RENPAR'15 / CFSE'3 / SympAAA'2003
La Colle sur Loup, France, 15 au 17 octobre 2003

Jnjvm : Une Plateforme Java Adaptable pour Applications Actives

Gaël Thomas, Bertil Folliot, Frédéric Ogel

Université Paris 6, LIP6/CNRS
4, place Jussieu – F-75252 Paris
gael.thomas,bertil.folliot@lip6.fr
frederic.ogel@inria.fr

Résumé
Le nombre de machines virtuelles Java dédiées à des domaines applicatifs particuliers ne cesse d'augmenter. Chacune de ces machines virtuelles modifie ou enrichit la sémantique de la machine virtuelle standard de Sun [23] pour implanter des mécanismes dédiés. Ces mécanismes ne modifient pas fondamentalement la structure de cette machine virtuelle (ramasse-miettes, JIT, chargeur etc...).
Nous proposons dans cet article une solution alternative : une machine virtuelle java ouverte permettant à une application de spécifier précisément son environnement d'exécution. La partie fonctionnelle de l'application reste écrite en java et les mécanismes non fonctionnels permettent de construire à la volée une machine virtuelle Java adaptée à l'application.

Mots-clés : Adaptabilité, extensibilité, JVM

1. Introduction

Ces dernières années, Java est devenu un standard de fait dans le développement d'applications réparties. Java permet de s'affranchir des difficultés liées à l'hétérogénéité matérielle. Les binaires Java sont portables grâce au bytecode Java et peuvent être exécutés sur toutes plateformes possédant une machine virtuelle Java (JVM). Le bytecode Java est fortement typé et peut être vérifié dans la JVM.
Toutefois, les JVM sont monolithiques ce qui empêche toute adaptation dynamique des mécanismes internes de la machine virtuelle. Elles sont aussi fermées ce qui empêche toute adaptation du système sous-jacent. La rigidité des JVM entraîne un foisonnement de plateformes dédiées à certains types de matériels (comme KVM [1] pour des plateformes à mémoire limitée ou JavaCard [2] pour les cartes à puces) et de domaines applicatifs (comme PJama [22] ou Gemstone [3] pour des JVM à objets persistant, MetaXa [19] pour une JVM reflexive...). La réalisation de ces plateformes est coûteuse en terme de temps de développement et ces plateformes sont elles-mêmes figées et inadaptables à d'autres domaines. Par exemple, bien que le temps réel et la réflexivité soient deux domaines orthogonaux, il faut implanter une nouvelle JVM pour obtenir les deux fonctionnalités.
Plutôt que de réécrire ou de modifier une machine virtuelle statiquement, nous proposons une machine virtuelle Java standard (dans le sens où elle respecte la spécification de Sun [23, 16]) mais ouverte et spécialisable à l'exécution. Cette machine virtuelle, appelée Jnjvm (Jnjvm is Not a Java Virtual Machine[1]) prend en entrée un programme Java et un méta-programme permettant de modifier le fonctionnement de la machine virtuelle. L'ensemble de ces deux parties est ce que nous appellerons une application active. Les applications actives permettent à la fois de garder une compatibilité ascendante avec la machine virtuelle de Sun et de spécialiser à la volée la Jnjvm en fonction des besoins applicatifs.
Ces travaux s'inscrivent dans le cadre du projet "Machine Virtuelle Virtuelle" (MVV) [7, 8, 9, 12] visant à construire un environnement spécialisable à l'exécution. Cet environnement spécialisable est une machine virtuelle minimale et adaptable disposant d'un système de nommage, d'un compilateur en ligne, d'un

[1] Jnjvm est un ensemble de scripts chargés, spécialisés et executés dans la MVV, cet ensemble de scripts ne constitue une JVM dédiée qu'à l'execution et n'est pas en soit une JVM

éditeur de lien, de fortes propriétés d'introspection et d'un langage d'entrée proche de Scheme. La MVV constitue l'entrée pour la partie active (ou script actif) de l'application active. Grâce à cette architecture, le script actif s'exécute dans le même environement que la Jnjvm et peut modifier en profondeur les mécansimes de la Jnjvm.
La suite de ce document est le suivant : dans la section 2 nous détaillons d'autres approches visant à adapter les machines virtuelles Java, dans la section 3 nous présentons l'architecture générale de Jnjvm et dans la section 4 nous décrivons la MVV. La section 5 présente le prototype de Jnjvm que nous avons réalisé, puis la section 6 nous donne quelques exemples d'applications actives et quelques mesures de performances. Enfin, la section 7 présente les perspectives de ce projet.

