ASM : un outil de manipulation de code pour la réalisation de systèmes adaptables

Eric Bruneton, Romain Lenglet, Thierry Coupaye

France Télécom R&D, DTL/ASR
28, chemin du Vieux Chêne
BP 98, 38243 MEYLAN cedex
{Eric.Bruneton,Romain.Lenglet,Thierry.Coupaye}@rd.francetelecom.com

RÉSUMÉ. *ASM est un outil de manipulation de classes Java conçu pour la génération et la manipulation dynamiques de code, qui sont des techniques très utiles pour la réalisation de systèmes adaptables. ASM est basé sur une approche originale, par rapport aux outils existants équivalents, qui consiste à utiliser le patron de conception « visiteur »* sans *représenter explicitement l'arborescence visitée sous forme d'objets. Cette nouvelle approche permet d'obtenir des performances bien supérieures à celles des outils existants, pour la plupart des besoins courants.*

ABSTRACT. *ASM is a Java class manipulation tool designed to dynamically generate and manipulate Java classes, which are useful techniques to implement adaptable systems. ASM is based on a new approach, compared to equivalent existing tools, which consists in using the "visitor" design pattern* without *explicitly representing the visited tree with objects. This new approach gives much better performances than those of existing tools, for most of practical needs.*

MOTS-CLÉS : *génération, transformation, dynamique, classe, code, Java, visiteur, adaptabilité.*

KEYWORDS: *generation, transformation, dynamic, class, code, Java, visitor, adaptability.*

1. Introduction

La plupart des techniques utilisées pour mettre en œuvre des systèmes dynamiquement adaptables sont basées sur une forme ou une autre d'interposition, qu'il s'agisse d'objets d'interposition, de méta-objets... L'interposition permet en effet non seulement de modifier la sémantique de base des composants, pour leur ajouter des propriétés fonctionnelles ou non fonctionnelles, mais aussi d'ajouter et de supprimer dynamiquement ces propriétés additionnelles.

Or les techniques d'interposition nécessitent à leur tour des outils pour générer du code ou pour modifier du code existant. En effet un objet d'interposition doit avoir le même type que celui de l'objet qu'il masque, ce qui requiert de générer du code spécifique à chaque type d'objet d'interposition. De même, la réalisation d'un protocole à méta-objets 100% Java passe généralement soit par l'utilisation d'objets d'interposition, soit par la modification du code des classes de base, pour y introduire le code réifiant les événements souhaités.

Les outils de génération et de manipulation de code sont donc utiles pour la mise en œuvre de systèmes adaptables. On peut les utiliser statiquement ou dynamiquement, la solution dynamique (i.e., invoquer un compilateur comme `rmic` à la volée, par programme) étant d'ailleurs la plus pratique pour l'utilisateur. La solution dynamique est même indispensable pour les systèmes ouverts, où de nouveaux types d'objet, non prévisibles à l'avance, peuvent être chargés dynamiquement : il faut alors générer du code d'interposition, ou modifier le code nouvellement chargé, *à la volée*.

Une solution pour générer dynamiquement du code serait de générer dynamiquement du code source, puis de compiler ce code, toujours dynamiquement. Mais cette solution dégraderait beaucoup les performances des applications, et aussi leur taille (puisqu'elles devraient inclure un compilateur comme `rmic` ou `javac` complet). Il est donc nécessaire d'utiliser un outil aussi petit et rapide que possible. Malheureusement, selon nous, les outils existants ne sont pas suffisamment petits et rapides. Nous avons donc développé un nouvel outil, appelé ASM[1], basé sur une nouvelle approche, afin d'atteindre cet objectif.

L'un des premiers principes pour optimiser les performances d'une application est d'optimiser le code le plus fréquemment exécuté, autrement dit de se concentrer sur les cas d'utilisation les plus courants. Le deuxième objectif d'ASM était donc de fournir un outil optimisé en priorité pour les transformations de classes *dynamiques* les plus courantes. C'est à dire, selon nous, mais aussi d'après [CHI 00], l'ajout, la suppression ou le renommage de champs ou de méthodes, l'interception des accès aux champs d'une classe, l'introduction de code avant ou après le code original d'une méthode...

