présent dans l'environnement en entrée, il le sera aussi en sortie, sauf pour les instructions `invokespecial` *C*`.<init>` où *C* est égal à `uninitThis` ou à sa super-classe. Tel que mentionné à la page précédente, des références non initialisées ne doivent ici non plus être présentes dans l'environnement lors de branchements arrière, d'invocations et retours de sous-routines et lors d'une entrée dans un `exception_handler`.

Enfin, pour le noeud final représentant l'instruction `return`, on doit s'assurer que l'instance de la classe courante a été initialisée par le constructeur en vérifiant que le type `uninitThis` n'est pas présent dans l'environnement en entrée du noeud.

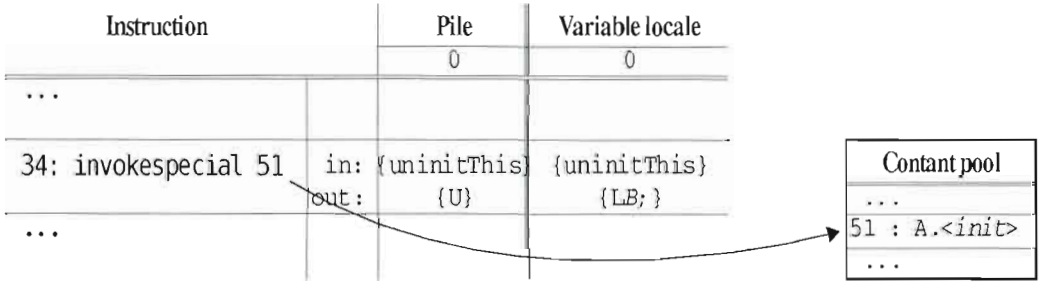

Figure 3.20 : Vérification de compatibilité des sous-types (invokespecial)

La figure 3.20 montre l'invocation d'une méthode d'initialisation de la classe courante dans le cadre d'un constructeur. L'instruction `invokespecial 51` indique le descripteur de la méthode dans le `constant_pool` et cela implique que la classe courante soit compatible avec la référence *A*.

3.6 Résumé

Nous avons présenté de façon formelle dans ce chapitre des structures de données et un algorithme d'analyse de flot de données permettant d'abstraire tous les chemins d'exécution possibles dans le code d'une méthode et d'estimer de manière conservatrice le type de chaque variable avant l'exécution des instructions qui la composent. Cette information est utilisée pour vérifier que les instructions respectent les contraintes structurelles du code-octet.

L'approche présentée a comme avantage de visiter chaque noeud du graphe une seule fois et de ne pas inclure le code mort dans le graphe. Cet algorithme pourrait être amélioré en associant des blocs de base à chaque noeud du graphe, ce qui permettrait d'en diminuer le nombre, tel que suggéré par Muchnick [Much97] p.232, 233 et Appel [AppP02] p.361 ou mieux, en ne conservant dans le graphe que les noeuds ayant plus d'un prédécesseur, comme nous le verrons dans le prochain chapitre.

CHAPITRE IV

VÉRIFICATION EN DEUX TEMPS

Dans le chapitre 3 nous avons suggéré d'utiliser une analyse de flot de données pour effectuer la vérification du code-octet. Compte tenu de l'implication du processus de vérification pour une machine virtuelle, il y a lieu de considérer des moyens de l'optimiser.

Ce chapitre examine une technique effectuant la vérification en deux phases. La première phase utilise une analyse de flux de données et fait l'ajout d'un certificat à une classe. Ce calcul peut être effectué pendant ou après la compilation de la classe, mais avant sa distribution, ce qui a pour effet d'alléger la deuxième phase de vérification. Cette phase, qui a lieu lors du chargement de la classe par la machine virtuelle, effectuera donc la vérification en utilisant ce certificat dans un processus linéairement proportionnel à la taille du code d'une méthode.

Il faut distinguer cette technique de vérification des certificats électroniques utilisés en sécurité informatique pour l'authentification d'entités (données, machines, personnes, ...). Cette dernière faisant en général appel à la cryptographie ne permet cependant pas de garantir que les données (en l'occurrence, les fichiers *class*), sont exempts d'anomalies et que le code-octet respecte les contraintes statiques et dynamiques. Les certificats proposés dans ce chapitre sont donc un outil simplifiant le processus de vérification du code-octet exposé au chapitre 3.

Nous proposons dans ce chapitre un format de certificat et un algorithme pour le calculer ainsi qu'un algorithme utilisant ce certificat pour vérifier le code de méthodes. Cette technique couvre l'intégralité des instructions en code-octet. Enfin nous justifions l'utilisation de cette technique en montrant sa validité.

4.1 Structure du certificat

Cette section montre comment structurer un certificat. Le format de fichier *class* prévoit un mécanisme simple pour ajouter de l'information à une classe : les attributs. Nous introduisons deux attributs : `VerificationConstantPool_attribute` et `TypeStateMapTable_attribute`.

4.1.1 Table de constantes de vérification

Nous avons vu au chapitre 3 que l'analyse de flot de données produit des ensembles de types. Dans le même sens, la table de constantes de vérification représente des ensembles de types qui serviront à vérifier les méthodes de la classe. Cette table est encodée dans l'attribut `VerificationConstantPool_attribute` de la classe (structure `ClassFile`).

```
VerificationConstantPool_attribute {
     u2 attribute_name_index;
     u4 attribute_length;
     u2 verification_constant_pool_count;
     cp_info verification_constant_pool[
               verification_constant_pool_count];
}
```

L'item `attribute_name_index` représente un index dans le `constant_pool` de la classe courante qui doit être de type `CONSTANT_Utf8_info` et avoir la valeur « `VerificationConstantPool` ». La valeur de l'item `attribute_length` est la longueur de l'attribut en octets moins les six premiers octets. La valeur de `verification_constant_pool_count` indique le nombre d'éléments dans la table de constantes de vérification qui est indicée de 0 à `verification_constant_pool_count - 1`. La table de constantes de vérification (`verification_constant_pool`) est conçue de façon similaire à la table de constantes (`constant_pool`) d'un fichier *class*. Elle peut comporter des éléments de deux types : `CONSTANT_Utf8_info` et `CONSTANT_TypeSet_info` qui sont identifiés par l'item `tag`.

```
CONSTANT_Utf8_info {
    u1 tag = 1;
    u2 length;
    u1 bytes[lentgh];
}
```

Tel que présenté dans la spécification de la machine virtuelle Java à la section 4.4.7 [LinY99], une `CONSTANT_Utf8_info` représente une chaîne de caractères. L'item `length` indique le nombre d'octets requis pour représenter cette chaîne de caractères encodés par le tableau `bytes`.

```
CONSTANT_TypeSet_info {
    u1 tag = 22;
    u2 element_count;
    verification_type type_set[element_count];
}
```

Pour une constante `CONSTANT_TypeSet_info`, l'item `element_count` indique le nombre d'éléments de type `verification_type` dans l'ensemble `type_set`. Ces types de vérification correspondent à ceux utilisés par l'analyse de flot de données exposée dans le chapitre 3. Les types les plus représentatifs sont ici présentés et une liste complète se trouve en Appendice A.

4.1.1.1 Types de vérification

Les types de primitifs sont représentés par des structures simples ne comportant qu'un item `tag` pour les identifier. Ainsi, le type `int` est représenté par la structure :

```
Verification_Type_Int {
     u1 tag = 1;
}
```

Les types références, en plus de l'item `tag`, ont un item `name_index` qui désigne un indice dans le `constant_pool` de la classe courante. Cet élément doit être de type `CONSTANT_Utf8_info` et représente le nom de la classe ou de l'interface.

```
Verification_Type_Reference {
     u1 tag = 16;
     u2 name_index;
}
```

Comme nous le verrons lors de la présentation de l'algorithme de calcul du certificat, certaines références utilisées par l'analyse de flot de données ne sont pas présentes individuellement dans le `constant_pool` en tant que `CONSTANT_Utf8_info`. Ces

références supplémentaires sont ajoutées dans le `verification_constant_pool` et l'item `name_index` en désigne alors l'indice.

```
Verification_Type_Extra_Reference {
    u1 tag = 17;
    u2 name_index;
}
```

Les types références désignant des tableaux sont représentés par la structure `Verification_Type_Array`. Elle comporte l'item `tag` l'identifiant de même que le nombre de dimensions et le type de base du tableau. Le type du tableau est une structure `verification_type` et peut représenter un type primitif ou une référence.

Par exemple, le tableau `int[][]` est représenté par la structure `Verification_Type_Array{2,Verification_Type_Int}`.

```
Verification_Type_Array {
    u1 tag = 18;
    u1 dimensions;
    verification_type base_type;
}
```

Les informations contextuelles utilisées pour la vérification des sous-routines sont représentées par les structures `Verification_Type_Contextual_Stack_Entry` et `Verification_Type_Contextual_Local_Entry`. L'item `tag` les différencie et la valeur de l'item `entry_index` désigne un emplacement dans la pile ou les variables locales.

```
Verification_Type_Contextual_Stack_Entry {
     u1 tag = 21;
     u4 entry_index;
}
```

4.1.2 Table des environnements de blocs d'activation

L'analyse de flot de données présentée au chapitre 3 calcule, pour chaque instruction du code, les ensembles de types les atteignant. Cela est représenté par les emplacements des blocs d'activation des environnements qui sont en entrée des noeuds associés. Cette information peut être inscrite dans un fichier *class* avec l'attribut `TypeStateMapTable_attribute` au niveau de la méthode (structure `method_info`).

```
TypeStateMapTable_attribute {
     u2 attribute_name_index;
     u4 attribute_length;
     u2 subroutine_address_count;
     u4 subroutine_addresses[subroutine_address_count];
     u2 type_state_map_count;
     TypeStateMap type_state_maps[type_state_map_count];
}
```

L'item `attribute_name_index` représente un indice dans le `constant_pool` de la classe courante devant être de type `CONSTANT_Utf8_info` et avoir la valeur « `TypeStateMapTable` ». La valeur de l'item `attribute_length` est la longueur de l'attribut en octets moins les six premiers octets. L'item `subroutine_address_count` désigne le nombre de sous-routines dans la méthode. L'item `subroutine_addresses` est un tableau des adresses de sous-routines de la méthode. L'item `type_state_map_count`

indique le nombre d'entrées dans la table `type_state_maps`, qui est un tableau de structures `TypeStateMap`.