2. État de l'art

Dans cet état de l'art nous présentons d'une part d'autres approches visant à rendre Java plus flexible, d'autre part, nous détaillons les modifications apportées aux JVM par les plateformes dédiées pour en abstraire les besoins en flexibilité.
Les solutions à base de pré-processeur comme OpenJava [21] étendent le langage Java avec de nouveaux mots clés. Un programme OpenJava est d'abord pré-compilé par le pré-processeur et ensuite compilé par un compilateur Java. Cette méthode permet d'injecter directement dans le binaire Java les appels à des méta-objets. Cette technique a le défaut d'être statique (car effectuée à la compilation), mais permet d'optimiser le code grâce au compilateur. Une amélioration des techniques à base de pré-processeur vient avec la manipulation de bytecode à la volée comme dans BCEL [18]. Cette technique permet d'adapter l'application lors de son chargement. Ces deux approches permettent une grande flexibilité et n'ont pas besoin de plateforme dédiée pour s'exécuter. Toutefois, la modification de bytecode à la volée est coûteuse en temps de calcul (une couche est ajoutée lors du chargement d'une classe) et est difficilement optimisable. D'autre part, ces deux approches ne permettent ni de modifier les interactions entre le système sous-jacent et la machine virtuelle, ni de modifier les mécanismes internes de la machine virtuelle.
Les plateformes dédiées à un matériel particulier comme KVM [1] et JavaCard [2] visent principalement à alléger la Machine Virtuelle Java. Quand une plateforme manque de ressource, certains mécanismes de la spécification de Sun ne sont pas implantés (les tâches dans JavaCard par exemple ou les ClassLoader dans les KVM). Ces plateformes montrent qu'il faut d'une part adopter une architecture modulaire dans la JVM et d'autre part qu'il ne faut charger ces modules que si on en a besoin. Le bytecode Java peut aussi être différent d'une machine à l'autre : il faut donc pouvoir spécialiser le compilateur de bytecode.
Les machines virtuelles dédiées à la réflexion comme MetaXa [20] ou Guaraná [6] permettent d'étendre l'API de réflexion Java (*java.lang.reflect*) grâce à un système d'interception et de méta-objets. Pour implanter ces systèmes, il faut pouvoir intercepter tous les appels de méthodes et accès aux variables pour déléguer le travail aux méta-objets. Le mécanisme d'interception est facile à mettre en place dans une machine virtuelle dédiée : il suffit de modifier les mécanismes de liaison interne de la machine virtuelle. Ces deux projets modifient la machine virtuelle plutôt que le bytecode pour des raisons de performances. Ils montrent qu'il faut pouvoir agir au niveau de l'édition de liens de la JVM pour pouvoir implanter des mécanismes de réflexion.
Les machines virtuelles Java dédiées à la persistance (PJama [22], Gemstone [3]...) modifient la notion d'accès aux champs d'un objet (écriture sur un support persistant de l'état de l'objet). Les objets persistants sont soit des objets persistants racines, soit contenus dans des objets persistants. Pour implanter un tel système, il suffit de modifier légèrement le ramasse-miettes pour suivre les objets persistants dans la mémoire et les mettre à jour : la fonction qui trace un objet doit pouvoir savoir si elle s'occupe d'un objet persistant ou non. Ce type de solution montre qu'il faut pouvoir spécialiser le ramasse-miettes et en particulier les fonctions qui tracent les objets.
Des systèmes sont construits à partir de Java comme KaffeOS [13]. KaffeOS permet de définir la notion de processus dans la machine virtuelle, d'assurer un partage des ressources (CPU, mémoire) et une isolation entre application (la faute d'un processus n'a aucune incidence sur les autres processus). KaffeOS modifie la colle entre le système d'exploitation et l'application. Nous montrerons dans la suite que le code actif permet aussi de modifier les mécanismes systèmes car il s'exécute au même niveau que la machine virtuelle.
Le Real-Time for Java™ Expert Group [5] propose une spécification pour construire une plateforme Java temps réel en introduisant la possibilité pour une application de spécifier son ordonnanceur. Pour

construire une JVM temps réel, il faut avoir une machine virtuelle standard qui offre la possibilité de remontrer au niveau applicatif un certain nombre de ces mécanismes internes (le temps d'une collection par exemple). Ce type de machine virtuelle doit présenter une API particulière aux applications et offrir des mécanismes pour modifier et/ou consulter des fonctionnalités systèmes. Les applications actives sont tout à fait adaptées à ce type de problème. Le code actif s'exécute dans le même environnement que la JVM et peut redescendre au niveau du système ce qui permet, par exemple, d'implanter des ordonnanceurs particuliers (cf. section 6).

Ces différentes plateformes sont dédiées à un certain type d'application. Si l'application a la possibilité de modifier la machine virtuelle Java grâce à du méta-code, on peut utiliser une JVM unique qui répond à tous ces besoins, à condition que la JVM soit construite de manière modulaire et qu'elle offre suffisamment d'outils d'adaptation, d'introspection et de réflexivité.

Un autre type d'adaptation est souhaitable : la gestion d'attributs dans la machine virtuelle. Un attribut Java est un nom et une structure de données dans le fichier de description de classe. Un certain nombre d'attributs doivent être obligatoirement gérés, comme l'attribut CODE d'une méthode qui donne le bytecode de la méthode Java. Lorsqu'une machine virtuelle ne sait pas gérer un attribut, elle l'ignore silencieusement. L'analyse statique des fichiers Java permet de spécifier des comportements de l'application ou d'optimiser son chargement. Cet analyse entraîne la création d'un attribut dédié qui sera utilisé par une JVM dédiée.

Un exemple d'attribut non standard est un attribut de PCC (Proof Carrying Code) [14, 10] qui permet de donner une preuve que le typage du bytecode est correct. Cette preuve est vérifiable dans la machine virtuelle en une seule passe et permet d'économiser une grande quantité de mémoire lors de la vérification. Lorsqu'une telle technique est trouvée, se pose le problème de la tester et de la déployer. En effet, l'ajout de la gestion d'un attribut va impliquer la modification de la machine virtuelle (avec l'exemple précèdent, le vérifieur de code est modifié). Les applications actives permettent de résoudre ce problème. C'est l'application qui modifie sa plateforme d'exécution pour prendre en compte ce nouvel attribut.