Enfin, le dernier objectif d'ASM était de fournir un outil permettant de réaliser n'importe quel type de transformations (contrairement à certains outils, comme Javas-

1. Le nom ASM ne signifie rien de particulier : c'est juste une référence au mot clef `__asm__` en C, qui permet de programmer certaines fonctions en assembleur.

sit [CHI 00] ou BCA [KEL 98], qui se limitent à un ensemble de transformations prédéfinies, supposées être les plus courantes). Par ailleurs, la gestion des dépendances entre classes (qui consiste, par exemple, à modifier automatiquement toutes les classes référençant une classe venant d'être renommée) ou entre transformations (comme dans JMangler [KNI 01], qui permet de composer certaines transformations dans le « bon » ordre), *n'était pas* un objectif.

Le reste de cet article est organisé comme suit. Après une présentation succincte du format des classes Java, et des problèmes qui se posent pour les manipuler, dans la section 2, nous présentons les approches existantes (section 3), puis l'approche que nous proposons (section 4). Nous présentons également des mesures de performances en section 5, avant de conclure.

2. Le format des classes Java

2.1. *Structure générale*

La structure générale d'une classe Java au format binaire [LIN 99] est une structure arborescente. La racine de cette structure contient les nœuds suivants (* signifie 0 ou plus) :

– `ConstantPool` : table des symboles (cf. section 2.2) ;

– `FieldInfo*` : descriptions des champs de la classe ;

– `MethodInfo*` : descriptions des méthodes de la classe ;

– `Attribute*` : nœuds optionnels supplémentaires.

Un nœud `FieldInfo` contient le nom d'un champ, son type, ainsi que des drapeaux indiquant si le champ est public, privé, statique... Un nœud `FieldInfo` peut également contenir des *attributs* optionnels supplémentaires (un attribut est un nœud dont le type et la taille sont indiqués explicitement). L'attribut `ConstantValue` permet d'indiquer la valeur d'un champ statique constant. Les attributs `Deprecated` et `Synthetic` permettent d'indiquer qu'un champ est obsolète, ou qu'il a été ajouté par le compilateur (et ne correspond donc pas à un champ dans le code source de la classe).

Un nœud `MethodInfo` contient le nom d'une méthode, les types de ses paramètres et de sa valeur de retour, ainsi que des drapeaux indiquant si la méthode est publique, privée, statique, abstraite... Un nœud `MethodInfo` peut également contenir des attributs optionnels supplémentaires. Le plus important est l'attribut `Code` qui contient le code des méthodes non abstraites (cf. section 2.3). L'attribut `Exceptions` contient la liste des exceptions que peut lancer la méthode. Les attributs `Deprecated` et `Synthetic` ont le même rôle que pour les champs.

Les attributs optionnels du nœud racine sont l'attribut `SourceFile`, pour stocker le fichier source à partir duquel une classe a été compilée, l'attribut `InnerClasses`, pour stocker les noms des classes internes d'une classe, et enfin l'attribut `Deprecated`. Les attributs permettent de rendre extensible le format des classes Java. On peut en effet

ajouter des informations dans une classe Java en lui ajoutant des attributs non standard, sans que cela affecte la possibilité de charger cette classe dans une machine virtuelle ne connaissant pas ces attributs (les machines virtuelles doivent en effet simplement ignorer les attributs qu'elles ne reconnaissent pas, ce qui est possible grâce au fait que chaque attribut contient son type et son taille).

2.2. *La table des symboles*

La table des symboles d'une classe, ou `ConstantPool`, est un tableau contenant un ensemble de constantes de type numérique ou chaîne de caractères, ou d'un type construit à partir de ces types de base, qui peuvent être référencés, via leur index dans ce tableau, par d'autres nœuds de la structure arborescente d'une classe.

Le but de cette table est d'éviter les redondances. Ainsi, un nœud `FieldInfo` ne contient pas directement le nom d'un champ, mais l'index de ce nom dans la table des symboles. De même, les instructions `GETFIELD` et `PUTFIELD` ne contiennent pas directement le nom du champ qui doit être lu ou modifié, mais l'index de ce nom. Ainsi le nom du champ est stocké une et une seule fois, même si de nombreuses instructions y font référence.