La structure `TypeStateMap` représente les ensembles de types pour les emplacements d'un *environnement* de blocs d'activation avant l'exécution de l'instruction à l'adresse relative désignée par l'item `offset`. L'item `type_state_frame_count` correspond au nombre de blocs d'activation et détermine la taille du tableau de structures `TypeStateFrame`.

```
TypeStateMap {
    u4 offset;
    u4 type_state_frame_count;
    TypeStateFrame type_state_frames[type_state_frame_count];
}
```

La structure `TypeStateFrame` représente un bloc d'activation. La pile et les variables locales sont encodées comme des tableaux d'entiers. Ces valeurs doivent représenter des indices de constantes `CONSTANT_TypeSet_info` dans la table de constantes de vérification (`verification_constant_pool`).

```
TypeStateFrame {
    u4 number_of_stack_entries;
    u2 stack_entries_indexes[number_of_stack_entries];
    u4 number_of_locals_entries;
    u2 locals_entries_indexes[number_of_locals_entries];
}
```

4.1.3 Exemple

Voyons un exemple. La figure 4.1 illustre la disposition de ces attributs dans un fichier *class* ainsi que les relations entre leurs composantes. Nous voyons la classe `Exemple`, avec sa méthode `m()`. Si nous supposons que l'instruction « `16:astore_0` » a plusieurs prédécesseurs, il y a donc une entrée de type `TypeStateMap (TSM)` correspondante dans l'attribut `TypeStateMapTable (TSMT)`. Cette entrée indique qu'il y a un élément sur la pile dont la valeur est décrite par la constante `9` dans la table de constantes de vérification `(VCP)`. Il s'agit d'un ensemble de types comprenant deux éléments : une référence dont le nom est dans la table de constantes `(CP)` à l'indice `22` et une autre dont le nom a été ajouté dans la table de constantes de vérification à l'indice `15`. Il est aussi indiqué qu'il y a une variable locale dont la valeur est décrite par la constante `12` dans la table de constantes de vérification : le type `uninitialized`. Nous voyons aussi que les deux dernières constantes de la table de constantes donnent les noms des deux nouveaux attributs.

La figure 4.2 montre une représentation du contenu du certificat de la figure 4.1 en utilisant les structures pour l'analyse de flot de données du chapitre 3.

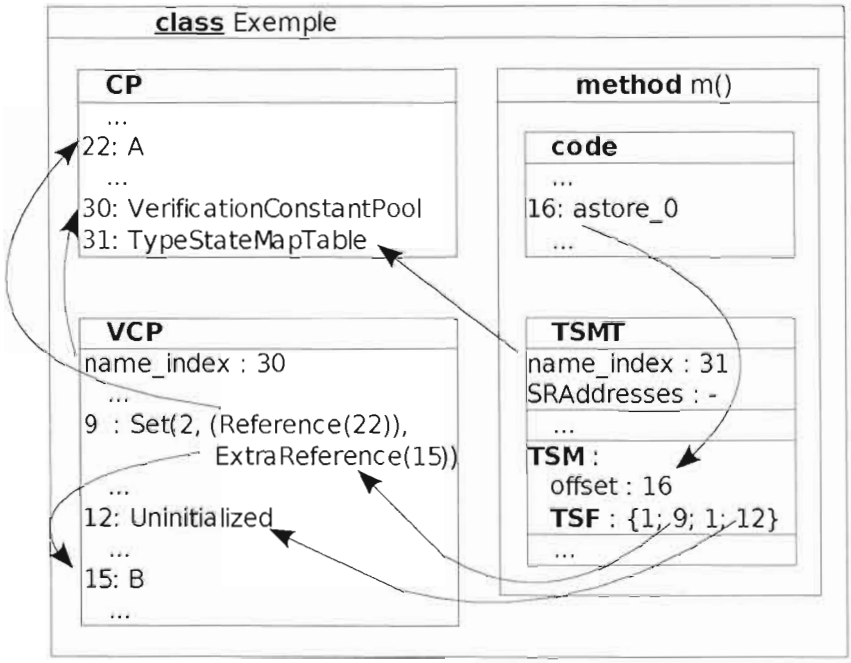

Figure 4.1 : Attributs pour la vérification en deux temps

Instruction		Pile	Variable locale
		0	0
...			
16: astore_0	in:	{LA;, LB;}	{U}
...			

Figure 4.2 : Attributs versus analyse de flot de données

4.2 Calcul du certificat

Dans cette section nous présentons un algorithme de calcul du certificat. Ce processus effectue aussi l'ajout d'attributs à une classe existante. Cette opération peut avoir lieu lors de la compilation, ou ultérieurement. Cependant, cela doit se faire avant la distribution des classes à l'environnement d'exécution.

L'attribut de classe `VerificationConstantPool_attribute` représente des ensembles de types. Comme nous le verrons, cette information est calculée conjointement avec le calcul de l'attribut `TypeStateMapTable_attribute`, de façon incrémentale.

L'attribut de méthode (structure `method_info`) `TypeStateMapTable_attribute` est une table associant à une adresse relative des séquences d'indices référant à la table de constantes de vérification. Plus précisément, cette information représente tous les types pouvant atteindre les variables avant l'exécution de l'instruction à l'adresse donnée, les variables étant les emplacements d'un environnement de blocs d'activation tel que décrit au chapitre 3.

Pour calculer le certificat, un attribut `TypeStateMapTable_attribute` sera calculé pour chaque méthode non abstraite et non native[1]. Pour ce faire, calculer premièrement l'analyse de flot de données d'une méthode tel qu'exposé au chapitre 3. Comme mentionné précédemment, l'information qui nous intéresse sont les types en entrée des noeuds (*n.in*). Ces données sont ensuite légèrement simplifiées en remplaçant tous les ensembles de types qui comportent l'élément `uninitialized` par l'ensemble comprenant exclusivement ce type. Ceci est justifié par le fait que si parmi tous les types pouvant atteindre une variable avant l'exécution d'une instruction se trouve le type `uninitialized`, cela signifie que soit cette variable n'a pas été initialisée ou soit son contenu a été invalidé sur un des chemins menant à cette instruction. L'usage de cette variable est donc proscrit. Le fait de conserver seulement le type `uninitialized` permet de simplifier l'ensemble de types et aussi de réduire le nombre total d'ensembles distincts dans le graphe.

---

1 Les méthodes abstraites n'ont pas de code et les méthodes natives ont du code dans un langage autre que le code-octet et qui n'est pas encodé dans le fichier *class*.

Deuxièmement, les adresses de début des sous-routines de la méthode sont ajoutées à l'item `subroutine_addresses` de l'attribut. Rappelons-nous qu'il y a autant de blocs d'activation dans les environnements des noeuds du graphe qu'il y a de sous-routines.

Troisièmement, des structures `TypeStateMap` sont créées et ajoutées à l'item `type_state_maps` de l'attribut. Ces structures sont associées à chaque noeud du graphe ayant plus d'un prédécesseur. Ce critère permet de réduire le nombre d'entrées dans l'attribut tout en conservant l'information nécessaire à la vérification. Nous y reviendrons dans la section 4.4. L'item `offset` d'une structure `TypeStateMap` aura pour valeur l'adresse relative associée à l'instruction du noeud. L'item `type_state_frames` est une séquence de structures `TypeStateFrame`. Chaque élément de cette séquence est créé en convertissant les ensembles de types pour chaque emplacement des blocs d'activation de l'environnement en entrée d'un noeud (*n.in*) en indices. Ces indices doivent correspondre à des structures de type `CONSTANT_TypeSet_info` dans la table de constantes de vérification (`verification_constant_pool`) de l'attribut `VerificationConstantPool_attribute`.

La table de constantes de vérification est remplie progressivement durant le calcul de l'algorithme. Au départ, elle est vide. Ensuite, si un ensemble de types à encoder dans le certificat n'est pas présent dans la table, il est converti et est ajouté à la fin de celle-ci. L'indice associé est ensuite utilisé dans la structure `TypeStateFrame`.

La conversion d'un ensemble de types produit par l'analyse de flot de données en structure `CONSTANT_TypeSet_info` est relativement directe, avec quelques exceptions. Les principaux cas sont expliqués dans ce qui suit, les autres conversions procèdent de façon similaire. Chaque type d'un ensemble est converti en une structure `verification_type` et ajouté à la séquence formant l'item `type_set` d'une `CONSTANT_TypeSet_info`.

Les types primitifs sont convertis directement en leur équivalent. Par exemple, `int` en `Verification_Type_Int`, `double` en `Verification_Type_Double` et `double2` en `Verification_Type_Double2`.

Les références, initialisées ou non, sont traitées de la même manière. Si le nom de la référence (sous forme de `CONSTANT_Utf8_info`) est présent dans la table de constantes (`constant_pool`) de la classe, une structure `Verification_Type_Reference` (ou `Verification_Type_Uninitref`) est créée avec l'indice du nom comme valeur de l'item `name_index`.

Mais l'analyse de flot de données génère certains types qui ne sont pas inscrits individuellement dans la table de constantes. Pensons aux arguments et au type de retour de la méthode qui sont encodés dans le descripteur de la méthode et au type de base d'un tableau. Donc, si le nom de la référence n'est pas trouvé dans le `constant_pool`, une structure `CONSTANT_Utf8_info` représentant ce nom est d'abord créée et ajoutée à la fin de la table de constantes de vérification (`verification_constant_pool`). Ensuite, une structure `Verification_Type_Extra_Reference` (alternativement `Verification_Type_Extra_Uninitref`) est créée et son item `name_index` aura comme valeur l'indice du nom dans la table de constantes de vérification.

Les tableaux sont convertis en une structure `Verification_Type_Array`, l'item `dimensions` ayant pour valeur le nombre de dimensions du tableau et l'item `base_type` étant la conversion du type de base du tableau en structure `verification_type`.

Lorsque le calcul des attributs est complété, leurs noms (les chaînes de caractères « `VerificationConstantPool` » et « `TypeStateMapTable` ») sont ajoutés à la fin de la

table de constantes de la classe (`constant_pool`) sous forme de structures `CONSTANT_Utf8_info` et leurs indices sont inscrits dans leur item `name_index`. Les attributs pourront ainsi être lus ou interprétés par une machine virtuelle ou un autre logiciel.

4.3 Vérification avec certificat

Nous montrerons dans cette section un algorithme de vérification du code-octet utilisant les attributs « `VerificationConstantPool` » et « `TypeStateMapTable` ».

4.3.1 Analyse du certificat

En premier lieu, les attributs sont analysés et convertis sous forme de structures de données telles qu'exposées au chapitre 3 dans le cadre de l'analyse de flot de données, l'intention étant de reconstituer un résumé de cette analyse : le *point-fixe* des données pour les instructions ayant plusieurs prédécesseurs.

Ainsi, chaque entrée de l'attribut `TypeStateMapTable` sera convertie et mémorisée dans une table associant son adresse relative avec un environnement de blocs d'activation. Ces entrées sont des structures de type `TypeStateMap` qui consistent, en effet, en séquences de structures de type `TypeStateFrame` qui modélisent la pile et les variables locales pour chaque bloc d'activation d'un environnement. Les emplacements des piles et des variables locales contiennent des indices devant référer à un élément `CONSTANT_TypeSet_info` dans l'attribut `VerificationConstantPool` désignant l'ensemble de types pour cet emplacement. Ces types, représentés sous forme de structures `verification_type`, sont convertis inversement à la manière exposée à la section 4.2. Par exemple, une structure `Verification_Type_Int` correspond au type `int` et une structure `Verification_Type_Array{2,Verification_Type_Boolean}` correspond au type `array(2,`

`boolean`). Le nom des références est encodé dans la table de constantes dans le cas de structures de type `Verification_Type_Reference` et dans la table de constantes de vérification dans le cas de structures `Verification_Type_Reference`.