3. Architecture de la Jnjvm

Compte tenu des limitations et des besoins des approches existantes décrites dans la section précedente, l'architecture de la Jnjvm respecte un certain nombre de critères :

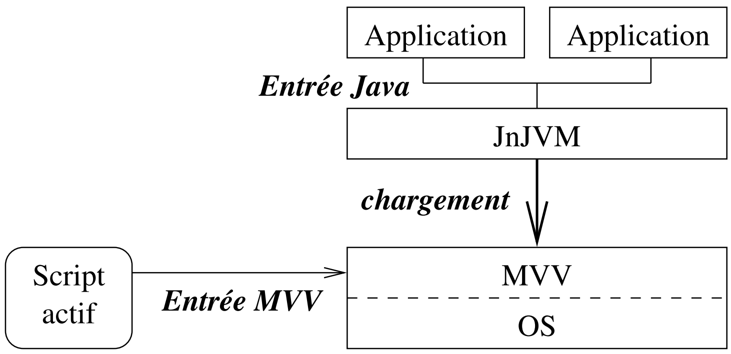

FIG. 1 – Architecture générale

- une structure modulaire pour permettre la construction de JVM légère,
- la possibilité d'intercepter des appels à des méthodes et des accès à des champs pour mettre en place un système méta-objets, de tissage d'aspects etc...
- un ramasse-miettes spécialisable pour effectuer une action sur certains objets de la mémoire (par exemple l'écriture sur disque),

– un haut niveau d'introspection et de flexibilité pour pouvoir modifier les mécanismes internes de la JVM (pour la prise en compte du PCC par exemple),
– la mise à disposition des mécanismes du système sous-jacent dans les applications actives.

Une application active est chargée via un script actif de configuration qui encapsule le chargement de la JVM et de l'application (Fig. 1). Ce script s'occupe de charger les modules de Jnjvm nécessaires au bon fonctionnement de l'application et éventuellement de les spécialiser.

La machine virtuelle Java est, au départ, constituée d'une seule tâche. Un des exemples donné dans la section 6 montre qu'avec deux tâches, on peut utiliser cette architecture pour reconfigurer à la volée la machine virtuelle.

Une série de modules est proposée pour faire tourner une application Java standard. Chaque module est écrit dans le langage de la MVV : la machine virtuelle Java est intégralement compilée au démarrage et est donc portable. Les modules sont dépendants les uns des autres. Sur la figure 2, on voit l'architecture de dépendance des modules. Certains modules ont été volontairement omis car ils ne présentent qu'un faible intérêt.

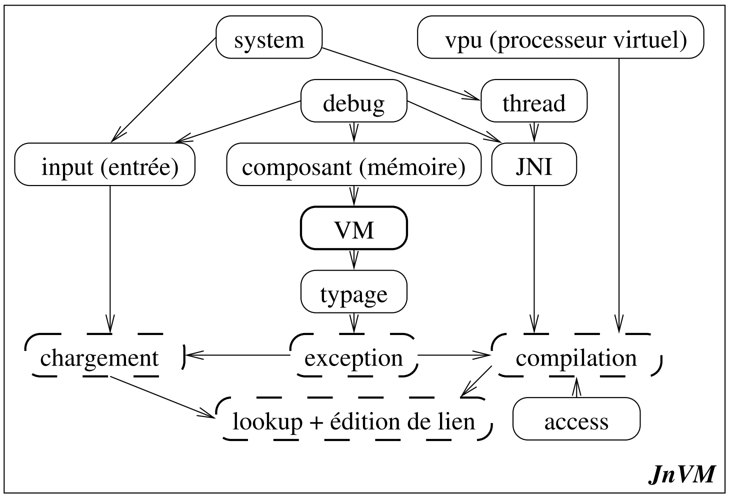

FIG. 2 – Dépendance de modules

Chaque module peut être remplacé au chargement ce qui permet à l'application de personnaliser globalement un module. Le remplacement de module à la volée n'est pas encore possible et fait partie des perspectives. Chaque module est conçu pour être reconfiguré à la volée en remplaçant certaines fonctions dans ces modules. La section 5 detaille l'implantation des principaux modules de la Jnjvm.

4. La machine virtuelle virtuelle

La machine virtuelle virtuelle est la plateforme d'exécution de notre machine virtuelle Java. Dans cette section, nous allons montrer certains aspects de la MVV qui permettent de mettre en place la Jnjvm.

4.1. Fonctionnement de la MVV

La machine virtuelle virtuelle (MVV) est un compilateur/éditeur de lien à la volée. Le langage d'entrée de la MVV est un langage proche de Scheme. Chaque phrase lue en entrée est envoyée au parseur qui s'occupe de transformer cette phrase en un arbre syntaxique abstrait (AST). Cet arbre est ensuite envoyé au compilateur de la MVV qui s'occupe de transformer cette arborescence en une séquence d'instructions pour un processeur virtuel : le VPU (Virtual Processor Unit). Le processeur virtuel est une machine à pile abstraite qui transforme une séquence d'instructions en une méthode assemblée. L'adresse de la méthode assemblée est ensuite envoyée à l'éditeur de lien qui s'occupe de lier le résultat à un symbole (Fig. 3). La MVV possède donc une propriété forte : *toute fonction est d'abord compilée puis exécutée.* Cette propriété va permettre à la Jnjvm d'implanter un compilateur en ligne (JIT).

