UNIVERSITÉ DU QUÉBEC À MONTRÉAL

INTERFACE DE DÉBOGAGE DE LA MACHINE VIRTUELLE JAVA

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR
NIZAR AHMOUDA

MARS 2006

Remerciements

En premier lieu, je souhaiterais remercier chaleureusement mon Directeur de Recherche, Étienne M. Gagnon, pour la confiance qu'il m'a accordée. Son encadrement, sa patience et sa bienveillance tout au long de cette réalisation ont été inestimables. Merci également à Grezgorz B. Prokopski pour son œil attentif et son avis éclairé durant la phase de développement.

Mon dévouement à ce projet n'aurait été tel sans soutien financier. Je voudrais réitérer mes remerciements les plus vifs à Étienne M. Gagnon pour sa politique de financement des étudiants en recherche. Je tiens à exprimer ma profonde gratitude envers la Fondation UQAM pour son appui via le Fonds à l'Accessibilité et à la Réussite des Etudes. Toute ma reconnaissance à la Faculté des Sciences de l'UQAM pour m'avoir soutenu grâce à son programme de Bourses d'Excellence.

Remerciements également aux membres du Laboratoire de Recherche sur les Technologies du Commerce Electronique (LATECE) de l'UQAM qui m'ont offert les moyens matériels me permettant de mener à bien mes recherches. Une pensée aux étudiants et professeurs du laboratoire Sable de l'Université McGill pour m'avoir accueilli lors des leurs séminaires, et avoir mis à ma disposition un environnement de test.

Enfin, mille pardons à ma famille et mes proches pour ma disponibilité sporadique ces derniers mois.

A tous, merci.

Table des matières

Liste des figures

Liste des sigles et acronymes

J2SE Java 2 Platform Standard Edition

JDI Java Debug Interface.

JDWP Java Debug Wire Protocol.

JNI Java Native Interface.

JPDA Java Platform Debugger Architecture.

JRE Java Runtime Environment.

JVM Java Virtual Machine.

JVMDI Java Virtual Machine Debugger Interface.

JVMPI Java Virtual Machine Profiler Interface.

JVMTI Java Virtual Machine Tool Interface.

VM Virtual Machine.

Résumé

Le débogage tient une place grandissante dans le cycle de développement d'un logiciel. Les recherches dans ce domaine tentent de créer des outils permettant un accès plus rapide aux fautes, quel que soit le langage de programmation utilisé. Étant donné l'indépendance du code Java vis-à-vis de la plateforme sur laquelle il est exécuté, la machine virtuelle Java doit fournir un ensemble de mécanismes permettant aux outils de débogage d'accéder aux informations relatives à l'exécution de l'application déboguée. Bien que la grande majorité des machines virtuelles commerciales soient dotées de mécanismes de support au débogage, aucune libre, en revanche, n'offrait une telle fonctionnalité à l'achèvement de nos travaux.

La principale motivation derrière ce mémoire a été la mise en lumière des différentes étapes jalonnant la mise en place d'une architecture de débogage Java totalement libre. Nous décrivons ici le choix de l'architecture et les critères nous ayant conduits à ce choix. Nous détaillons également les entités intervenant dans cette architecture, leur nature et leur rôle. Nous proposons enfin une critique constructive des normes régissant ce domaine, suggérant quelques améliorations possibles.

Dans le cadre de nos travaux, nous avons réalisé l'implantation de l'interface de débogage Java (Java Virtual Machine Debug Interface, JVMDI) au sein de SableVM, machine virtuelle Java libre et conforme aux normes. D'autre part, nous avons développé un module indépendant permettant d'établir la connexion entre machine virtuelle Java et débogueur. Ce module gère également les objets manipulés durant une session de débogage, ainsi que les événements générés par la machine virtuelle.

Finalement, nous avons connecté les éléments conçus ou modifiés dans le cadre de notre étude à d'autres éléments existants au préalable (Eclipse, un débogueur Java disponible librement). Les résultats obtenus lors des tests nous ont conforté dans les différents choix effectués lors du développement. L'utilisation de débogueurs totalement indépendants de la machine virtuelle utilisée, tel Eclipse, et la bonne tenue des sessions de débogage effectuées ont permis la validation de la conformité de nos travaux aux normes en vigueur.