4.3.2 Algorithme de vérification

L'algorithme a pour but d'évaluer si chaque instruction d'une méthode peut être exécutée considérant les types que peuvent avoir les emplacements de la pile et des variables locales préalablement à leur exécution, tout en respectant les contraintes structurelles sur le code-octet (spécification de la machine virtuelle [LinY99] à la section 4.8.2).

Cet algorithme est un hybride entre les algorithmes d'analyse de flot de données et de vérification du chapitre 3. En effet, il prend en entrée un certificat représentant les types atteignant certaines instructions d'une méthode. Ceci constitue une partie de l'information produite par une analyse de flot de données. Nous ne connaissons cependant pas le graphe de contrôle ni la table des sous-routines mais ces informations peuvent être facilement déduites à partir du code et du certificat. En effet, le problème du flot de contrôle dynamique causé par l'utilisation des instructions `jsr - ret` est réglé car les adresses de sous-routines sont présentes dans le certificat. (Encore faut-il que le certificat soit valide, mais nous y reviendrons.)

L'algorithme commence avec la première instruction de la méthode. Les types l'atteignant sont déterminés soit par une entrée du certificat, `TypeStateMapTable[0]` si elle est présente, sinon par le descripteur de la méthode (de la même façon que l'analyse de flot de données est initialisée à la section 3.2.5). Cette instruction est associée avec un noeud de graphe et avec ses types en entrée (*n.in*). Le noeud est ensuite inséré dans une *worklist*.

Pour calculer l'analyse, répéter itérativement le processus suivant tant que la *worklist* n'est pas vide. Récupérer un noeud *n* de la *worklist*. La fonction de transition associée à l'instruction de ce noeud est calculée en prenant pour entrée l'état des types de l'environnement tel que mémorisé (*n.in*). Mais ce calcul sera effectué comme une simulation défensive de l'instruction en validant que seuls les types permis soient manipulés. Les mêmes règles exposées à la section 3.5 sont donc appliquées (le flot de contrôle et le polymorphisme des sous-routines, les gestionnaires d'exception, les contraintes de sous-type, les constructeurs, ...).

Le noeud *n* est ensuite marqué comme étant visité et ses successeurs sont identifiés en suivant les règles mentionnées aux sections 3.1.3 et 3.4.7. Pour tout successeur *s*, si *s* n'a pas déjà été visité et qu'il y a un item dans le certificat (`TypeStateMapTable[s]`) donnant les types en entrée pour cette instruction, on valide le branchement. Un branchement est valide si le résultat de la fonction de transition sur *n* est compatible avec l'item du certificat. Pour être compatibles, les deux structures de données doivent avoir le même nombre d'éléments et chaque ensemble de types calculé par la fonction de transition sur *n* pour chaque emplacement d'un environnement doit être sous-ensemble de l'ensemble pour l'emplacement correspondant dans le certificat.

Le noeud *s* est ensuite ajouté à la *worklist* avec les types du certificat comme types en entrée (*n.in*). Mais si *s* n'a pas déjà été visité et que le certificat ne contient pas d'information relativement à cette instruction, le noeud *s* est simplement ajouté à la *worklist* avec les types calculés par la fonction de transition sur *n* comme types en entrée (*n.in*). Si par contre le noeud *s* a déjà été visité, il doit avoir une entrée correspondante dans le certificat sinon une erreur de vérification est lancée. Ceci se justifie parce que l'algorithme ne visite chaque instruction qu'une fois et que s'il y a un branchement à une instruction déjà visitée, c'est que plusieurs instructions s'y branchent. Nous y reviendrons. Le branchement est donc validé en

vérifiant que chaque élément du résultat de la fonction de transition sur *n* est sous-ensemble des éléments de l'entrée du certificat.

Il reste un traitement particulier applicable aux instructions `jsr` et `ret`. Nous avons exposé en détail aux sections 3.4 et 3.5 les implications liées à ces instructions. Entre autres, la table de sous-routines, la détermination du flot de contrôle suite au `ret` et la fonction de transition *jsrBis* servant à propager les types entre le `ret`, le `jsr` et l'instruction de retour de la sous-routine. Ces notions sont ici utilisées.

Lorsque le noeud visité par l'analyse représente une instruction `jsr t`, la transition normale est d'abord calculée avec l'instruction à l'adresse `t` comme successeur. Ensuite, si la sous-routine `t` a déjà un `ret` d'associé, la transition *jsrBis* est calculée. D'autre part, lorsque l'analyse visite un noeud associé à une instruction `ret`, la fonction de transition *jsrBis* est encore utilisée pour la propagation des types aux instructions suivant les `jsr` associés à cette sous-routine dans le tableau de code.

Lorsque des entrées dans le certificat (`TypeStateMapTable[s]`) sont présentes pour les instructions de retour de la sous-routine (suivant le `ret`), ces types sont utilisés pour valider que le branchement est valide et sont ensuite mémorisés en entrée du noeud (*s.in*) lorsque celui-ci doit être ajouté à la *worklist* (c'est-à-dire, s'il n'a pas déjà été visité). Dans le cas contraire, le résultat de la fonction *jsrBis* est utilisé comme entrée pour les noeuds de retour de la sous-routine.

4.4 Validité de l'algorithme et du certificat

Il y a lieu de s'interroger sur la validité de la technique de vérification en deux temps. L'algorithme et les informations contenues dans le certificat sont-ils suffisants pour assurer la validité du code vérifié? Cette information est-elle fiable? Aussi, cette technique est-elle

viable quant à la taille mémoire théorique de son implémentation? Nous examinerons ces questions dans cette section.

Mais tout d'abord, il faut mentionner que l'algorithme ne fait qu'ajouter deux constantes dans la table des constantes (`constant_pool`) de la classe ainsi qu'un attribut à la classe et un attribut aux méthodes non abstraites et non natives dont le code contient des instructions ayant plus d'un prédécesseur. L'algorithme ne certifiera donc que les classes qui en ont besoin et ne modifie en rien le code des méthodes ou le reste de la classe. Cela permet d'apporter des modifications localisées, incrémentales et de conserver la compatibilité avec des versions antérieures des fichiers *class*, ce qui constitue un argument en faveur de cette technique.

4.4.1 Cohérence et caractère complet du certificat

D'abord, l'information nécessaire à la vérification du code-octet avec un algorithme apparenté à l'analyse de flot de données est principalement l'ensemble des types pouvant atteindre chaque variable pour tout point du programme, nommé *point-fixe*. Cette information dépend du chemin ou des chemins atteignant une instruction. Rappelons que les types en entrée pour une instruction sont égaux à l'union des types en sortie de tous ses prédécesseurs.

Considérant cela, nous observons qu'une instruction appartenant au code d'une méthode peut avoir 0, 1 ou plusieurs prédécesseurs (avec théoriquement un maximum de $n - 1$ si toutes les autres instructions se branchent à la même). Une instruction n'ayant aucun prédécesseur est une instruction « morte » : elle ne sera jamais exécutée et aucun type n'atteindra ses variables, si ce n'est le type `uninitialized` constituant son *point-fixe*. Quant à la première instruction de la méthode, elle a au moins un prédécesseur qui est le

noeud d'entrée du graphe servant à propager les types des arguments de la méthode, ce qui est en soi un *point-fixe*. Il est clair que les types en entrée pour une instruction avec un seul prédécesseur sont, intégralement, les types en sortie de ce dernier et que si ces types correspondent au *point-fixe*, leur état ne changera pas. Enfin, avant que les types en entrée d'une instruction avec plusieurs prédécesseurs n'atteignent un *point-fixe*, il faut que tous ses prédécesseurs atteignent aussi un *point-fixe* car la fonction d'union calculera un résultat différent tant que les ensembles en entrée augmenteront.

Le choix de n'inscrire dans le certificat que les informations relatives au *point-fixe* en entrée des instructions ayant plus d'un prédécesseur permet donc à l'algorithme de vérification en deux temps de ne visiter les instructions « vivantes » du code d'une méthode qu'une seule fois. Lorsqu'une instruction n'a qu'un prédécesseur, cette visite suffit à calculer le *point-fixe*. Lors de la propagation des types d'un noeud p à un successeur s, s'il y a une entrée associée à s dans le certificat, c'est supposément parce que s a plusieurs prédécesseurs et conséquemment le *point-fixe* en sortie de p doit être un sous-ensemble de cette entrée dans le certificat. Enfin, lorsqu'une instruction p effectue un branchement à un successeur s, si s est marqué comme étant visité, cela signifie que s a plusieurs prédécesseurs et qu'il doit y avoir une entrée associée au *point-fixe* de s dans le certificat, sinon l'information est incomplète et il y a donc erreur.

Quant à la table des constantes de vérification, elle doit contenir tous les ensembles de types présents dans les *points-fixes* du certificat. Si une information est manquante (un ensemble de types ou un type), l'algorithme de vérification le détectera lorsqu'il échouera la lecture du certificat ou le calcul d'une fonction de transition.

4.4.2 Intégrité du certificat

La question de la fiabilité de l'information du certificat reste ouverte car nous n'avons pas utilisé de mécanismes d'authentification des classes ou de renforcement de l'intégrité des données. Le certificat peut donc montrer des malformations, soit intentionnelles lorsque le format de fichier *class* est exploité malicieusement, soit accidentelles lorsqu'il y a erreur de transmission des données (problème d'alignement ou autre). Nous montrerons ici qu'un certificat altéré ne signifie pas forcément que le code-octet de la méthode n'est pas vérifiable, mais l'algorithme de vérification identifiera les cas ou l'information ne respecte pas les contraintes de vérification du code-octet, lesquelles classes seront qualifiées de non vérifiables.

Cinq types de malformation ont été identifiés pour l'attribut `VerificationConstantPool`. Ils sont ici présentés.

Premièrement, l'attribut peut comporter trop de constantes. Cela ne pose pas de problème à la vérification tant que les constantes nécessaires sont présentes.

Deuxièmement, l'attribut peut ne pas en comporter assez. Ceci sera détecté par l'algorithme lorsqu'il tentera de récupérer une constante qui est absente du certificat.

Troisièmement, une constante de type `CONSTANT_TypeSet_info` peut comporter trop d'éléments. Cela voudrait dire que l'ensemble de types pour un emplacement d'une entrée dans le certificat serait un sur-ensemble du *point-fixe*. Pour fin de vérification, cela ne pose un problème que si le calcul des fonctions de transitions associées aux instructions du code ne peut être effectué. Donc, l'ajout de types compatibles aux autres ne pose pas de problème et l'ajout de types incompatibles est détecté et la classe est rejetée comme étant non vérifiable.