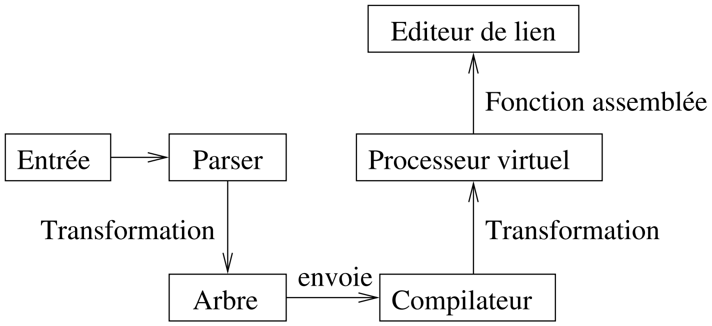

FIG. 3 – La MVV

Un AST est constitué d'objets MVV. Il existe principalement cinq types d'objet MVV : les entiers, les chaînes de caractères, les symboles, les listes et les modules MVV (différents des modules de la Jnjvm décrits précédemment). Un symbole est une référence vers un élément, chaque symbole est contenu dans un module. Un symbole contient trois champs : valeur, syntaxe et objet. Le champ objet est laissé libre à l'application et est utilisé pour stocker des objets MVV[2]. Tout symbole est stocké dans un module ce qui permet de regrouper des symboles ensemble, par exemple le symbole :object.list.at représente le symbole at du module list lui-même contenu dans le module object. Le : représente la racine des modules appelée module global et le . est le séparateur de modules.
Tout objet MVV peut être compilé (sauf les modules) :
- une chaîne de caractère est compilée vers l'adresse de la chaîne,
- un entier est compilé vers sa valeur,
- un symbole est compilé vers sa valeur,
- une liste est compilée comme une application fonctionnelle, on a deux possibilités :
  - si le premier élément de la liste est un symbole et si la syntaxe de ce symbole est non nul, alors, la fonction contenue dans le champ syntaxe est appelée (avec l'AST et le compilateur comme arguments) et c'est l'AST résultat de cette fonction qui est compilé à la place de l'AST original,
  - sinon, chaque objet de la liste est remplacé par l'objet compilé et l'appel de la fonction est réalisé suivant la sémantique C ;

La présence du champ syntaxe dans les symboles permet d'implanter un système de macro puissant. Les syntaxes définissent la manière dont une application fonctionnelle va être compilée en modifiant les AST, mais aussi en pilotant directement le compilateur (VPU). Les mots clés du langage MVV sont implantés en utilisant ce système de syntaxe. Une application peut étendre le langage MVV en ajoutant des syntaxes ou le modifier en changeant les syntaxes des mots clés.
Les syntaxes sont utiles dans la Jnjvm pour rendre transparents les appels à des symboles Java à partir de la MVV. En effet, la sémantique d'un appel Java est une sémantique objet différente de la sémantique naturelle de la MVV.
La MVV est développée sur plusieurs plateformes : linux, windows/cygwin, macosx, mais aussi en démarrage seul au dessus de Think [17] (un micro noyau developpé par France Telecom R&D). La MVV est aussi embarquée dans le noyau linux comme module et dans un intergiciel : OpenCCM.
Comme la MVV donne une abstraction du système sous-jacent et du processeur, Jnjvm peut être exécutée sur toutes ces plateformes, en particulier au dessus de Think, ce qui permet d'avoir un système entièrement Java.

[2] Par défaut, le compilateur stocke l'arbre syntaxique abstrait qui a donné lieu à la création du symbole

4.2. Utilisation de la MVV dans la Jnjvm

Tous les symboles Java sont projetés vers des symboles MVV. Un package ou une classe Java est projeté vers un module MVV, une méthode f d'une classe pack.A Java devient le symbole :pack.A.f_sign où *sign* représente la signature Java, un champs c d'une classe pack.A devient le symbole :pack.A.c_sign.
Grâce à cette projection, les scripts actifs de l'application active appellent directement des méthodes Java ou consultent des champs de classe ou d'instance.
L'appel d'une méthode virtuelle Java se fait de manière différente de celle de C : c'est l'objet cible qui décide quelle méthode est appelée. Par exemple, un appel à la méthode java.lang.Object.toString(), sur un objet de type A qui surdéfinit cette méthode, est effectué sur la méthode de la classe A et non sur celle de java.lang.Object. La sémantique du compilateur de la MVV n'est donc pas appropriée. Tous les symboles projetés ont leur champ syntaxe non vide ce qui permet d'effectuer un appel virtuel, spécial[3] ou statique[4] de manière transparente. De la même façon, l'algorithme utilisé pour accéder aux champs est masqué derrière une syntaxe.
L'autre aspect de la MVV largement exploité par la Jnjvm est la machine virtuelle à pile interne (VPU) pour construire un compilateur de bytecodes en ligne.

5. Implantation de la Jnjvm

Nous décivons plus précisément certains modules de la machine virtuelle Java dans cette section et nous montrons comment ceux-ci peuvent être adaptés par l'application active. Actuellement, il n'existe qu'une implantation de chaque module pour permettre la construction de la machine virtuelle de référence de Sun.

5.1. Le module Système

Le module système fait l'interaction entre le système sous-jacent (OS et MVV) et la Jnjvm. Le fait d'isoler dans un seul module les fonctions systèmes permet de modifier l'interface entre la Jnjvm et le système. On trouve principalement les fonctions d'entrée/sortie dans ce module pour pouvoir lire un fichier .class. Les fonctions systèmes liées à la gestion de processus sont dans un autre module.

5.2. Le module Objet

Le module objet s'occupe de la gestion mémoire de bas niveau. Le ramasse-miettes implanté est un mark-and-trace exact, coloré et incrémental inspiré des algorithmes de Boehm [15] et de Dijkstra [11]. C'est le même allocateur qui peut servir dans la MVV[5] et qui est utilisé dans ce module. Le ramasse-miettes est écrit en C++ et est interfacé avec la Jnjvm via le module Objet. Le ramasse-miettes a d'abord été développé dans le cadre de la machine virtuelle Java c'est pourquoi nous le présentons comme une partie de la Jnjvm et non de la MVV.
Tous les objets (MVV et Java) sont des composants présentant une interface commune, un vecteur de fonction à l'offset 0 de l'objet. Pour plus de clarté, les signatures sont données en C :
– void trace(Object *, size_t, void *unused) : trace l'objet,
– void finalize(Object, size_t sz, void *unused) : finalise l'objet,
– void compile(Object *, Compiler *) : compile l'objet dans le compilateur,
– void print(Object *, PrintBuffer *) : écrit le contenu de l'objet dans le tampon ;
Le vecteur d'interface des objets contient aussi 48 octets inutilisés prévus pour étendre le comportement de tous les objets de la MVV. Par exemple, on peut de cette façon implanter dynamiquement un système de sérialisation en augmentant l'interface à la volée.
La fonction print des objets Java appelle la méthode toString Java et remplit le PrintBuffer avec le contenu de la chaîne résultat. La fonction finalize des objets Java appelle la méthode finalise Java. Les fonctions trace Java s'occupent de tracer de manière sélective les objets Java. Chaque fonction trace est compilée dynamiquement pour chaque classe de manière à augmenter le performances. Par exemple, une classe ne contenant que douze entiers et une référence possède une fonction trace qui n'essaye de tracer que