Mots clés : machine virtuelle, Java, SableVM, débogage, interface de débogage, architecture de débogage, JDWP, JVMDI, JPDA, JVMTI, JRE

Abstract

Debugging is a major part of the software development cycle. Research tries to quicken fault access, independently of the programming langage. As Java programs are isolated from the underlying platform, the Java virtual machine must provide a set of tools to allow debuggers to access information about the execution state of the debugged application. Although most of the commercial Java virtual machines offer debugging support, none of the free ones does.

The main motivation behind this thesis is to bring the process of building an open source Java debugging architecture out. We describe the choice of an architecture and present our selection criteria. We also detail each entity inside this architecture, as well as its nature and role. Then we put forward an appreciation of the specification and suggest some possible improvement.

To achieve our objective, we introduce the first implementation of the Java Virtual Machine Debug Interface in a free and specification-compliant virtual machine : SableVM. We also present a virtual-machine-independent library in charge of connecting a virtual machine and a debugger, as well as managing objects and events.

Finally, we connect the created or modified elements to the already existing ones in order to complete the debug architecture. Our experimental results show that our work is specification-compliant as we are able to start a debugging session using our library, SableVM and Eclipse, an independent open source Java debugger.

Key words : Java, debug interface, debug architecture, virtual machine, SableVM, JDWP, JVMDI, JPDA, JVMTI, JRE

Chapitre I

Introduction

1.1 Contexte et problématique

L'évolution du développement de logiciels tend vers une croissance de la complexité des produits, une diminution du cycle de développement et des exigences du client toujours plus hautes. Dans ce contexte, l'importance des phases de débogage, test et vérification est accrue. Ainsi, la proportion du coût total de développement qui leur est allouée peut varier, selon les cas, de 50 à 75% (Hailpern et Santhanam, 2002).

La quête perpétuelle d'outils de débogage permettant un accès plus prompt à l'erreur est une conséquence de cette tendance. Les recherches effectuées quant aux nouveaux paradigmes de débogage illustrent cette volonté d'amélioration du rendement des phases de débogage : débogage bidirectionnel (Cook, 2002), delta débogage (Cleve et Zeller, 2005) ou vue en coupe (Zhang et Gupta, 2004) sont autant de techniques permettant une localisation plus rapide des erreurs. Certaines techniques peuvent être propres à un langage de programmation (C/C++, Java, etc), propres à un paradigme de programmation (orientée objet, procédurale, etc) ou génériques. Dans tous les cas, un outil permettant l'utilisation de cette technique doit être développé.

Le langage Java est un langage de haut niveau. Destiné à être exécuté sur une machine virtuelle, sa principale caractéristique est l'indépendance du code vis-à-vis de la machine sur laquelle ce dernier sera exécuté. Cette propriété est désirable du fait de la suppression des tâches de portabilité inhérentes à certains autres langages (C++, par

exemple) : le programmeur ne développe plus son application pour une machine spécifique, mais en vue de l'exécution sur une machine virtuelle Java, dont le comportement est normalisé (Lindholm et Yellin, 1999).

Lors du débogage classique, l'outil de débogage interagit avec le système d'exploitation, afin de recueillir des données sur l'état de l'environnement (lecture de la valeur de variables en mémoire, par exemple) ou d'influer sur l'exécution de l'application (suspension, reprise...). Pour ce faire, le débogueur utilise des routines mises à sa disposition par le système d'exploitation, variant selon l'architecture du système hôte (système d'exploitation, type de processeur...). Dans le cas de Java, du point de vue de l'application exécutée, la machine virtuelle agit en lieu et place du système d'exploitation. De fait, elle doit être munie de mécanismes suppléant les routines utilisées par le débogueur. Ici encore, un standard permet de spécifier les attributs de la machine virtuelle relatifs au débogage et codifie un protocole de communication en les différents outils intervenant dans l'exécution et le débogage Java (Sun, 2002).

Java a été le choix, ces dernières années, de nombreuses technologies (cellulaires, web, etc). Plus d'une décennie après la création du langage Java, sa popularité souffre toutefois des relations ambiguës entre la firme Sun Microsystems, détentrice du brevet, et la communauté du logiciel libre. Aujourd'hui, de nombreuses machines virtuelles Java (entités en charge de l'exécution du code Java compilé), libres ou commerciales, sont disponibles. Bien que la grande majorité des machines commerciales soient dotées de mécanismes permettant le débogage, aucune libre, en revanche, n'offre une telle fonctionnalité.