2.3. *Le code des méthodes*

Le code d'une méthode est stocké dans un attribut `Code`. Ce code est une suite linéaire d'instructions élémentaires basées principalement sur la notion de pile, bien qu'une notion de *variable locale*, similaire à la notion de registre, soit également disponible. La plupart des instructions sont codées sur un seul octet et n'ont pas d'argument (d'où le nom d'instruction de *bytecode*).

L'attribut `Code` contient également une table des traitants d'exceptions, qui correspondent aux blocs `try catch finally` de Java, ainsi qu'un entier indiquant la taille maximale que peut atteindre la pile lors de l'exécution du code de la méthode. Enfin, l'attribut `Code` peut également contenir des sous attributs optionnels supplémentaires. Ainsi l'attribut `LineNumberTable` stocke des informations permettant de retrouver, pour chaque instruction de bytecode, le numéro de ligne correspondant dans le fichier source. L'attribut `LocalVariableTable` stocke quant à lui les noms des paramètres formels de la méthode ainsi que les noms des variables locales, tels que déclarés dans le code source.

2.4. *Problèmes principaux pour la manipulation des classes*

Le problème principal pour manipuler une classe Java est qu'une classe compilée se présente sous la forme d'un tableau d'octets, qu'il est quasiment impossible de modifier directement. La simple lecture de ce tableau, pour analyser la structure

arborescente de la classe, est déja une tâche fastidieuse (il faut en effet suivre scrupuleusement les nombreux détails du processus permettant d'encoder la structure arborescente présentée ci-dessus sous la forme d'un tableau d'octets).

Un autre problème est de maintenir à jour les références vers la table des symboles. Par exemple, si on supprime ou si on insère une constante dans cette table, les index des constantes suivantes sont décalés d'une unité, et il faut donc mettre à jour toutes les références à ces constantes dans le reste de la classe. De même, suite aux modifications effectuées dans le reste de la classe, certaines constantes peuvent devenir inutilisées, et pourraient donc être supprimées de la table des symboles, mais il n'est pas toujours simple de s'en apercevoir.

Un autre problème est de maintenir à jour les adresses d'instruction, relatives ou absolues, contenues dans certaines instructions comme `GOTO` et `IFEQ`, et dans certains attributs comme `LineNumberTable` et `LocalVariableTable`. En effet ces adresses peuvent changer si on insère ou si on supprime des instructions dans le code d'une méthode (et le problème est encore compliqué par le fait que toutes les instructions n'ont pas la même taille).

Un dernier problème est de mettre à jour l'entier contenu dans l'attribut `Code` qui indique la taille maximale que la pile peut atteindre lors de l'exécution d'une méthode. En effet cette taille maximale est une propriété globale du code de toute la méthode et, dans le cas général, suite à une modification de ce code, la nouvelle taille maximale ne peut pas être calculée en se basant seulement sur l'ancienne taille et sur la modification effectuée : il faut au contraire réexaminer le code complet de la méthode, à l'aide d'un algorithme d'analyse de flot de contrôle.

3. Approches existantes

Cette section présente les principaux outils de manipulation de classes Java existants, *parmi ceux qui permettent de réaliser n'importe quel type de transformations*, et, pour chaque outil, présente les solutions qu'il apporte aux problèmes présentés dans la section 2.4.

3.1. *BCEL*

BCEL [DAH 99] est l'outil de manipulation de classes Java qui semble être actuellement le plus utilisé (au moins une trentaine de logiciels l'utilisent). Avec cet outil la modification d'une classe se fait en trois phases. La première phase consiste à désérialiser le tableau d'octets représentant la classe, et à construire une représentation objet de la structure arborescente correspondante. Un objet est donc créé pour chaque nœud de la structure, jusqu'au niveau des instructions de bytecode (il y a donc un objet Java par instruction de bytecode). La seconde phase consiste à modifier la représentation objet construite précédemment. Enfin, la dernière phase consiste à sérialiser ce

graphe d'objets en un nouveau tableau d'octets. La création d'une classe *ex nihilo* est similaire : il suffit de créer sa représentation objet, puis de sérialiser cette structure.