[3] cette notion est celle des constructeurs en Java ou celle des méthodes d'instance non virtuelles en C++
[4] appel d'une méthode de classe
[5] La MVV possède deux ramasse-miettes, un ramasse-miettes léger adapté à la version sur Think et un ramasse-miettes lourd que nous présentons ici

la référence. Cette technique évite de deréférencer les douze entiers inutilement. De plus cette méthode évite de confondre un entier avec une référence et permet de rendre le ramasse-miettes exact.
En plus de son interface MVV, chaque objet Java possède une classe (à l'offset 4 de l'objet). Une classe est un objet non typé stockant les méta-données de l'objet (cf. 5.5).
De son côté le ramasse-miettes présente aussi une interface :
- void markAndTrace(Object *, void *reserved) : marque un objet comme atteignable et le trace,
- void (*markFromRoot)(void *unused) : trace les objets à partir des racines.

Lorsqu'une collection commence, la mémoire objet est bloquée en écriture (pour éviter les problèmes liés aux mutateurs) et la fonction markFromRoot est appelée. La fonction markFromRoot utilisée par défaut trace le symbole global (qui contient le module MVV global dans son champ objet).
L'algorithme du ramasse-miettes ne peut pas être changé (certains paramètres comme le nombre d'objets alloués entre deux collections peuvent être spécifiés en cours d'exécution). En revanche, la fonction markAndTrace et les fonctions trace des objets peuvent être modifiées par une application active.
Au démarrage, le module Objet peut aussi être remplacé globalement (à condition qu'il présente la même interface) par une application active pour utiliser un autre algorithme. Le remplacement à la volée du module Objet n'est actuellement pas envisagé.

5.3. Le module d'Entrée

Ce module définit une entrée Java. Une entrée est définie sous la forme d'un composant. Ce composant possède une interface (un vecteur de fonction) qui définit les opérations de base que l'on peut effectuer sur une entrée. Ces opérations sont au nombre de quatre : close, seek, tell et read-u1.
Instancier une entrée revient à créer une table virtuelle implantant ces fonctions. De cette manière, on peut lire un fichier .class à partir d'un descripteur unix, de la mémoire ou laisser libre l'implantation d'une entrée à une application active.
Il n'existe pas de fonction d'ouverture d'entrée dans cette table. En effet, l'ouverture d'une entrée est effectuée dans le module de chargement. Pour créer un nouveau type d'entrée, il faut aussi modifier la fonction d'ouverture du module MV.

5.4. Le module MV

C'est principalement grâce à ce module qu'une application active peut modifier le comportement de Jnjvm. Chaque méthode, champ, ou classe est associé à une machine virtuelle. Le module MV ne contient qu'une structure qui décrit le cycle de vie des classes Java, et un certain nombre de fonctions nécessaires pour l'éditeur de lien et le module d'exception. Les signatures sont données en C :
- char *name : le nom de la machine virtuelle permettant d'identifier la machine virtuelle ;
- Input *open(char *name) : ouvre un fichier de description de classe. Par défaut cette fonction essaye de trouver le .class dans le système de fichier ;
- int (*load)(MV *, Class *) : la fonction de chargement d'une classe. C'est grâce à cette fonction qu'on peut implanter un chargement partiel de .class par le réseau ;
- void (*resolve)(MV *, Class *) : résout la classe. Cette fonction est appelée après le chargement des classes parentes d'une classe et ne fait rien par défaut ;
- void (*initialise-class)(MV *, Class *) : cette fonction a pour rôle d'initialiser la classe. Par défaut cette fonction appelle la méthode <clinit> de la classe pour initialiser la partie statique de la classe ;
- void (*print-error)(Object *excp) : affiche une exception qui n'a pas été attrapée. Par défaut cette fonction affiche le nom de l'exception et l'état de la pile au moment de l'exception ;
- int (*check-error)(MV *src, Object *excp) : cette fonction permet de vérifier si une exception excp levée par la machine virtuelle src doit être attrapée en un point particulier. Par défaut, cette fonction réalise l'algorithme d'attrapement Java ;
- void *(*compile)(Class *, Method *, int access) : cette fonction compile la méthode passée en argument et renvoie un pointeur vers celle-ci. On peut modifier totalement le compilateur grâce à cette fonction ;
- char reserved[32] : des octets réservés pour une application active particulière.

Une application active peut réifier ces fonctions (ou valeurs) en cours d'exécution ou au démarrage. La structure de machine virtuelle est le pivot de la plupart des autres modules : il est peu probable qu'une application puisse modifier cette structure sans réimplanter intégralement la machine virtuelle. Plusieurs instances de cette structure peuvent cohabiter simultanément : chaque classe, champ ou méthode est lié

à une structure de machine virtuelle particulière.
Cette aproche étend la notion classique de ClassLoader Java : une structure de VM spécifie non seulement comment des classes sont chargées, mais aussi comment le bytecode des fonctions est compilé et les exceptions gérées.
Cette structure assez simple peut être utilisée avec les modules d' Édition de liens et d'Exception pour implanter d'autres machines virtuelles et pour les faire cohabiter entre elles.