1.2 Objectifs

Nous proposons, dans cette étude, de développer la première architecture de débogage Java totalement libre. Pour ce faire, nous nous sommes basés sur l'architecture proposée par Sun Microsystems : la Java Platform Debugger Architecture (JPDA). Cette architecture a l'avantage d'offrir, entre autres, une claire séparation des différentes fonc-

tionnalités, facilitant la maintenance. Mais la principale raison de notre choix a été la compatibilité de cette architecture avec l'ensemble des outils conformes aux normes régissant le langage Java.

De nombreux éléments de cette architecture sont déjà disponibles librement. Ainsi, notre but n'est pas le développement d'un nouveau débogueur Java, ni la conception d'une nouvelle machine virtuelle. Nous utiliserons des débogueurs libres existants, conformes aux normes. D'autre part, nous nous limiterons à la modification d'une machine virtuelle Java libre existante, afin de lui apporter les attributs nécessaires au support du débogage. Nous avons opté pour l'utilisation de SableVM, une machine virtuelle développée en langage C et conforme aux normes Java.

De plus, nous proposons la mise en place d'un module indépendant de la machine virtuelle, en charge d'établir la communication entre débogueur et machine virtuelle. Ce module est également responsable de toutes les fonctionnalités propres au débogage, et dont l'implantation au sein de la machine virtuelle est, à nos yeux, inappropriée.

En dehors des réalisations pratiques, un des objectifs de ce mémoire est la mise en lumière des différentes étapes de la conception et de la réalisation des éléments de l'architecture de débogage Java. Ainsi, nous tenterons de jalonner le développement de ces éléments, en indiquant les relations existant entre les différentes entités en jeu.

Enfin, cette étude se veut une clarification partielle des normes régissant le débogage Java. En exposant les choix faits pour répondre aux différentes contraintes du standard suivi, nous présentons au lecteur les enjeux inhérents à l'implantation de certains concepts de débogage.

1.3 Contributions

Les principales contributions de cette étude sont de deux ordres : théorique et pratique. D'un point de vue théorique, les principales contributions sont :

- ce mémoire décompose les différentes phases de la mise en place d'une archi-

tecture de débogage Java. Le lecteur y trouvera un guide balisant la conception et le développement des éléments de cette architecture.

• les normes publiées par Sun Microsystems ne décrivant que les interfaces de chacun des éléments de l'architecture proposée, notre étude en présente les mécanismes sous-jacents, dévoilant « la partie immergée de l'iceberg »,

• une critique constructive des normes régissant le débogage Java. Nous proposons, à l'issue de ce mémoire, certaines propositions d'amélioration de la JPDA.

D'ordre plus pratique, on peut dénombrer les contributions suivantes :

• l'implantation au sein de SableVM des mécanismes nécessaires au support du débogage,

• la maintenance évolutive des mécanismes internes de SableVM, telle le remaniement de la gestion des processus, par exemple,

• la conception et le développement d'un module libre, indépendant de la machine virtuelle, en charge de la communication entre débogueur et machine virtuelle. Ce module gère également l'ensemble des objets manipulés lors du débogage ainsi que les événements générés par la machine virtuelle,

• la mise en place de la première architecture de débogage Java totalement libre. Plus précisément, nous avons démontré la possibilité de connecter les outils développés aux outils conformes aux normes, disponibles librement.

L'intégralité de travaux réalisés est publiée sur le site de SableVM[1]. Les directives permettant de récupérer et mettre en place l'architecture proposée sont disponibles en Annexe A.

[1] http ://sablevm.org

1.4 Structure du mémoire

Le reste de ce mémoire s'organise comme suit.

Le chapitre 2 introduit les notions préliminaires nécessaires à la compréhension de la suite de cette étude. Il aborde tout d'abord le concept de machine virtuelle, avant de présenter différents principes relatifs au débogage.

Le chapitre 3 présente l'architecture de débogage (Java Platform Debugger Architecture, JPDA) telle que décrite par les spécifications de Suns Microsystems. La fonction de chacun des éléments présents est sommairement décrite, ainsi que les interactions possibles entre chaque entité. Enfin, nous évaluerons les avantages et inconvénients d'une telle architecture afin de valider notre choix de la JPDA.