Cette approche résout les problèmes de sérialisation et de désérialisation, puisque ces fonctions sont fournies une fois pour toutes par l'outil lui même. L'utilisateur ne manipule en effet que la représentation objet des classes, et tous les détails de sérialisation lui sont complètement cachés.

Cette approche pourrait également permettre de résoudre les problèmes de gestion de la table des symboles mais, curieusement, BCEL n'offre pas beaucoup d'aide dans ce domaine. En effet l'utilisateur doit indiquer explicitement les index des constantes dans cette table, et doit mettre à jour ces index « à la main » s'ils deviennent invalides (en pratique cela ne se produit pas souvent car BCEL n'offre pas la possibilité de supprimer une constante de la table des constantes. Comme en plus les nouvelles constantes sont ajoutées en fin de table, les index existants restent valides - sauf si on remplace la table des constantes par une table complètement nouvelle, ce qui est la seule manière pour supprimer des constantes devenues inutiles !)

BCEL résout par contre le problème des adresses d'instructions de façon satisfaisante. En effet les adresses relatives ou absolues, en nombre d'octets, sont remplacées par des références aux objets Java représentant les instructions, au moment de la phase de désérialisation. Ces références ne sont reconverties en adresses qu'au moment de la sérialisation. L'utilisateur (et l'outil lui même) n'a donc pas à se préoccuper de mettre à jour ces adresses à chaque fois qu'une instruction est supprimée ou insérée.

Enfin, BCEL fournit une fonction pour calculer la taille de pile maximale d'une méthode. Cette fonction, basée sur un algorithme d'analyse de flot de contrôle, doit être appelée une fois toutes les modifications voulues effectuées sur une méthode, juste avant la phase de sérialisation.

3.2. *SERP*

SERP [WHI] utilise une approche similaire à celle de BCEL, basée sur une représentation objet des classes à modifier. Cependant SERP utilise moins de classes que BCEL pour représenter les quelque 200 types d'instructions de bytecode, ce qui conduit à un outil plus petit en terme de taille de code (par exemple SERP utilise une seule classe `MathInstruction` pour représenter toutes les instructions arithmétiques, alors que BCEL définit les classes `IADD`, `ISUB`, `IMUL`, `DADD`, `DSUB`...).

SERP résout donc les problèmes de la section 2.4 de la même façon que BCEL : les fonctions de sérialisation et désérialisation sont fournies une fois pour toutes par l'outil, les adresses d'instruction sont remplacées par des références aux objets Java représentant les instructions, et l'outil offre également une fonction pour calculer la taille de pile maximale d'une méthode. La seule différence est que SERP résout mieux que BCEL les problèmes de gestion de la table des symboles. En effet l'utilisateur peut spécifier l'argument d'une instruction en donnant directement sa valeur, au lieu

de donner son index dans la table des symboles (ce qui nécessiterait, au préalable, d'ajouter manuellement cette valeur dans la table). La gestion de la table des symboles est donc complètement cachée à l'utilisateur[2]. Par contre SERP ne supprime pas automatiquement de cette table les constantes inutilisées.

3.3. *JOIE*

BCEL et SERP sont très similaires à JOIE [COH 98], qui fut l'un des premiers outils de ce type pour le langage Java. En effet, d'après [COH 98], JOIE utilisait déja une représentation objet des classes à transformer, jusqu'au niveau des instructions de bytecode, et apportait les mêmes solutions que ces outils, notamment en ce qui concerne les adresses d'instructions. Mais, à notre connaissance, cet outil n'est plus maintenu. En fait nous n'avons pas réussi à trouver un site permettant de le télécharger, aussi n'avons nous pas pu l'examiner plus en détails.