5.5. Le module de Chargement

Ce module contient cinq fonctions : la fonction générique de chargement et les quatres fonctions qui implantent ce chargement. La fonction générique appelle les fonctions open et load de la machine virtuelle passée en argument puis charge les classes parentes (avec la même machine virtuelle), ensuite elle appelle les fonctions resolve et initialise-class.
La lecture d'un fichier .class (fonction load) revient à remplir les champs qui décrivent une classe et à créer les symboles de la classe. La structure qui décrit un fichier .class est un objet placé dans le champ objet du symbole new du module associé à la classe chargée. On trouve dans cette structure les informations nécessaires à l'édition de liens, le sémaphore de la classe, les droits d'accès de la classe etc...
La description d'une classe se trouve dans un symbole appelé class-def. Cette structure ne peut être que difficilement adaptées. Par contre la manière de remplir cette structure peut tout à fait être modifiée par une application active. On trouve aussi dans ce module les structures qui permettent de décrire les méthodes et les champs.
L'ajout du traitement de nouveaux attributs comme le Proof Carrying Code peut se faire de manière aisée. Le chargeur (load), lorsqu'il trouve un attribut, vérifie si un symbole MVV portant le nom de cet attribut existe. Si c'est le cas, la fonction contenue dans le champ valeur du symbole est appelée. C'est avec cette technique que l'attribut CODE est lue. Ajouter la gestion d'un attribut revient simplement à créer un symbole.

5.6. Le module d'Édition de liens

Ce module s'occupe d'implanter les algorithmes de liaison. Huit types de liaison sont implantés : les liaisons virtuelles, statiques et spéciales pour les méthodes, les liaisons statiques et virtuelles, en lecture et en écriture pour les champs, et la liaison de la pseudo-méthode new qui alloue la mémoire d'un objet. Chacun de ces algorithmes utilise à peu près les mêmes concepts c'est pourquoi nous ne nous focaliserons que sur un seul de ces algorithmes (la liaison virtuelle de méthodes) dans la suite de cette sous-section.
La liaison virtuelle repose sur la syntaxe du symbole synt-virtual-call et la méthode vpu-virtual-call.
Vpu-virtual-call s'occupe de piloter le processeur virtuel pour effectuer l'appel virtuel selon un algorithme particulier. La syntaxe effectue le même travail mais au niveau langage. Elle remplace l'AST d'appel à la méthode (par exemple (A.meth_sign obj)) par un autre AST qui implante l'algorithme d'appel (par exemple (invoke-virtual X obj) où X est l'objet de description de la méthode et invoke-virtual une fonction de liaison).
Par défaut, l'algorithme utilisé est un algorithme de liaison au plus tard via un cache en ligne. Le cache est une structure minimale permettant de pouvoir éditer le lien au plus tard :
- Method *meth : la description de la méthode à lier,
- Class *last : la dernière cible d'appel,
- void *ptr : le pointeur vers la méthode compilée.

Lors de l'appel de la syntaxe synt-virtual-call, un cache en ligne est alloué. Il est rempli avec l'objet associé à la méthode[6], l'objet nul et le pointeur vers la fonction de liaison virtuelle. L'appel (A.meth_sign obj) est remplacé par ((:caches.ptr cache) cache obj), c'est-à-dire l'appel de la fonction ptr du cache. La fonction de liaison virtuelle est donc appelée lors de la première exécution. Elle s'occupe de charger la classe le cas échéant et de compiler la méthode si ce n'est pas déjà fait. Le cache est ensuite mis à jour avec l'objet cible et la méthode compilée. À l'appel suivant, le code de la méthode est directement appelé. Si la cible de l'appel change, le cache est invalidé et la liaison recommence.
Une application active peut modifier l'algorithme de liaison en remplaçant la syntaxe syn-virtual-call, la méthode vpu-virtual-call et la méthode compile de la machine virtuelle (pour ne pas prendre en compte

[6] qui se trouve dans le champ objet du symbole associé à la méthode

l'invalidation du cache).

5.7. Le module de Compilation

Le module de compilation ne possède qu'un point d'entrée : la fonction compile placée dans la structure de machine virtuelle. Cette fonction s'occupe de compiler le bytecode Java de la méthode. La fonction compile s'occupe d'abord de générer le code nécessaire à la gestion du cache en ligne pour les méthodes virtuelles et appelle ensuite une fonction qui s'occupe de générer réellement la méthode.
La compilation des bytecodes se fait via un vecteur de 256 fonctions (un par bytecode). Modifier le fonctionnement d'un bytecode revient à modifier une des entrées de ce vecteur. Si on veut supprimer la gestion des sémaphores par exemple, il suffit de supprimer les bytecodes monitorenter et monitorexit.
Le compilateur de bytecode utilise le processeur virtuel plutôt que la construction d'un AST pour des raisons de performance. La création d'un arbre syntaxique est coûteuse en mémoire (donc ralentit le ramasse-miettes) et en temps (une étape supplémentaire dans la génération de code). On perd à cette étape la possibilité de reconstruire l'AST de la méthode en suivant, par exemple, l'algorithme de Krakatoa [24]. Si une application active Java souhaite travailler directement sur l'AST, rien ne l'empêche de réimplanter le compilateur de bytecodes.

5.8. Le module de Tâches

Le module de Tâches implante les primitives nécessaires à la gestion de processus légers dans la Jnjvm. Ce module respecte une interface proche de l'API des pthread : la création de processus, la synchronisation entre processus, les plages de mémoire locale, etc...
La plupart des fonctions de ce module reposent directement sur les fonctions des pthread : ce module est donc non portable sur différentes architectures (en particulier au dessus de la MVV/Think [17]).