Le chapitre 4 relate de l'implantation de l'interface de débogage au sein de SableVM, notre machine virtuelle Java. Ainsi, après une présentation des fonctions présentes dans cette interface et des structures de données utilisées, nous donnerons les détails de l'implantation au sein de SableVM de chacune des fonctionnalités principales de débogage. Un survol des fonctions demandant un travail d'implantation moins complexe est ensuite effectué.

Le chapitre 5 traite du développement de la bibliothèque JDWP (Java Debug Wire Protocol), module prenant place entre la machine virtuelle et le débogage, dans l'architecture de débogage Java proposée par Sun Microsystems. Après une présentation de ce module, ce chapitre expose les choix techniques faits lors de la mise en place de la gestion des objets au sein de ce module. Puis nous décrirons la méthode utilisée pour le traitement des événements, avant de nous pencher sur le mécanisme d'exécution des commandes reçues par le biais de paquets en provenance du débogueur.

Le chapitre 6 retrace les tests effectués sur chacune des entités développées. Nous présentons dans ce chapitre comment nous avons validé nos travaux, tout au long du développement. La présentation des résultats des tests est suivie d'une liste de recommandations permettant d'améliorer les performances des modules créés. Enfin, ce cha-

pitre est conclu par nos suggestions quant aux nouvelles fonctionnalités à apporter à la JPDA.

Notre travaux sont basés sur le paradigme de débogage classique. L'extension des possibilités de SableVM en matière de débogage pourrait s'inspirer d'autres paradigmes. Le chapitre 7 présente quelques uns des nombreux paradigmes de débogage existants. Plus précisément, nous détaillons les concepts et applications du delta débogage, de la vue en coupe de programme et du débogage bidirectionnel.

Pour terminer, nous présentons notre conclusion.

Chapitre II

NOTIONS PRÉLIMINAIRES

Ce chapitre présente les notions nécessaires à la compréhension du reste de cette étude, portant sur l'interface de débogage de la machine virtuelle Java. Il apparaît donc capital de posséder des connaissances suffisantes des notions relatives d'une part au débogage, d'autre part aux machines virtuelles de manière générale.

La première partie de ce chapitre définit la notion de machine virtuelle, en présente les caractéristiques et sa place au sein d'un système informatique. On introduit ensuite SableVM, la machine virtuelle Java sur laquelle nos travaux ont été réalisés. La seconde partie traite des différentes notions liées au débogage, présentant les fonctionnalités élémentaires de tout outil, ainsi que le support matériel à la recherche d'erreurs.

2.1 Les Machines virtuelles

Dans cette section, nous présentons les notions générales relatives aux machines virtuelles, et nécessaires à la bonne compréhension de nos travaux.

Dans un premier temps, nous dressons un historique des machines virtuelles. Puis, nous présentons la place de la machine virtuelle au sein d'une système informatique. Ensuite, nous détaillons les propriétés communes à tous les types de machine virtuelle. Enfin, la dernière section est consacrée à la présentation de la machine virtuelle Java, plus précisément à SableVM, la machine virtuelle Java utilisée dans notre étude.

2.1.1 Historique

Les premiers systèmes informatiques étaient développés avec leur propre jeu d'instructions. Les logiciels (systèmes d'exploitation ou simples applications) étaient alors conçus pour chaque jeu d'instructions, en fonction du système d'accueil. Avec l'accroissement de la communauté d'utilisateurs, la rapidité d'évolution matérielle et la complexité croissante des applications développées, la réécriture et la distribution de logiciels pour chaque architecture devinrent bientôt une charge considérable. Les concepts de compatibilité et portabilité ont alors pris toute leur importance, menant à une séparation claire des attributs logiciels et matériels d'une machine et à la définition d'interface de communication entre ces deux couches. Cette interface est nommée architecture du jeu d'instructions (Instruction Set Architecture, ISA).

Les systèmes d'exploitation se sont vus attribuer, avec l'évolution des systèmes informatiques, des responsabilités quant à la protection des données, la gestion des applications en fonctionnement, la gestion des ressources matérielles, etc. De plus, l'avènement du réseau a fait qu'un système d'exploitation n'évolue plus dans une relative isolation, mais doit régir également le partage contrôlé et la protection des ressources et données sur ce réseau. Devenus les pierres angulaires du système informatique de par leur rôle central, les systèmes d'exploitation offrent aujourd'hui une multitude de services, exploités par les applications. Une dépendance forte est alors créée entre une application et le système d'exploitation sous-jacent. Bien que des efforts de standardisation aient été menés, la portabilité et la compatibilité des logiciels restent limitées aux propriétés de la machine (système d'exploitation, ressources matérielles...) pour laquelle une application a été développée (Smith et Nair, 2005).