4. Approche proposée

4.1. *Idée générale*

L'idée centrale d'ASM est de ne *pas* utiliser une représentation objet en mémoire des classes à manipuler. En effet, ceci requiert la définition d'un grand nombre de classes, correspondant aux différents types de nœuds de la structure arborescente des classes et aux différents types d'instruction de bytecode (SERP contient environ 80 classes dévolues à cette tâche, contre 270 pour BCEL), ce qui conduit nécessairement à des outils assez gros en terme de taille de code. De plus, cette approche conduit à créer un grand nombre d'objets en mémoire lors de la désérialisation d'une classe, et ceci prend beaucoup de temps et de mémoire. Une telle approche n'était donc pas utilisable pour ASM, dont le but était de fournir un outil aussi petit et rapide que possible.

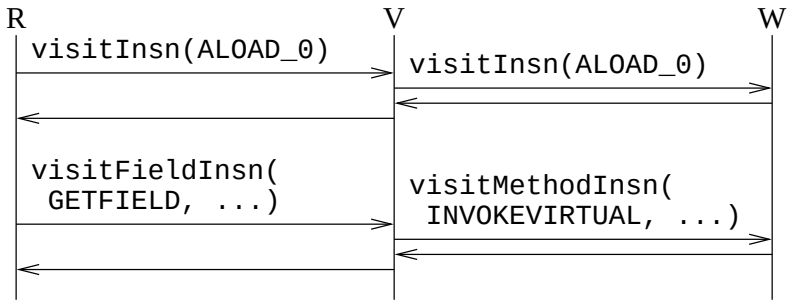
Mais comment offrir une interface de manipulation de classes de « haut niveau », c'est à dire qui ne reste pas au niveau des octets, sans utiliser de représentation objet des classes ? La solution choisie dans ASM consiste à utiliser le patron de conception appelé *visiteur* [GAM 95], comme dans de nombreux outils de transformation (dont BCEL, SERP, OpenJava [TAT 00] . . .), mais *sans* représenter explicitement l'arborescence visitée sous forme d'objets. On peut en effet utiliser ce patron :

– pour parcourir une version sérialisée d'un graphe d'objet, sans le désérialiser complètement (c'est à dire en désérialisant uniquement les types de base, numériques ou de type chaînes de caractères) ;

2. En fait SERP offre également des fonctions de bas niveau permettant de manipuler directement les index et la table des symboles elle même.

– pour modifier une structure existante ou, plus précisément, pour *reconstruire* une version modifiée de cette structure (et que l'on peut construire directement sous une forme sérialisée, sans construire de représentation objet intermédiaire).

Pour illustrer cette idée, montrons comment on peut transformer à l'aide de visiteurs la séquence d'instructions `ALOAD_0` `GETFIELD` *f*, qui lit la valeur du champ *f* de `this`, en `ALOAD_0` `INVOKEVIRTUAL` *getf*, qui invoque la méthode *getf* à la place. Il faut pour cela utiliser trois objets : un analyseur de classe *R*, qui fait visiter la classe à un visiteur *V* (on peut aussi utiliser plusieurs visiteurs intermédiaires, pour composer des transformations de code indépendantes), qui lui même délègue à un autre visiteur *W* qui reconstruit la classe au fur et à mesure :



R analyse la séquence d'instructions originale, qui se présente pour lui sous la forme d'un tableau d'octets, et appelle les méthodes appropriées de *V* au fur et à mesure de l'analyse. Ici cela conduit à l'appel de la méthode `visitInsn` avec l'*entier* `ALOAD_0` en paramètre, puis à l'appel de la méthode `visitFieldInsn` avec en paramètres l'entier `GETFIELD` et trois `String` représentant le nom de classe contentant le champ *f*, ainsi que le nom et le type du champ *f*.

V se contente d'appeler sur *W* toutes les méthodes qu'on appelle sur lui, avec les mêmes paramètres, sauf pour la méthode `visitFieldInsn`. Dans ce cas, si l'instruction est égale à `GETFIELD`, *V* appelle `visitMethodInsn` sur *W* (sinon il appelle `visitFieldInsn`).

Quant à *W*, il reconstruit la classe au fur et à mesure, directement sous la forme d'un tableau d'octets. Ainsi, quand *V* appelle sa méthode `visitInsn` avec `ALOAD_0` en paramètre, *W* se contente d'ajouter, à la fin du tableau en cours de construction, l'octet désignant cette instruction. De même quand *V* appelle `visitMethodInsn`, sauf qu'ici l'instruction requiert 3 octets.