6. Évaluation

Dans cette section, nous décrivons nos premières applications actives ainsi que nos premières mesures. La réalisation du prototype est récente c'est pourquoi nous avons peu de mesures de performance et pas encore d'exemple complet d'applications actives.

6.1. La machine virtuelle de référence

Notre première application active charge les modules prédéfinis et lance une application Java passée en paramètre. C'est une application active servant de test de référence pour vérifier le bon fonctionnement global de la machine virtuelle. Pour tester cette application, nous utilisons les classes de base du projet GNU ClassPath [4].
Le script de démarrage de l'application active est assez sommaire. Il suffit de charger les modules par défaut, de positionner le chemin de recherche de classes vers un répertoire contenant les classes de GNU et d'appeler la fonction main de l'application.
Cette application active correspond exactement à une JVM comme celle de Sun ou la machine virtuelle kaffe[7].

6.2. Mise à jour à distance de la Jnjvm

Cette application active permet de modifier à la volée et à distance la machine virtuelle Java. Elle est basée sur l'application active précédente pour faire tourner une machine virtuelle standard. L'application active commence par créer un processus de contrôle qui s'occupe de modifier la mémoire de la machine virtuelle pendant qu'un (ou plusieurs) processus exécute(nt) le code de l'application Java (Fig. 4).
La communication entre la Jnjvm et le serveur de mise à jour se fait en sérialisant l'AST d'une expression à évaluer. Le serveur lit un fichier (ou l'entrée standard), construit les AST des expressions à évaluer, les sérialise et les envoie à la Jnjvm via une socket UDP.
Le processus de contrôle de la Jnjvm ouvre une socket UDP et attend de recevoir un AST sérialisé. Lorsque cette arbre arrive, il est compilé puis exécuté. Nous présenterons dans la sous-section suivant un exemple d'utilisation de cette application active. Le script actif fait 238 lignes (ouverture de la socket,

[7] Actuellement cette JVM implante la spécification CLDC [1] car certains aspects de la spécification J2SE [23] ne sont pas encore developpés.

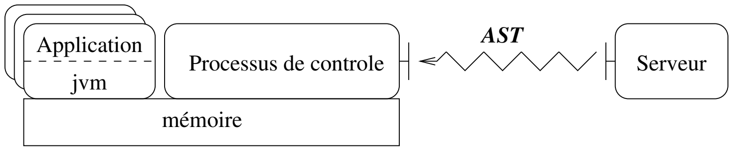

FIG. 4 – Architecture de la mise à jour

desérialisation, lancement du processus de contrôle).

6.3. Modification de la structure de classe

Cette application active définit des techniques pour modifier la description d'une classe. Deux versions de cette application active existent : une première version de cette application modifie la description au chargement, un seconde effectue le même travail en cours d'exécution.

6.3.1. Modification au chargement

Le moment adéquat pour modifier la définition d'une classe dans son cycle de vie se situe après le chargement de la classe (méthode load de la machine virtuelle) et avant l'initialisation de la classe (méthode initialise-class), c'est-à-dire que l'application active ajoute une méthode resolve dans le cycle de chargement appelée global-reifie.

Global-reifie commence par regarder si un symbole reifie existe dans le module MVV associé à la classe. Si ce symbole existe, global-reifie évalue l'application fonctionnelle (reifie Class), où Class est la classe en cours de chargement. De cette manière, on peut spécifier quelle classe sera modifiée au chargement.

Dans cet exemple, on commence par dupliquer la méthode finalize de la classe Java.lang.Object. On remplace cette méthode une fonction qui affiche quel objet est détruit et qui appelle l'ancienne méthode. Grâce à cette petite manipulation (le document actif fait 42 lignes), toutes les destructions d'objets Java sont tracées. Aucune synchronisation n'est nécessaire car l'accès à la classe est bloqué par un sémaphore lors du chargement.

6.3.2. Modification à la volée

Cette application active effectue la même action que la précédente (l'affichage de tous les objets détruits). Le code de cette application active est plus complexe. En effet, il faut non seulement mettre à jour les méta-données de la classe java.lang.Object, mais il faut aussi mettre à jour tous les caches en ligne. La mise à jour des caches en ligne se fait simplement en réinitialisant tous les caches associés aux méthodes finalize(). De plus, plusieurs processus peuvent travailler simultanément sur la classe ou sur les caches : il faut acquérir le sémaphore associé à la classe et celui associé à la création d'un cache en ligne.

Cette application active utilise aussi le code de l'application active de mise à jour à distance pour injecter la modification. Le script envoyé par le serveur sous forme d'arbre sérialisé ne fait que 10 lignes.

Ces deux exemples montrent que les outils nécessaires à la mise en place d'un tisseur d'aspects sont présents et offrent un interface claire.

6.4. Modification de l'ordonnancement

Cette application active ne peut s'exécuter qu'en mode administrateur sous linux. L'application active démarre avec un ordonnanceur FIFO (tant que l'application ne fait pas un sleep, elle reste sur le processeur). La fonction de création de tâches est redéfinie pour créer des tâches ayant un ordonnanceur round-robin. Les fonctions du noyau linux (sched_setparam) sont utilisées pour modifier l'ordonnancement. Cet exemple illustre la possibilité de modifier l'interface entre la JVM et le Système.