Quatrièmement, une constante de type `CONSTANT_TypeSet_info` peut ne pas comporter suffisamment d'éléments. Cela voudrait dire que l'ensemble de types pour un emplacement d'une entrée dans le certificat serait un sous-ensemble du *point-fixe*. Comme les entrées du certificat associent aux instructions du code ayant plusieurs prédécesseurs le *point-fixe* supposément égal à l'union des sorties de tous ces prédécesseurs, si ce *point-fixe* est un sous-ensemble de cette union, cela signifie qu'au moins un des prédécesseurs ne pourra effectuer le branchement. En effet, la fonction d'union doit avoir des ensembles en entrée et un sur-ensemble en sortie, et pas le contraire.

Cinquièmement, une constante de type `CONSTANT_Utf8_info` peut ne pas représenter le type attendu. Cela implique qu'un type référence sera erroné dans un des point-fixe. Comme précédemment, les fonctions de transition détermineront si ce type est acceptable et s'il est compatible avec le type attendu (pensons ici aux contraintes de sous-type).

Cinq types de malformation ont aussi été identifiés pour l'attribut `TypeStateMapTable`. Les voici.

Premièrement, le certificat peut avoir trop d'entrées, c'est à dire que des *points-fixes* sont associés à des instructions ayant zéro ou un prédécesseur. Si cette entrée indique les types justes, il n'y a pas de problème et ces types seront utilisés pour poursuivre l'analyse. Si les types sont erronés, les fonctions de transition détermineront s'il y a lieu de rejeter la classe.

Deuxièmement, le certificat peut ne pas avoir assez d'entrées, c'est-à-dire que des instructions ayant plusieurs prédécesseurs n'ont pas de *points-fixes* associés. Ce problème a déjà été évoqué et nous avons expliqué que le code est rejeté car nous ne sommes pas en mesure de calculer facilement le *point-fixe* associé.

Troisièmement, une entrée du certificat peut comporter trop d'éléments (trop de blocs d'activation ou trop d'emplacements pour un bloc). Dans ce cas, soit l'attribut n'aura pu être lu ou converti correctement, soit une fonction de transition ne pourra être calculée correctement (par exemple s'il y a trop d'éléments sur la pile).

Quatrièmement, une entrée du certificat peut ne pas comporter assez d'éléments (de blocs d'activation ou d'emplacements pour un bloc). Ce problème est détecté de la même manière que le troisième.

Cinquièmement, un emplacement d'un bloc d'activation peut référer à la mauvaise constante dans la table des constantes de vérification ou même avoir une référence hors bornes. Normalement, le certificat est défini de façon à ce que les emplacements de la pile et des variables locales ne réfèrent qu'à des constantes de type `CONSTANT_TypeSet_info` dans la table des constantes de vérification. Donc, si un emplacement réfère à un autre type de constante, il y a erreur. Les références hors bornes sont bien sûr erronées. Enfin, si les types référés ne sont pas admis par une fonction de transition du code, ou si un branchement entre deux instructions n'est pas réalisable, il y a aussi erreur.

4.4.3 Taille du certificat

Nous examinons ici la taille du certificat dans un cadre théorique. Le chapitre 5 apportera des mesures empiriques à ce sujet.

La spécification de la machine virtuelle Java, section 4.7, détermine le format d'un attribut. Le format général est le même, que l'attribut soit associé à une classe, une méthode ou à une autre structure de la classe. Notons que la structure d'un attribut indique son nom et sa longueur. Le nom réfère à une constante dans la table de constantes de la classe (`constant_pool`). Le certificat décrit précédemment utilise deux sortes d'attributs :

« `VerificationConstantPool` » et « `TypeStateMapTable` ». Conséquemment, notre technique requiert deux ajouts au `constant_pool`. Remarquez que c'est une valeur constante et minime compte tenu de la capacité maximale de 65535 entrées de la table de constantes.

La longueur maximale d'un attribut est encodée sur quatre octets et cela offre donc près de 4 giga-octets de mémoire pour encoder un seul attribut, ce qui est raisonnablement suffisant pour encoder un certificat pour toute classe bien constituée. Cela ouvre la porte à qui voudrait exploiter cet élément en générant intentionnellement du code nécessitant la manipulation d'une quantité gigantesque d'information pour encoder le certificat, mais il s'agit d'un cas marginal qui aura comme seule conséquence la consommation excessive de mémoire. Cette limitation est due au format de fichier *class* et concerne aussi dans d'autres structures du format de fichier *class*. Par exemple, la longueur maximale du code des méthodes est aussi de près de 4 giga-octets.

4.5 Conclusion

En résumé, nous avons montré que la technique de vérification en deux temps, les structures de données et l'algorithme proposés dans ce chapitre sont viables.

Nous avons vu qu'en ce qui concerne l'attribut de méthode, il est suffisant d'encoder le *point-fixe* associé aux instructions ayant plus d'un prédécesseur, cela ayant pour conséquence de diminuer le nombre d'entrées dans le certificat. De plus, le choix d'utiliser un attribut de classe (partagé par toutes les méthodes) permet la réutilisation des constantes pour les ensembles de types encodés dans le *point-fixe* et favorise l'économie d'espace mémoire.

Nous avons aussi vu que si le certificat montre une malformation (malveillante ou fortuite), d'une part, si cela entraîne une erreur de vérification, la classe est rejetée. Si

d'autre part cela n'entraîne pas d'erreur de vérification, la classe est acceptée, car l'algorithme de vérification est suffisamment défensif pour pouvoir garantir que le code de la classe n'effectue pas d'opération illégale. Notons bien le fait que l'algorithme ne peut pas toujours déterminer si c'est le code des méthodes ou le certificat qui est invalide, mais considérant qu'un élément d'information erroné est identifié par le vérificateur, il est raisonnable de rejeter la classe.

Enfin, nous avons vu que la structure de fichier *class* permet l'utilisation d'un espace mémoire suffisant pour pouvoir encoder le certificat. Le chapitre 5 exposera des mesures expérimentales à ce sujet.

CHAPITRE V

IMPLÉMENTATION ET EXPÉRIMENTATIONS

Dans ce chapitre, nous présentons l'implémentation du vérificateur, du compilateur de certificat ainsi que les données et les mesures des bancs d'essais.

Afin de valider les algorithmes de vérification présentés aux chapitres précédents, un vérificateur a été élaboré et des expérimentations ont été menées pour évaluer, d'une part, la validité du vérificateur et, d'autre part, le taux d'augmentation de la taille de classes lorsque certifiées pour la technique de vérification en deux temps. Des comparaisons ont aussi été faites avec les performances du compilateur *javac* du *jdk 1.6* [JdkX] de Sun qui utilise une technique de certification apparentée à la nôtre, issue du travail de d'Eva Rose [RosR98], [Rose02].

5.1 Implémentation du vérificateur et du compilateur de certificats

Les algorithmes de vérification en une et deux phases ont été implémentés en langage Java. Le vérificateur constitue une application autonome avec interface en ligne de commande et est aussi intégré à la machine virtuelle SableVM [Gagn02]. Les principales composantes logicielles seront exposées dans ce qui suit.

Afin de structurer les éléments constitutifs d'une classe, tel que décrit dans le chapitre 4 de la spécification de la machine virtuelle Java [LinY99], le paquetage `classFile` regroupe des classes les représentant. La correspondance entre le code et la spécification est directe : `classFile.SClassFile` comme structure de plus haut niveau et qui comprend des champs tels `classFile.SConstantPool` pour la table de symboles, `classFile.SMethods` pour les méthodes de la classe, et ainsi de suite. Les instructions du code de méthode sont représentées par les classes du paquetage `classFile.sinstructions`.

Le paquetage `parser` regroupe des classes permettant la lecture et l'écriture de fichiers au format *class* dans un système de fichiers local. Les classes de ce paquetage sont associées à chaque classe du paquetage `classFile`. Par exemple, `parser.SClassFileParser` sert à lire/écrire la classe `classFile.SClassFile`, `parser.SConstantPoolParser` pour la table de symboles ou encore `parser.SCodeAttributeParser` qui permet la lecture et l'écriture de l'attribut `Code` d'une méthode.

Les paquetages `dataFlowAnalysis` et `types` regroupent les classes servant à implémenter l'analyse de flot de données. Les contraintes de sous-type sont représentées par les classes `util.SAssignmentCompatibilityConstraint` et `util.SAssignment CompatibilityConstraintRepository`.

La classe `SVerifier` implémente l'algorithme de vérification en un temps.

Le paquetage `test` regroupe des classes enfreignant les contraintes statiques et structurelles du code-octet ainsi que le format de certificat présenté au chapitre 4. Des anomalies dans le code causent des erreurs de débordement de pile, d'indice hors borne du `VerificationConstantPool`, de types incompatibles, etc.

Le paquetage `typeCheckingToolkit` implémente l'algorithme de vérification en deux temps en réutilisant une partie des services de la classe `SVerifier` et du paquetage `dataFlowAnalysis`. Il implémente aussi un compilateur de certificats avec la classe `typeCheckingToolkit.CertificateCompiler`. Ce compilateur prend en entrée des fichiers *class* (des classes déjà compilées en code-octet), génère un certificat pour la classe et chacune de ses méthodes comportant du code et réécrit les fichiers sur disque. Afin d'ajouter les attributs spéciaux à une classe, les sous-paquetages `typeCheckingToolkit.classFile` et `typeCheckingToolkit.parser` permettent une extension transparente des structures de données des paquetages `classFile` et `parser`.

Les deux algorithmes de vérification ont la même interface utilisateur. L'application autonome peut être invoquée avec la classe `SVerifier` en ligne de commande. Si la classe contient un certificat, il sera utilisé pour effectuer la vérification (deuxième phase), sinon l'algorithme en un temps sera invoqué. Des options en ligne de commande permettent différentes fonctionnalités : vérifier statiquement ou dynamiquement les contraintes de sous-type impliquant des références à d'autres classes que la classe courante (les contraintes peuvent être vérifiées relativement aux chemins des bibliothèques de classes passées en paramètre au vérificateur ou retournées sous forme d'ensemble pour vérification ultérieure), générer de l'information de débogage et effectuer la vérification soit d'une méthode précise ou de toutes les méthodes (par défaut) dans une classe.

Le vérificateur a aussi été intégré à la machine virtuelle SableVM [Gagn02]. Après le chargement d'une classe, la machine virtuelle effectuera éventuellement l'édition de liens de cette classe, ce qui implique sa vérification. SableVM réalise ce processus avec la fonction `_svmf_verify_class()` du fichier `verifier.c` qui invoque simplement le vérificateur avec comme paramètres le chargeur de classe et le nom complet de la classe à vérifier.