Le patron visiteur permet donc bien de manipuler le code des méthodes, et plus généralement la structure complète d'une classe, et ce sans créer en mémoire un objet par instruction. Il permet donc bien d'atteindre notre premier objectif (cf. section 1), qui était d'obtenir un outil aussi petit et rapide que possible. Il permet également d'atteindre le troisième objectif, qui était d'offrir un outil permettant de réaliser n'importe quel type de transformations. Cependant le patron visiteur n'est facilement utilisable

que pour des transformations « simples », c'est à dire essentiellement locales et non contextuelles. Pour des transformations plus complexes, comme l'élimination du code « mort », ou la décompilation d'une classe en code source équivalent, il faut utiliser des visiteurs plus complexes qui, d'une manière ou d'une autre, construisent une représentation objet des classes qu'ils visitent, transforment cette structure, puis la font visiter au visiteur suivant.

L'approche consistant à utiliser le patron visiteur est donc plus efficace que l'approche utilisée dans BCEL ou SERP (cf. section 5), mais *seulement*, a priori, pour les transformations simples, qui sont aussi les plus courantes, et pour lesquelles ASM devait être optimisé en priorité (cf. section 1). Le patron visiteur permet donc d'atteindre tous les objectifs que nous nous étions fixés. Les sections suivantes montrent qu'il permet également de résoudre les problèmes présentés dans la section 2.4.

4.2. *Sérialisation et désérialisation*

Comme dans BCEL ou SERP, tous les détails de sérialisation et de désérialisation sont pris en charge une fois pour toutes par ASM, et sont donc complètement cachés à l'utilisateur. Plus précisément, ASM fournit une classe `ClassReader` permettant d'analyser une classe sous forme binaire et de la faire visiter à un visiteur implantant l'interface `ClassVisitor`, ainsi qu'une classe `ClassWriter`, qui implante `ClassVisitor`, pour construire de nouvelles classes directement sous forme sérialisée.

4.3. *Gestion de la table des symboles*

ASM masque complètement la gestion de la table des symboles. L'utilisateur n'a donc pas à manipuler les index des constantes dans cette table (ainsi, dans l'exemple de la section 4.1, les paramètres de la méthode `visitFieldInsn` sont directement le propriétaire, le nom et le type du champ, sous forme de `String`, et non pas les index de ces constantes dans la table des symboles). De plus, le fait que les classes soient modifiées en les « reconstruisant », au lieu de les modifier « en place », permet d'assurer facilement et automatiquement que la table des symboles ne contient pas de constantes inutilisées.

4.4. *Gestion des adresses d'instruction*

ASM ne peut pas traiter le problème des adresses d'instructions comme BCEL ou SERP, c'est à dire en utilisant des références aux objets représentant les instructions, puisque les instructions ne sont pas représentées sous forme d'objets. ASM utilise à la place des *étiquettes*, représentées par des objets de type `Label`, comme dans les langages d'assemblage. Considérons par exemple la séquence d'instructions ci-dessous (à gauche), qui décrémente la variable locale n°1, charge sa valeur sur la pile,

et revient à la première instruction, 5 octets en arrière, tant que cette valeur n'est pas nulle :

```
IINC 1 -1        l: IINC 1 -1          IINC 1 -1
ILOAD 1             ILOAD 1            ILOAD_1
IFNEQ -5            IFNEQ l            IFNEQ -4
```

La classe `ClassReader` effectue une analyse préliminaire du code pour détecter les instructions de saut, et en déduit les étiquettes qui sont nécessaires (ici `l`). Le `ClassReader` fait ensuite visiter les instructions *et* les étiquettes, en appelant les méthodes `visitLabel(l)` `visitIincInsn(1, -1)` `visitVarInsn(ILOAD, 1)` puis `visitJumpInsn(IFNEQ, l)`. Les adresses d'instructions sont donc cachées à l'utilisateur, tout comme dans BCEL ou SERP. Comme dans ces outils, elles ne sont recalculées que lors de la phase de sérialisation, c'est à dire dans la classe `ClassWriter`. Ici, par exemple, le `ClassWriter` mémorise l'adresse absolue correspondant à `l` lors de la visite de cette étiquette puis, lors de la visite de `IFNEQ`, calcule l'adresse relative correspondante[3] (ici -4 au lieu de -5, car le `ClassWriter` a remplacé automatiquement l'instruction `ILOAD 1` par `ILOAD_1`, qui prend un octet de moins).