6.5. Mesures de performances

Aucune mesure poussée n'a encore été effectuée, c'est pourquoi nous ne donnons que des ordres de grandeur. Les valeurs données sont celles obtenues avec l'application active de machine virtuelle standard. Le démarrage de la Jnjvm est assez lent : environ 3 secondes sur un G3 à 366 MHz sous linux contre 1 seconde pour la machine virtuelle de SUN sur la même plateforme. Cet écart s'explique facilement : la

Jnjvm est intégralement compilée au démarrage alors que la JVM de Sun est déjà compilée.
En revanche, à l'exécution, les performances sont assez proches de la machine virtuelle de Sun. La Jnjvm une fois compilée a des performances proches du C et le coût des appels indirects des fonctions de la structure de machine virtuelle semble assez négligeable.
Les performances de l'allocateur du ramasse-miettes sont légèrement meilleures que celle de la fonction malloc de la glibc. Une collection prend 3ms pour 130ko dans la mémoire objet. Les fonctions trace compilées à la volée multiplient les performances d'une collection par 2.5. Aucune comparaison avec le ramasse-miettes de Boehm [15] n'a encore été effectuée.
Dans l'ensemble, les performances de la Jnjvm semblent raisonnables, mais seules des mesures systématiques nous permettront de le prouver.

7. Perspectives et Conclusions

Nous avons présenté dans cet article une machine virtuelle Java adaptable pour applications actives. La Jnjvm respecte les caractéristiques énumérées dans la section 3. Une application active peut modifier en profondeur le comportement de la Jnjvm en fonction de ses besoins, que ce soit au démarrage ou pendant l'exécution.
Le fait de découpler l'application entre une partie fonctionnelle (l'application) et non fonctionnelle (le script actif) n'est pas récente mais la technique utilisée permet d'exécuter la partie non fonctionnelle au même niveau que la machine virtuelle Java. Agir directement à ce niveau permet une adaptation plus poussée de l'application et de la plateforme sans perte de performances notables.
Cette plateforme va maintenant être utilisée pour construire des applications actives plus importantes, en particulier, une application active dédiée à la réplication est en cours de développement et une application active dédiée à la persistance est en projet.
Un document actif dédiée à la mesure est en cours de réalisation. Ce document actif va permettre de mesurer les performances de la Jnjvm et de montrer les points forts et les points faibles de celle-ci.
Une autre application active est en cours de développement : la gestion d'un attribut donnant des renseignements sur la durée de vie des objets alloués. La gestion de cet attribut va permettre d'accélérer le ramasse-miettes.

Bibliographie

1. http ://java.sun.com/products/cldc/.
2. http ://java.sun.com/products/javacard/.
3. www.gemstone.com.
4. The GNU Classpath Project. www.gnu.org/software/classpath/classpath.html.
5. The Real-Time Specification for Java. www.rtj.org/rtsj-V1.0.pdf.
6. A. Olivia and L.E. Buzato. Design and implementation of Guarana. In *Conference on Object-Oriented Technologies and System (COOTS)*, May 1999.
7. B. Folliot, I. Piumarta, and F. Riccardi. Virtual Virtual Machines. *Cabernet Radical Workshop*, 1997.
8. B. Folliot, I. Piumarta, and F. Riccardi. A Dynamically Configurable, Multi-Language Execution Platform. In *8th ACM SIGOPS European Workshop*, 1998.
9. B. Folliot, I. Piumarta, L. Seinturier, C. Baillarguet, C. Khoury, A. Léger, and F. Ogel. Beyond flexibility and reflection : the virtual virtual machine approach. *NATO Advanced Research Workshop, Environments, Tools and Applications for Cluster Computing*, pages 17–26, 2002. LNCS 2326, Springer-Verlag.
10. E. Rose and K.H. Rose. Lightweight Bytecode Verification. *Formal Underpinnings of Java (an OOPSLA workshop), ACM*, 1998. Vancouver, BC, Canada.
11. E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection : an exercise in cooperation. *CACM*, 21(11), November 1978.
12. F. Ogel, G. Thomas, B. Folliot, and I. Piumarta. Application-Level concurrency Management. *NATO Advanced Research Workshop, Concurrent Information Processing and Computing*, July 2003. Sinaia, Romania.
13. G. Back, W.C. Hsieh, and J. Lepreau. Processes in KaffeOS : Isolation, Resource Management, and

Sharing in Java. In *4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000. USENIX.

14. G.C. Necula. Proof-Carrying Code. *Conference Record of POPL'97 : The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, January 1997. Paris, France.
15. H-J. Boehm, A.J. Demers, and S. Shenker. Mostly parallel garbage collection. *Conference on Programming Language Design and Implementation (ACM SIGPLAN'91)*, 1991.
16. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification (2nd Edition)*. Paperback, 1996.
17. JP. Fassino and JB. Stephani. THINK : un noyau d'infrastructure répartie adaptable. *Proceedings of CFSE 2*, April 2001. Paris, France.
18. M. Dahm. Byte Code Engineering with the BCEL API. Technical Report B-17-98, Institut für Informatik, Freie Universit at Berlin, April 2001.
19. M. Golm. *Design and implementation of a meta architecture for Java*. PhD thesis, University of Erlang, January 1997.
20. M. Golm and J. Kleinoder. metaXa and the Future of Reflection. *Workshop on Reflective Programming in C++ and Java*, 1998.
21. M. Tatsubori. An Extension Mechanism for the Java Language. Master's thesis, Graduate School of Engineering, University of Tsukuba, Ibaraki, Japan, February 1999.
22. M.P. Atkinson, L. Daynes, M.J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *ACM Sigmod Record*, 25(4), December 1996.
23. T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Paperback.
24. T.A. Proebsting and S.A. Watterson. Krakatoa : Decompilation in Java (Does Bytecode Reveal Source ?). *Third USENIX Conf. Object-Oriented Technologies and Systems (COOTS)*, pages 185–197, June 1997.