5.2 Bancs d'essais

Les expérimentations répondent à deux objectifs : s'assurer de la fiabilité et de l'exactitude des calculs du vérificateur dans ses différentes configurations et évaluer l'impact du nouveau certificat sur la taille de classes existantes. Cette section présente les données utilisées pour les bancs d'essais ainsi que des expérimentations montrant le niveau d'atteinte de ces deux objectifs.

5.2.1 Données

Nous avons choisi un échantillon de programmes comprenant en tout un grand nombre de classes et d'interfaces. Ils ont été choisis pour leur représentativité, ces applications étant d'usage assez répandu, et parce qu'elles sont des projets logiciels libres. Les programmes constituant les bancs d'essais sont les suivants.

Gnu-classpath

Gnu-classpath [GnuCP] est une implémentation de l'API de Java. La version 0.14 a été utilisée. Le projet classpath comporte un grand nombre de classes.

Soot

Soot [Soot] est un compilateur optimisant du code Java. La version 2.2.1 est utilisée. Soot a été choisi pour sa taille et aussi pour son utilisation poussée des interfaces, des classes abstraites et de l'héritage.

Sablecc

Sablecc [SablCC], version 3.1, est un générateur de compilateurs et, bien que comportant relativement peu de classes, est intéressant pour les mêmes raisons que Soot.

Ashes

Ashes [Ashes] est une collection de programmes pour les bancs d'essais de compilateurs optimisant ou d'interpréteurs de code-octet. Il s'agit de la version de l'année 2 000 (la dernière en date). Ashes est intéressant pour ses classes atypiques, produites par des compilateurs autres que le *javac* de Sun (principalement *jikes* [Jikes] et *jasmin* [Jasmin]) et comprend plus de 200 sous-routines. L'inconvénient est que le code source n'est pas distribué en totalité avec Ashes. Ainsi, les mesures de notre technique ne peuvent être comparées à ce que produirait le compilateur *javac* version 1.6. Nous l'avons tout de même conservé pour sa singularité et parce qu'il permet quand même de mesurer les performances de notre technique.

Eclipse

Eclipse [Eclips] est un environnement de développement, logiciel libre, largement utilisé par l'industrie logicielle et les milieux de recherche informatique. La version utilisée est 3.2.1 (la dernière en date). Eclipse comporte un nombre imposant de classes (plus de 20 000) et plus de 3 000 sous-routines. Cette version d'Eclipse est conçue pour être compilée avec le JDK 1.4 et le JRE 1.5 [JdkX] conjointement; la compilation avec le *javac* 1.6 n'a donc pas été réalisée.

Tests

Le paquetage `verifier.test`, présenté précédemment, comporte près d'une cinquantaine de classes qui ont servies à tester le bon comportement du vérificateur.

5.2.2 Validation du vérificateur

Les 36 940 classes des cinq premiers bancs d'essais ont toutes été vérifiées avec succès par le vérificateur en un temps. Elles ont ensuite été certifiées par notre compilateur

et ont été vérifiées avec succès par le vérificateur en deux temps. Le bon comportement de nos programmes pour un échantillon de cette taille de classes « réelles » (représentatives d'un usage courant) indique la justesse des algorithmes et un bon niveau de fiabilité de l'implémentation.

Les erreurs dans les classes de `verifier.test` ont toutes été détectées par le vérificateur en un temps. Le générateur de certificats ne peut cependant générer de certificat pour toutes les classes erronées. Par exemple, s'il y a dépassement de pile causé par une instruction ou si le flot de contrôle est impossible, le certificat n'est pas généré. Par conséquent, nous ne pouvons tester le vérificateur en deux temps avec ces classes. Mais ce vérificateur utilisant les mêmes fonctions de transition pour les instructions du code-octet que le vérificateur en un temps, les erreurs seraient aussi détectées.

5.2.3 Mesures

Un des objectifs des expérimentations est d'évaluer l'impact du nouveau certificat sur la taille de classes existantes et de le comparer avec *javac* version 1.6 . Les données des bancs d'essais sont utilisées et mesurées dans différents contextes. Les mesures sont effectuées avec les APIs de Java (les classes `File`, `JarFile` et `JarEntry`) qui donnent le nombre d'octets de fichiers. Nous portons notre investigation sur le rapport entre la taille originale des classes et leur taille une fois certifiées. Nous évaluons aussi le taux de compressibilité relatif des classes avec et sans certificat en utilisant l'application `jar`. Nous utilisons cette application car elle est d'utilisation généralisée pour la distribution de classes.

Nous présentons d'abord les caractéristiques des bancs d'essais et montrons la proportion des classes et interfaces qui nécessitent l'ajout de certificats. Les tableaux 5.1 et 5.2 montrent les mesures pour notre technique et pour *javac* version 1.6.

Tableau 5.1 Caractéristiques des bancs d'essais avec notre certificat

Banc d'essai	Classes	Sous-routines	Classes certifiées	Méthodes	Méthodes avec du code	Méthodes certifiées	Ratio classes certifiées	Ratio méthodes avec code certifiées
sablecc	286	0	141	2372	2275	613	0,49	0,27
soot	2249	3	1033	17303	16045	3853	0,46	0,24
classpath	3862	80	1711	33296	27248	7339	0,44	0,27
sous-total	6397	83	2885	52971	45568	11805	-	-
ashes	5587	210	2832	30621	29720	10563	0,51	0,36
eclipse	24956	3735	14345	194256	179488	65360	0,57	0,36
total	36940	4028	20062	277848	254776	87728	-	-

Tableau 5.2 Caractéristiques des bancs d'essais avec *javac* 1.6

Banc d'essai	Classes	Sous-routines	Classes certifiées	Méthodes	Méthodes avec du code	Méthodes certifiées	Ratio classes certifiées	Ratio méthodes avec code certifiées
sablecc	286	0	140	2372	2275	691	0,49	0,3
soot	2249	3	1120	17333	16075	4778	0,5	0.3
classpath	3862	80	1829	33405	27357	8918	0,47	0,33
total	6397	83	3089	53110	45707	14387	-	-

La comparaison des résultats des tableaux 5.1 et 5.2 montrent que notre technique génère en général moins de certificats pour un programme que *javac*.

Dans notre cas, environ la moitié des classes sont certifiées. Cela est dû au fait que les interfaces n'ont en général pas de code, sauf parfois des initialiseurs statiques et que les classes abstraites n'implémentent souvent aucune méthode et ne comportent donc aucun attribut code.

Résultat également singulier, seulement le tiers des méthodes avec du code nécessitent un certificat. Cela indique qu'une grande majorité du code des méthodes contiennent des chemins de contrôle simples de sorte qu'aucune instruction a plus d'un prédécesseur (cela exclut généralement les *exception-handlers*, les sous-routines ou les instructions de branchement causant des itérations par exemple).

Du côté de *javac* 1.6, nous constatons d'abord qu'il génère plus de méthodes que les versions précédentes du compilateur : 30 de plus pour *soot* et 109 de plus pour *classpath*. Cela semble être dû à une nouvelle manière de compiler l'héritage de méthodes pour des classes internes.

Nous voyons ensuite qu'il produit près de 4% de plus de classes certifiées que notre technique pour *soot* et 3% de plus pour *classpath*. Du côté des méthodes certifiées, l'augmentation est de l'ordre de 3% pour *sablecc*, 6% pour *soot* et 3% pour *classpath*.

Ces écarts sont vraisemblablement dus aux différents critères utilisés par les algorithmes pour sélectionner les adresses du code à inclure dans le certificat. Le nombre d'entrées dans notre certificat est proportionnel au nombre d'instructions d'une méthode ayant plus d'un prédécesseur tandis que pour celui de Sun il est proportionnel au nombre d'instructions de branchement d'une méthode.

Nous examinerons maintenant dans quelle mesure l'ajout de certificats à des classes cause l'augmentation de leur taille.

Tableau 5.3 Ratios d'augmentation de la taille de classes suite à l'ajout de certificats, relativement aux classes pouvant être certifiées

Banc d'essai		Ratio minimal (octets)	Ratio maximal (octets)	Ratio moyen (octets)
sablecc	-cert	8807 / 8621 = 1,02	2938 / 2124 = 1,38	538673 / 480824 = 1,12
	-cert -jar	1857 / 1781 = 1,04	1250 / 1000 = 1,25	204523 / 184405 = 1,11
	-java6	8677 / 8621 = 1,01	2921 / 2452 = 1,19	505167 / 478152 = 1,06
	-java6 -jar	1813 / 1781 = 1,02	1418 / 1223 = 1,16	195065 / 183064 = 1,07
soot	-cert	14038 / 13956 = 1,01	6424 / 4601 = 1,40	4140790 / 3704487 = 1,12
	-cert -jar	11077 / 10965 = 1,01	338 / 268 = 1,26	1845296 / 1696452 = 1,09
	-java6	29178 / 29130 = 1,00	1621 / 1040 = 1,56	4028425 / 3847023 = 1,05
	-java6 -jar	30182 / 30294 = 1,00	784 / 584 = 1,34	1854725 / 1761684 = 1,05
classpath	-cert	37021 / 36903 = 1,00	11038 / 3338 = 3,31	7391010 / 6668132 = 1,11
	-cert -jar	14646 / 14574 = 1,00	1045 / 801 = 1,30	3427229 / 3174361 = 1,08
	-java6	36939 / 36903 = 1,00	12061 / 8644 = 1,40	7121318 / 6820578 = 1,04
	-java6 -jar	14611 / 14574 = 1,00	4724 / 3705 = 1,28	3415270 / 3251949 = 1,05
ashes	-cert	60749 / 60586 = 1,00	1022 / 553 = 1,85	9446179 / 8300428 = 1,14
	-cert -jar	24488 / 24398 = 1,00	578 / 405 = 1,43	4414102 / 4011061 = 1,10
eclipse	-cert	71573 / 71444 = 1,00	15839 / 4772 = 3,32	83229574 / 75355742 = 1,10
	-cert -jar	17214 / 17141 = 1,00	2747 / 2050 = 1,34	34625064 / 32033856 = 1,08

Tableau 5.4 Ratios d'augmentation de la taille de classes suite à l'ajout de certificats, relativement à toutes les classes d'un banc d'essai