4.5. *Calcul des tailles de pile maximales*

Tout comme BCEL et SERP, ASM offre la possibilité de calculer automatiquement la taille de pile maximale d'une méthode. Cette taille maximale est calculée à l'aide d'un algorithme d'analyse de flot de contrôle. Cet algorithme nécessite de décomposer le code des méthodes en *blocs de base* (suite d'instructions sans sauts en entrants ni sortants), de relier ces blocs en un graphe, où un arc représente la possibilité de sauter d'un bloc au début d'un autre, de calculer la taille maximale de pile pour chaque bloc, et enfin de parcourir le graphe pour déterminer la taille maximale de pile globale.

Les blocs de base, représentés par les objets `Label`, ainsi que leur taille maximale de pile associée, sont déterminés au fur et à mesure de la visite des instructions. Le graphe reliant ces blocs est lui aussi calculé au fur et à mesure. La taille maximale de pile globale est calculée en parcourant ce graphe lorsqu'il est complet, c'est à dire quand toutes les instructions ont été visitées.

5. Performances

Cette section compare les performances d'ASM à celle de BCEL et de SERP. Nous n'avons pas comparé ASM à Javassit ou à BCA, qui n'offrent pas le même niveau de fonctionnalités.

3. En cas de référence en avant ces adresses ne peuvent pas être calculées immédiatement. Dans ce cas ASM réserve la place nécessaire pour stocker ces adresses, mémorise les emplacements de ces « trous », et les comble dès que l'étiquette adéquate est visitée.

ASM ne contient que 13 classes et n'occupe que 21K (sous forme de `.jar`), alors que SERP occupe 150K et BCEL 350K (sans compter le vérifieur de code, fonction qui n'est pas fournie par ASM). ASM est donc beaucoup plus petit que ces outils.

Pour mesurer les performances à l'exécution, nous avons mesuré le temps nécessaire pour charger un grand nombre de classes à partir d'un fichier `.jar` (a) avec un `ClassLoader` standard, (b) en désérialisant puis en resérialisant chaque classe telle quelle avant de la charger avec le `ClassLoader`, (c) en recalculant en plus la taille maximale de pile de toutes les méthodes et (d) en ajoutant un compteur dans chaque classe, et des instructions pour incrémenter ces compteurs au début de chaque méthode, avant de resérialiser puis de charger chaque classe. Les résultats sont les suivants (pour charger 1155 classes extraites du fichier `tools.jar` du JDK, d'une taille moyenne de 3K, avec le JDK1.3.1 Hotspot VM, Linux 2.4.9, et sur un Pentium III 1 GHz - mesures valables à plus ou moins 0,01s) :

	Cas a	Cas b	Cas c	Cas d
ASM	1,99s	3,22s	3,29s	3,26s
BCEL	1,98s	16,6s	19,2s	16,8s
SERP	1,98s	24,3s	34,6s	28,0s

ASM n'augmente donc les temps de chargement que de 60% environ, alors que BCEL les augmente d'au moins 700%, et SERP d'au moins 1100% ! Concernant le temps de sérialisation/désérialisation seul, donné par $b - a$, on trouve qu'ASM est 12 fois plus rapide que BCEL et 18 fois plus rapide que SERP. ASM est également 35 (resp. 130) fois plus rapide que BCEL (resp. SERP) pour calculer la taille de pile maximale d'une méthode (temps donné par $c - b$), et 4 (resp. 80) fois plus rapide que BCEL (resp. SERP) pour réaliser la transformation en elle même ($d - b$). Tout ceci montre qu'ASM est effectivement beaucoup plus rapide que BCEL et SERP.