Banc d'essai		Ratio minimal (octets)	Ratio maximal (octets)	Ratio moyen (octets)
sablecc	-cert	5905 / 5905 = 1,00	2938 / 2124 = 1,38	620003 / 562154 = 1,10
	-cert -jar	1431 / 1431 = 1,00	1250 / 1000 = 1,25	248046 / 227928 = 1,09
	-java6	5905 / 5905 = 1,00	2921 / 2452 = 1,19	589169 / 562154 = 1,05
	-java6 -jar	348 / 350 = 0,99	1418 / 1223 = 1,16	239927 / 227928 = 1,05
soot	-cert	767 / 767 = 1,00	6424 / 4601 = 1,40	5062762 / 4626459 = 1,09
	-cert -jar	465 / 465 = 1,00	338 / 268 = 1,26	2332015 / 2183171 = 1,07
	-java6	767 / 767 = 1,00	1621 / 1040 = 1,56	4807861 / 4626459 = 1,04
	-java6 -jar	381 / 385 = 0,99	784 / 584 = 1,34	2276410 / 2183171 = 1,04
classpath	-cert	360 / 360 = 1,00	11038 / 3338 = 3,31	9025096 / 8302218 = 1,09
	-cert -jar	235 / 235 = 1,00	1045 / 801 = 1,30	4291552 / 4038684 = 1,06
	-java6	925 / 1024 = 0,90	4658 / 1974 = 2,36	8607783 / 8302218 = 1,04
	-java6 -jar	492 / 534 = 0,92	1536 / 760 = 2,02	4204000 / 4038684 = 1,04
ashes	-cert	652 / 652 = 1,00	1022 / 553 = 1,85	12438127 / 11292376 = 1,10
	-cert -jar	395 / 395 = 1,00	578 / 405 = 1,43	5995099 / 5592058 = 1,07
eclipse	-cert	1067 / 1067 = 1,00	15839 / 4772 = 3,32	95166164 / 87292332 = 1,09
	-cert -jar	548 / 548 = 1,00	2747 / 2050 = 1,34	39940737 / 37349529 = 1,07

Le tableau 5.3 montre la comparaison des tailles de fichiers avant et après l'ajout de certificats relativement aux seules classes certifiées. Le tableau 5.4 mesure l'impact relatif à toutes les classes d'un banc d'essai, incluant celles ne nécessitant pas de certificat. Les trois premiers bancs d'essais sont mesurés dans quatre configurations différentes : certifié par notre algorithme, certifié par *javac*, en taille réelle et compressé avec l'outil *jar*. Les deux autres bancs d'essais ne sont pas compilés avec *javac 1.6* à cause de l'incompatibilité.

Le tableau 5.3 nous indique que l'augmentation minimale lors de l'ajout de certificat est entre 1% et 4%.

Le ratio d'augmentation maximale est entre 16% et 56% avec une contre-figure de plus de 230% pour deux classes de *classpath* et de *eclipse* avec notre technique. Nous voyons ensuite que ces chiffres sont ramenés à une augmentation d'au plus 43% lorsque les classes sont compressées. Du côté de *javac*, l'augmentation maximale est de 56% dans le cas de *soot*, sans compression, et descend aussi à près de 34% avec compression.

Cette dramatique augmentation de taille causée par notre technique concerne la méthode `classpath.gnu.java.rmi.registry.RegistryImpl_Skel.dispatch()`. Cette méthode comporte plusieurs éléments qui font augmenter la taille de notre certificat. D'abord elle possède 20 variables locales, ce qui n'est pas peu. Rappelons que cela implique que chaque entrée du certificat comprendra 20 espaces réservés aux variables locales. Ensuite, la méthode comprend plusieurs instructions de branchements (notamment 24 *exception_handlers*) qui font que plusieurs instructions ont plus d'un prédécesseur et sont donc incluses dans le certificat. De plus, on y retrouve 5 sous-routines. Cela signifie que chaque bloc d'activation de base encodé dans chaque entrée du certificat est quintuplé. Toutes ces conditions, nous le constaterons, surviennent rarement dans une même méthode.

L'augmentation de taille moyenne avec notre technique est plus acceptable et elle se situe entre 10% et 12% sans compression et entre 8% et 11% avec compression. Pour *javac*, l'augmentation va de 4% à 6% sans compression et de 5% à 7% avec compression.

Voyons si le ratio d'augmentation de la taille des classes relativement à toutes les classes des bancs d'essais est différent. Le tableau 5.4 nous indique que l'augmentation minimale et bien sûr nulle (avec même une diminution d'un pourcent lorsque certaines classes compilées par *javac* sont compressées, possiblement à cause d'optimisations du code-

octet effectuées par *javac*). Le ratio d'augmentation maximal pour notre technique reste semblable au précédent, soit entre 38% et 231% lorsque les classes ne sont pas compressées et s'améliore entre 25% et 43% lorsqu'elles sont compressées. *Javac* peut causer une augmentation supérieure à celle observée dans le tableau 5.3, avec un ratio allant de 19% à 136% sans compression et de 16% à 102% avec compression. Enfin, le ratio d'augmentation moyenne se situe pour notre technique à 9% ou 10% sans compression et entre 6% et 9% compressé. Du côté de *javac* les ratios sont de 4% ou 5% avec ou sans compression.

Nous pouvons constater que bien que notre technique effectue la certification de moins de classes et de méthodes que celle de Sun, leur technique est deux fois plus économique que la nôtre si l'on considère la taille normale des classes. Cet écart diminue à environ 30% lorsque les classes sont compressées. Comme facteur de cette différence, notons que leur algorithme utilise des techniques avancées de compression des données. De plus, leur certificat contient relativement moins d'information que le nôtre considérant que nous encodons le résultat d'une analyse de flot de données polyvariante et que nous supportons les sous-routines. Sommes toutes, nous jugeons que l'augmentation moyenne d'environ 10% de la taille des classes résultant de notre format de certificat est acceptable, d'autant plus que le taux de compression de ce certificat est supérieur à celui produit par *javac*, ce qui indique que notre format peut encore être optimisé.

5.3 Conclusion

Nous avons exposé dans ce chapitre les moyens et les données utilisés pour mener les expérimentations. Nous avons vu que les algorithmes de vérification et leur implémentation sont satisfaisants quant à leur justesse. Nous avons aussi vu que les nouveaux attributs utilisés pour la vérification en deux temps n'entraînent pas un coût trop important en espace

mémoire et que, de surcroît, leur taux de compressibilité est grand relativement aux autres données d'une classe.

CHAPITRE VI

TRAVAUX RELIÉS

La littérature portant sur la vérification du code-octet est volumineuse. Nous présentons dans ce chapitre les principaux travaux dans les trois thèmes qui sont plus le apparentés avec le sujet de notre travail : la vérification, la vérification de sous-routines et le code-certifié.

La technique de vérification du code-octet la plus utilisée se sert de l'analyse de flot de données et les techniques que nous avons présentées aux chapitres 3 et 4 n'y font pas exception. La première description d'une telle technique à être publiée est la spécification de la machine virtuelle Java [LinY96], chapitres 4.8 et 4.9.

Il s'agit d'une description informelle du vérificateur distribué avec la machine virtuelle de Java de la compagnie Sun Microsystems. Cette analyse effectue donc une interprétation défensive des instructions du code des méthodes de la classe à vérifier en propageant des types comme valeur pour les variables. Il s'agit d'une analyse monovariante car il y a un type par emplacement. Les types utilisés sont calqués sur les types du langage Java (`int`, `long`, `float`, `double`, `reference`, `returnAddress`, ...). Ils sont ordonnés selon un demi-treillis où toute paire de types a une borne supérieure minimale commune dans le treillis. Nous remarquons ici qu'en Java les interfaces permettent l'héritage multiple et qu'ainsi deux classes implémentant les mêmes interfaces se retrouveraient avec plusieurs

bornes supérieures minimales communes qui sont différentes. Dans ce cas, la spécification indique que le résultat de l'union de deux interfaces est la référence `java.lang.Object` qui est super-type de toutes les interfaces. Mais cela reporte à la machine virtuelle le fardeau de vérifier, lors de l'exécution, que les instructions `invokeinterface` *`I.m()`* opèrent effectivement sur une référence implémentant l'interface *I*.

Le comportement des instructions est simulé sur une abstraction de la pile et des variables locales en effectuant une interprétation défensive. Des compléments à l'algorithme concernent la vérification de constructeurs, de gestionnaires d'exception, d'objets non initialisés et des sous-routines.

Les sous-routines y sont décrites pour compiler les structures `try - finally` de Java. Pour en effectuer la vérification, la spécification suggère d'utiliser une pile pour gérer l'historique d'appels des sous-routines et des ensembles de variables locales modifiées pour chaque sous-routine. Ainsi, nous serions en mesure de déterminer, lors du retour d'une sous-routine, si cela entraîne aussi le retour d'autres sous-routines imbriquées. Chaque variable locale ayant été modifiée par une instruction considérée comme appartenant à une sous-routine sera ajoutée à l'ensemble des variables locales modifiées pour cette sous-routine. Ces ensembles seront utilisés pour propager sélectivement les types lors du retour d'une sous-routine : pour les variables locales modifiées, les types propagés seront ceux du `ret` et pour les autres variables locales, les types propagés seront ceux du `jsr` ayant invoqué la sous-routine. Cette technique est suffisante pour le code-octet produit par un compilateur Java standard. Mais comme nous l'avons montré à la section 3.4, les instructions `jsr` et `ret` offrent bien plus de richesse et aussi de difficultés à analyser (polymorphisme, flot de contrôle dynamique et non séquentiel). La technique de Sun est donc limitée. D'autre part, Qian [Quia00] a démontré que la fonction de transition pour le `ret` n'était pas monotone croissante. En effet, un type propagé lors d'une itération en utilisant le type au niveau du

`jsr` peut être remplacé par un autre type qui provient du `ret` si une variable locale est modifiée entre les deux itérations. Et si ce type est inférieur au premier, la fonction n'est plus monotone croissante. Mais Qian a aussi prouvé que cela n'est pas un problème et que l'algorithme se termine effectivement avec un résultat convenable car les données vont se stabiliser une fois que toutes les variables locales modifiées pour la sous-routine auront été identifiées.

Plusieurs travaux se sont basés sur la spécification de Sun soit pour en faire une implémentation aussi fidèle que possible ou alors pour l'étendre ou encore la formaliser.

Le projet logiciel libre *Byte Code Engineering Library* [BCEL] comprend une implémentation d'un vérificateur tel que décrit dans la spécification « officielle ».

Les travaux de Stata et Abadi [StaA99] constituent la première publication d'une formalisation du processus de vérification de code-octet utilisant une analyse de flot de données. Ils élaborent formellement des règles de types pour les transitions associées aux instructions. Pour la gestion des sous-routines, ils utilisent aussi une pile d'appels et chaque instruction est étiquetée selon sa position dans l'historique d'appels. Leur formalisme porte cependant sur un sous-ensemble du code-octet et il ne supporte pas les gestionnaires d'exception ni l'initialisation d'objets.

Freund et Mitchell [FreM99] ont poussé un peu plus le travail de Stata et Abadi pour y inclure la gestion d'exceptions. Mais pour y arriver, ils ont simplifié encore plus les sous-routines en posant comme restriction que seul un `ret` ou une exception puisse causer la sortie d'une sous-routine. Autrement dit, une instruction dans une sous-routine ne peut effectuer un branchement que si la cible est dans la même sous-routine. L'historique d'appels de sous-routines est structuré avec un graphe. Le traitement des sous-routines dans ce formalisme est rigide, ce qui réduit le domaine de programmes vérifiables.

Hagiya et Toyawa [HagT98] utilisent aussi un formalisme semblable en y ajoutant un type spécial qui est propagé par l'analyse. Il s'agit du type `last(n)` qui désigne le type de la variable locale `n` lors de l'appel de la sous-routine (le `jsr`). Ce type est utilisé pour calculer la transition du `ret` seulement pour les variables locales non utilisées par la sous-routine afin de palier partiellement au problème du polymorphisme. Cependant, ce problème est aussi causé par les variables locales qui sont modifiées par la sous-routine. De plus, le problème de la structure dynamique et de l'historique des sous-routines n'est pas résolu.

Les techniques précédentes sont toutes monovariantes car elles considèrent un seul état des types pour chaque instruction. Certains travaux ont étudié une approche polyvariante.

Leroy [Lero01] propose une technique de vérification polyvariante avec contours. Chaque instruction est étiquetée avec la liste des `jsr` ayant permis de l'atteindre; c'est le contour de l'instruction. Toute instruction est vérifiée une fois par contour lui étant associé en utilisant les types des variables au `jsr` correspondant, ce qui permet d'utiliser les bons types et d'éviter le polymorphisme. Mais cette technique, comme l'*inlining*, a une complexité exponentielle. En effet, si une instruction a un contour comptant n éléments, donc n sous-routines imbriquées, et que chaque sous-routine est appelée par m `jsr`, alors cette instruction sera analysée m^n fois. Aussi, certains cas particuliers d'utilisation de sous-routines ne peuvent être correctement identifiés par cet algorithme.

Une autre forme d'algorithme de vérification polyvariante utilise le *model checking*. Posegga et Vogt [PosV98] d'abord et ensuite Henrio et Serpette [HenS01], Coglio [Cogl02] et Nipkow avec Paulson et Wenzel [NiPW02] ont effectué des travaux dans cette direction. L'idée est d'évaluer tous les états atteignables par un interpréteur abstrait sur des types en les

calculant avec la relation $T : (p, S, R) \rightarrow (p', S', R')$, ce qui est la combinaison de la relation de transition sur les types des emplacements de la pile et des variables locales et de la relation de successeur entre les instructions. Nous voyons qu'en fait, lorsqu'il n'y a aucune sous-routine dans une méthode, ceci est équivalent au calcul effectué par une analyse de flot de données. Mais en présence de sous-routines, la technique consiste à dupliquer les états des variables, de façon semblable aux contours de Leroy [Lero01]. Cela est effectué, lors du calcul de la relation T, si une instruction a plus d'un prédécesseur et que les types en sortie de ceux-ci ne peuvent être réunis. Ils ne peuvent être réunis s'il y a présence d'adresses de retour de sous-routine (l'adresse de l'instruction suivant le `jsr`) différentes pour une variable. Cela advient, par exemple, lorsque plusieurs `jsr` invoquent une sous-routine. Dans ce cas, donc, un état des types est conservé pour chaque « contour » (jusqu'à ce que les types puissent à nouveau être réunis, comme lors d'un `ret`). Les instructions sont ensuite vérifiées une fois avec chaque état distinct. Cela permet donc d'éliminer le problème du polymorphisme et du flot de contrôle dynamique. L'utilisation du critère des adresses de retour différentes pour identifier les « contours » rend cette technique plus puissante que les autres précédemment présentées. L'inconvénient est qu'elle est aussi exponentielle quant au nombre d'états possibles pour du code avec des sous-routines, alors que notre approche croît de façon linéaire par rapport au nombre de sous-routines.

L'utilisation d'attributs ajoutés à une classe pour effectuer la vérification est certainement influencée par les travaux de Necula [Necu97] qui développa la technique de code-certifié (*proof-carrying code*). Son travail ne porte pas sur la vérification de la sûreté des types du code Java, tel que nous l'avons jusqu'ici décrite. Il se concentre à prouver certaines propriétés de programmes au bas niveau (borner la consommation de mémoire par exemple). Il propose d'encoder en notation de lambda-calcul typé les preuves du respect de

certaines propriétés par le code d'un programme. Les preuves sont calculées par un compilateur certifiant et ajoutées au code assembleur du programme. C'est donc un nouveau service du système d'exploitation qui va vérifier les preuves.

Une idée similaire a été développée par Rose [RosR98] [Rose02] et notre travail sur les certificats (chapitre 4) est influencé par ces ouvrages. Avec la vérification du code-octet « poids-léger » (*ligthweight bytecode verification*), elle propose d'encoder le résultat d'une analyse de flot de données dans le fichier *class* dans le but d'alléger la charge de travail pour le client. Cela est particulièrement utile pour les appareils avec des ressources limitées (les téléphones portables par exemple). Sa technique de vérification procède en deux phases. La pré-vérification, lors de la compilation du code, rend les sous-routines *inline*. Cela consiste à remplacer les `jsr` et `ret` par le code de la sous-routine invoquée. Des attributs `StackMap` sont ajoutés aux méthodes, encodant les types pour la pile et les variables locales pour les instructions au début de chaque bloc de base. L'analyse de flot de données calculant ces types est très similaire à l'approche « standard », telle que présentée par Sun. La deuxième phase est la vérification qui s'effectue en itérant séquentiellement sur le code et en vérifiant que toutes les instructions puissent s'exécuter en prenant l'information sur les types dans le certificat lorsqu'au début d'un bloc de base et en les propageant. Deux problèmes peuvent survenir en rendant les sous-routines *inline*. D'une part, cela peut potentiellement causer une croissance exponentielle de la taille du code en présence de multiples sous-routines, tel qu'expliqué par Leroy [Lero01]. D'ailleurs, l'algorithme d'*inlining* n'est pas exposé dans les travaux de Rose [RosR98] [Rose02]. D'autre part, la modification du code d'une méthode peut en changer la sémantique et briser la rétrocompatibilité.

L'algorithme « poids-léger » avec les attributs `StackMap` a été reprit par Sun avec les technologies CLDC[1] et la machine virtuelle KVM[2] et, plus récemment, ces concepts ont été intégrés dans la version 1.6 de Java. Nous savons que l'algorithme de vérification de java 1.6 utilise un format de certificat associé avec un algorithme de compression de données afin d'en réduire la taille et que les sous-routines sont rendues *inline*. Cette technologie fera certainement référence, mais sa nouveauté ne nous permet pas de nous prononcer à son sujet.

Stärk [StaS03] propose un formalisme pour effectuer la vérification du code-octet qui considère des liens entre le code source et le code-octet et entre le compilateur et le vérificateur. L'analyse de flot de données est similaire à celles présentées au début de ce chapitre. Une sous-routine est définie comme une séquence d'instructions dont la première instruction est un `astore` et la dernière un `ret`. Elle est bien sûr invoquée par un `jsr`. Il ne peut donc y avoir de branchement faisant sortir de la sous-routine mis à part un `ret` ou un gestionnaire d'exception (mais ce gestionnaire d'exception ne doit pas contenir un chemin d'exécution menant ensuite au `ret` associé à la sous-routine). Cela règle donc le problème du flot dynamique des sous-routines car cette simplification fait qu'il est facile à identifier. Stärk propose aussi que toutes les variables locales d'une méthode soient initialisées au début de la méthode. Le compilateur doit donc générer du code supplémentaire en ce sens. Cela permet de réduire les erreurs d'analyse dues au polymorphisme en entrée des sous-routines (dans le cas où une variable est initialisée pour un appelant et ne l'est pas pour un autre) mais ne couvre pas tous les cas (comme lorsque une variable est initialisée avec des types incompatibles pour différents appelants).

Stärk propose aussi d'ajouter un certificat associant à une instruction l'état des types des variables, tel que calculé par l'analyse de flot de données. Encore une fois, notre

1 Connected Limited Device Configuration. http://java.sun.com/products/cldc/

2 The K virtual machine. http://java.sun.com/products/cldc/wp/

technique présentée au chapitre 4 en est très proche. Cependant, il n'expose pas de formalisme précis d'encodage du certificat et pas d'expérimentation investiguant la faisabilité d'une telle approche.

CHAPITRE VII

TRAVAUX FUTURS ET CONCLUSIONS

Nous présentons dans ce dernier chapitre des propositions d'améliorations pouvant être apportées au format de certificat ainsi qu'aux algorithmes présentés au chapitre 4, suivi d'un résumé des objectifs, des contributions et des principaux résultats présentés dans ce mémoire.

7.1 Travaux futurs

Concernant le format du certificat, il serait sûrement possible d'en diminuer la taille en n'encodant que les blocs d'activation des sous-routines potentiellement « actives » pour une instruction au lieu de tous les encoder dans toutes les entrées du certificat, comme cela est présentement le cas dans l'attribut `TypeStateMapTable`. Une sous-routine est « active » à un point donné d'une méthode s'il existe un chemin atteignant cette instruction sur lequel un `jsr` est présent mais sans `ret` associé (autrement dit, une sous-routine a été invoquée mais aucune instruction `ret` n'a encore causé sa fin). Ces changements permettraient probablement de diminuer grandement la taille des certificats pour les méthodes utilisant des sous-routines, puisqu'une sous-routine n'est pas, en temps normal, active pour toute une méthode.

D'autre part, considérant le bon taux de compressibilité des certificats (légèrement supérieur aux classes compilées avec *javac* versions 1.5 ou 1.6), l'incorporation d'un algorithme de compression de données à même le calcul de certificat permettrait d'être indépendant d'un outil de compression externe afin d'optimiser la taille des fichiers utilisant les nouveaux attributs.

7.2 Conclusions

Nous avons présenté dans ce mémoire une technique de vérification du code-octet, processus favorisant la sécurité des systèmes distribués utilisant le format de fichier *class* pour structurer les programmes. L'objectif premier était de présenter un formalisme et des algorithmes permettant d'effectuer la vérification par code-certifié de l'entièreté du jeu d'instructions en code-octet et d'évaluer le coût en espace mémoire de cette technique.

Notre technique supporte notamment les sous-routines (en réglant les problèmes de polymorphisme et de dynamisme du flot de données qu'elles peuvent occasionner), l'héritage multiple des interfaces et les contraintes de sous-type et l'initialisation d'objets. L'algorithme de vérification que nous proposons est aussi avantageusement économique comparativement à l'algorithme « standard » de Sun, étant de complexité linéairement proportionnelle au nombre d'instructions d'une méthode.

En particulier, nous avons présenté une structure de certificat permettant d'encoder suffisamment d'information pour réaliser la vérification du code-octet. Nous avons également présenté, un formalisme et un algorithme d'analyse de flot de données polyvariant permettant d'estimer de façon précise le comportement d'une méthode avec un algorithme pour extraire l'information nécessaire à la vérification et l'encoder dans des certificats. Ces certificats sont ajoutés de façon transparente à des classes existantes en utilisant les

structures d'attributs de classe et de méthode du format de fichier *class*. Enfin, nous avons exposé un algorithme permettant la vérification de classes à l'aide de ces certificats.