6. Conclusion

Les outils de génération et de manipulation de code sont très utiles pour mettre en œuvre des systèmes adaptables mais, pour être utilisables dynamiquement (ce qui est indispensable pour les systèmes ouverts), ils doivent être aussi petits et rapides que possible. L'approche originale que nous proposons pour atteindre cet objectif, qui consiste à utiliser le patron « visiteur » *sans* représenter l'arborescence visitée sous forme d'objets, et que nous avons implémentée dans ASM, permet effectivement d'atteindre cet objectif, comme le montrent les mesures que nous avons effectuées.

ASM est actuellement utilisé pour la mise en œuvre du modèle à composants Fractal [BRU 02], pour générer dynamiquement les classes des objets d'interposition contenus dans les contrôleurs des composants, mais aussi pour fusionner dynamiquement certaines classes (ce qui nécessite des fonctions de manipulation de code), dans un but d'optimisation. ASM pourrait également être utilisé pour la *composition contractuelle* [COU 01], afin de générer dynamiquement du code optimisé pour

associer des contrats (exprimés sous forme d'assertions, de prédicats dans une logique temporelle...) aux composants. Nous envisageons également d'utiliser ASM pour mettre œuvre ce que nous appelons la *composition opératoire* [COU 01]. Dans ce but, un canevas de composants de transformation de code réutilisables, appelé JABYCE, est en cours de développement. Il sera basé sur les mêmes idées qu'ASM (et réutilisera sans doute les classes `ClassReader` et `ClassWriter`), mais en mettant moins l'accent sur les performances, et plus sur les principes de séparation des interfaces [MAR 96b], de fermeture commune [MAR 96a] et d'ouverture/fermeture [MEY 97] (afin de fournir des *composants* de transformation de classes, autonomes et réutilisables). ASM est disponible sur le site d'ObjectWeb, `www.objectweb.org`.

7. Bibliographie

[BRU 02] BRUNETON E., COUPAYE T., STEFANI J. B., « Recursive and Dynamic Software Composition with Sharing », *Workshop on Component Oriented Programming*, Malaga, Spain, 2002.

[CHI 00] CHIBA S., « Load-time Structural Reflection in Java », *ECOOP 2000 – Object-Oriented Programming*, vol. LNCS 1850, Springer, 2000, p. 313-336.

[COH 98] COHEN G. A., CHASE J. S., KAMINSKY D. L., « Automatic program transformation with JOIE », *USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, USA, 1998.

[COU 01] COUPAYE T., LENGLET R., BEAUVOIS M., BRUNETON E., DÉCHAMBOUX P., « Composants et composition dans l'architecture des systèmes répartis », *Journées Composants*, Besançon, France, 2001.

[DAH 99] DAHM M., « Byte Code Engineering », *Proceedings JIT'99*, Springer, 1999.

[GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns*, Addison-Wesley, 1995.

[KEL 98] KELLER R., HÖLZLE U., « Binary Component Adaptation », *ECOOP 1998 – Object-Oriented Programming*, vol. LNCS 1445, Springer, 1998, p. 307-329.

[KNI 01] KNIESEL G., COSTANZA P., AUSTERMANN M., « JMangler - A Framework for Load-Time Transformation of Java Class Files », *IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*, novembre2001.

[LIN 99] LINDHOLM T., YELLIN F., *The Java Virtual Machine Specification*, Addison-Wesley, 2nd édition, 1999.

[MAR 96a] MARTIN R. C., « Granularity », *C++ Report*, novembre1996.

[MAR 96b] MARTIN R. C., « The Interface Segregation Principle », *C++ Report*, août1996.

[MEY 97] MEYER B., *Object-Oriented Software Construction*, Prentice-Hall, second édition, 1997.

[TAT 00] TATSUBORI M., CHIBA S., KILLIJIAN M.-O., ITANO K., « OpenJava : A Class-based Macro System for Java », *Reflection and Software Engineering*, vol. LNCS 1826, Springer, 2000, p. 119-135.

[WHI] WHITE A., « Serp – http ://serp.sourceforge.net ».