D'autre part, l'implémentation d'un vérificateur, en Java, basé sur nos théories nous a permis de valider la justesse de celles-ci et d'en mesurer le comportement lorsqu'appliquées à de grandes applications. Le vérificateur a été testé avec succès sur plus de 35 000 classes valides et a montré de très bonnes performances. En effet, l'ajout de certificats à des classes existantes cause en moyenne une augmentation inférieure à 12% de la taille originale du fichier. Lorsque l'on considère la certification d'une application dans son ensemble, l'augmentation moyenne est entre 6% et 10%. De plus, une cinquantaine de classes comportant du code-octet sciemment erroné ont été utilisées pour tester la validité du vérificateur et des théories. Ces classes étaient modifiées et des erreurs ont été insérées dans le format général de la classe, dans le code de méthodes et dans le format des certificats générés par notre algorithme. Toutes ces erreurs ont été détectées par le vérificateur.

Ces résultats indiquent que l'utilisation de cette technique dans des contextes industriels ou de recherche académique est réaliste. Son amélioration serait de plus justifiée dans le but d'atteindre des performances optimales. Ainsi pourrait-on, lors de l'analyse de flot de données utilisée pour calculer les certificats, ne mémoriser que les noeuds avec plus d'un prédécesseur dans le graphe. Cela réduirait la taille du graphe, optimisant par conséquent la consommation mémoire de l'algorithme et pourrait aussi, éventuellement, accélérer son exécution.

GLOSSAIRE

API. De l'anglais *Application Programming Interface*. Ensemble de bibliothèques logicielles standardisant le développement d'applications et permettant d'automatiser certains processus.

Bloc d'activation. Structure de données représentant la pile et les variables locales de la machine virtuelle et utilisé pour l'analyse de flot de données.

Bottom. Borne inférieure d'un treillis dont la valeur est un ensemble vide.

Class. Classe. En langage orienté objet, structure de données définissant des variables et des méthodes pour un ensemble d'objets. Aussi, format de fichier structurant le code-octet et représentant une classe en orienté objet.

Code mort. Instruction d'un programme qui n'est atteignable par aucun chemin d'exécution et qui ne sera jamais exécutée.

Code-octet. Langage intermédiaire utilisé dans le format de fichier class.

Constant pool. Table de symboles encodée dans le fichier class.

Emplacement. Élément d'un bloc d'activation; correspond à un ensemble de types dans l'analyse de flot de données.

Exception handler. Mécanisme de gestion d'exception dans le code-octet.

Frame. Bloc d'activation.

Globale. Se dit d'une analyse portant sur toutes les instructions d'une méthode.

In. Dans l'analyse de flot de données, les données en entrée d'un noeud.

Inlining. Remplacement des appels à un fragment de code par la duplication et l'inclusion du code de ce fragment.

Intra-procédurale. Se dit d'une analyse portant sur le code d'une seule méthode.

Lattice. Treillis.

Local variable. Variable locale à une méthode.

Out. Dans l'analyse de flot de données, les données en sortie d'un noeud.

Paquetage. De l'anglais *package*. En programmation orientée objet, ensemble de classes logiquement reliées.

Pile. Pile d'opérandes d'une méthode.

Point-fixe. État stabilisé (ou résultat) des informations calculées par une analyse de flot de données.

Polymorphisme. Multiplicité de types dans les emplacements d'un bloc d'activation dû à l'invocation d'une sous-routine par plusieurs instructions `jsr`.

Polyvariant. Calcul de plusieurs états simultanés pour un même noeud du graphe durant l'analyse de flot de données.

Registre. Synonyme de variable locale.

Sous-routine. Chemin d'exécution entre une cible d'instruction `jsr` et une instruction `ret`.

Stack. Pile.

Top. Borne supérieure d'un treillis; ensemble de tous les types produits par une méthode.

Treillis. Structure algébrique représentant un ensemble dont les éléments sont partiellement ordonnés et dont toute paire d'éléments est bornée inférieurement et supérieurement.

Utf-8. (*8-bit Unicode transformation format*). Standard de représentation de caractères. Normalisé par ISO/IEC 10646-1:2000 Annexe D [ISO03].

Variables locales. Variables d'une méthode.

Worklist. Structure de données correspondant à un ensemble ordonné.

BIBLIOGRAPHIE

[AppP02] Appel, Andrew, et W., Palsberg, Jens. *Modern compiler implementation in Java, second edition.* Cambridge University Press, Cambridge, Royaume-Uni, 2002.

[Ashes] Suite de bancs d'essais Java réunis par le groupe de recherche Sable. http://www.sable.mcgill.ca/ashes/

[Cogl02] Coglio, Alessandro. *Simple verification technique for complex Java bytecode subroutines.* Dans « 4th ECOOP Workshop on Formal Techniques for Java-like Programs », 2002.

[Eclips] Environnement de développement intégré développé comme projet logiciel libre. http://www.eclipse.org/

[FreM99] Freund, Stephen N., et Mitchell, John C. *A formal framework for the java bytecode language and verifier.* Dans « OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications », pages 147-166, New York, 1999. Presses de l'ACM.

[Gagn02] Gagnon, Etienne. *A Portable Research Framework for the Execution of Java Bytecode.* Thèse de doctorat, Université McGill, décembre 2002.

[GnuCP] *Gnu-Classpath.* Projet logiciel libre. http://www.gnu.org/software/classpath/

[HagT98] Hagiya M., et Tozawa A. *On a new method for dataflow analysis of Java Virtual Machine subroutines.* Dans SIG-Notes, PRO-17-3, pages 13-18. Information Processing Society of Japan, 1998.

[HenS01] Henrio, Ludovic et Serpette, Bernard. *A framework for bytecode verifiers: Application to intra-procedural continuations*. Rapport de recherche, INRIA, 2001.

[Jasmin] Assembleur de code-octet pour les fichiers au format *class*. http://jasmin.sourceforge.net/

[JdkX] Plate-forme de développement Java, versions 1.3.1 à 1.6. http://java.sun.com/javase/index.jsp

[Jikes] Compilateur Java en code source libre. http://jikes.sourceforge.net/

[Kild73] Kildall, Gary. *A Unified Approach to Global Program Optimization. Dans Conference Record of the ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages,* Boston, MA, Octobre 1973.

[Lero01] Leroy, Xavier. *Java Bytecode Verification: An Overview*. Dans : Computer Aided Verification, 13th International Conference, CAV 2001, Vol. 2102 of LNCS. pages 265-285, G. Berry, H. Comon, et A. Finkel, 2001.

[LinY96] Lindholm, Tim, et Yellin, Frank. *The Java Virtual Machine Specification, première édition*. http://java.sun.com/docs/books/vmspec, 1996.

[LinY99] Lindholm, Tim, et Yellin, Frank. *The Java Virtual Machine Specification, deuxième édition*. http://java.sun.com/docs/books/vmspec, 1999.

[Much97] Muchnick, Steven S. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, États-Unis, 1997.

[Necu97] Necula, George C. *Proof-carrying code*. Dans Proceedings of the 24th ACM Symposium on Principles of Programming Languages, pages 106-119, Paris, janvier 1997.

[NiPW02] Nipkow, T., Paulson, L. C., et Wenzel, M. *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*, Vol. 2283 of LNCS. Springer, 2002.

[PosV98] Posegga, Joachim, et Vogt, Harald. *Java bytecode verification using model checking*. Dans Workshop Fundamental Underpinnings of Java, 1998.

[Quia00] Qian, Zhenyu. *Standard fixpoint iteration for Java bytecode verification.* ACM Transactions on Programming Languages and Systems, vol.22 no.4, pages 638-672, 2000.

[RosR98] Rose, Eva, et Rose, Kristoffer. *Lightweight bytecode verification*. Dans OOPSLA Workshop on Formal Underpinnings of Java, 1998.

[Rose02] Rose, Eva. *Vérification de code d'octet de la machine virtuelle Java: formalisation et implantation*. Thèse de doctorat, Université de Paris, septembre 2002.

[SaVM06] Lazaar, Jamal, Corbeil, Mathieu, et Gagnon, Etienne M. *A Sound Java Bytecode Verifier.* Rapport Technique numéro 1. Groupe de recherche SableVM, Université du Québec à Montréal, novembre 2006.

[SablCC] Compilateur de compilateurs, logiciel libre par le groupe de recherche Sable. http://sablecc.org/

[StaA99] Stata, Raymie, et Abadi, Martin. *A type system for java bytecode subroutines*. ACM Transactions on Programming Languages and Systems, vol.21 no.1, pages 90-137, 1999.

[StaS03] Stark, Robert F., Schmid, Joachim. *Completeness of a bytecode verifier and a certifying java-to-jvm compiler. Dans Journal of Automated Reasoning, Hingham, volume 30, numéros 3-4,* MA, États-Unis, 2003.

[Soot] Groupe de recherche Sable. Compilateur optimisant pour Java, logiciel libre. http://www.sable.mcgill.ca/soot/

[ISO03] *International Standard ISO/IEC 10646, Information technology – Universal Multiple-Octet Coded Character Set (UCS).* Troisième édition, Organisation internationale de normalisation, Genève, 2003.

APPENDICE A

TYPES DE VÉRIFICATION POUR CODE-CERTIFIÉ

```
Verification_Type_Int {
     u1 tag = 1;
}

Verification_Type_Byte {
     u1 tag = 2;
}

Verification_Type_Char {
     u1 tag = 3;
}

Verification_Type_Short {
     u1 tag = 4;
}

Verification_Type_Boolean {
     u1 tag = 5;
}
```

```
Verification_Type_Float {
     u1 tag = 6;
}

Verification_Type_Double {
     u1 tag = 7;
}

Verification_Type_Double2 {
     u1 tag = 8;
}

Verification_Type_Long {
     u1 tag = 9;
}

Verification_Type_Long2 {
     u1 tag = 10;
}

Verification_Type_Null {
     u1 tag = 11;
}

Verification_Type_SRAddress {
     u1 tag = 12;
     u4 offset;
}
```

```
Verification_Type_Uninitref {
     u1 tag = 13;
     u2 name_index;
}

Verification_Type_Extra_Uninitref {
     u1 tag = 14;
     u2 name_index;
}
Verification_Type_UninitThis {
     u1 tag = 15;
}

Verification_Type_Reference {
     u1 tag = 16;
     u2 name_index;
}

Verification_Type_Extra_Reference {
     u1 tag = 17;
     u2 name_index;
}

Verification_Type_Array {
     u1 tag = 18;
     u1 dimensions;
     verification_type base_type;
}
```

```
Verification_Type_Uninitialized {
     u1 tag = 19;
}

Verification_Type_Contextual_Local_Entry {
     u1 tag = 20;
     u4 entry_index;
}

Verification_Type_Contextual_Stack_Entry {
     u1 tag = 21;
     u4 entry_index;
}
```