Une machine virtuelle est un composant logiciel ajouté à la plateforme d'exécution, permettant à cette dernière d'apparaître comme une plateforme aux propriétés différentes, voire comme plusieurs plateformes. Ce concept fut introduit dans les années 1960, afin de s'affranchir des contraintes matérielles et d'offrir un degré plus élevé de portabilité et de flexibilité. Une machine virtuelle peut supporter un système d'exploi-

tation ou un jeu d'instructions qui diffèrent de ceux disponibles sur la machine réelle sous-jacente.

La machine virtuelle peut être définie comme une couche d'abstraction. Pouvant s'immiscer entre deux couches de l'architecture du système, elle prend alors l'apparence de la couche inférieure aux yeux de la couche supérieure, et présente des propriétés différentes de celles effectivement disponibles (d'où son utilité). Elle reste la cible des programmeurs et systèmes de compilation : une application est aujourd'hui développée pour être exécutée sur une machine virtuelle. Plutôt que d'écrire une application pour chaque architecture, on écrit une application pour une machine virtuelle, et une machine virtuelle pour chaque architecture.

2.1.2 La Machine virtuelle au sein du système

L'architecture d'une machine peut être divisée en plusieurs couches, ou abstractions. Ces couches sont mises en place afin de dissimuler les détails des couches adjacentes. On peut ainsi découper un système en trois couches : la couche matérielle, la couche système d'exploitation et la couche application (figure 2.1). La couche matérielle est communément représentée par la couche basse ou inférieure, la couche application étant la couche supérieure, ou haute. Dans cette représentation par couches, on peut considérer la machine virtuelle comme une couche supplémentaire intercalée entre deux couches existantes.

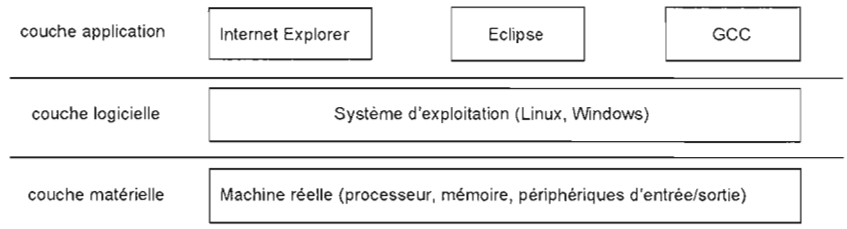


Fig. 2.1 Couches d'abstraction des machines modernes (Rosenblum, 2004)

Les couches sont munies d'interfaces normalisées, permettant aux couches adja-

centes un accès à ses ressources. Cette standardisation des interfaces, plus généralement la séparation des fonctions d'une machine en couches, possède avantages et inconvénients, présentés ci-après.

Tout d'abord, cette architecture en couches sépare clairement les fonctionnalités d'un système informatique. Ainsi, elle permet aux concepteurs de matériel et de logiciel d'œuvrer indépendamment. Les innovations, et modifications apportées à une couche ne remettent pas systématiquement en cause la conception des applications destinées à des couches différentes. De plus, un programme peut fonctionner sur tout type de support respectant la norme de communication utilisée lors de son développement.

Cependant, beaucoup voient dans cette harmonisation une limitation. En effet, un programme ne peut uniquement fonctionner sur une couche respectant la même norme de communication. Par exemple, une application compilée pour Linux ou Windows effectue des appels différents aux routines du systèmes d'exploitation. Si cette application a été développée pour une architecture Windows / Intel x86 par exemple, elle ne pourra pas être exécutée directement sur une plateforme Linux / Intel x86. Cette constatation est d'autant plus vraie si l'on considère un réseau d'ordinateurs utilisant des interfaces différentes : une application dont l'utilisation est restreinte à un support précis ne permet pas de tirer bénéfice du partage de ressources, spécialement dans un réseau aussi vaste que l'internet.

2.1.3 Propriétés des machines virtuelles

Bien que les machines virtuelles puissent être conçues pour des niveaux d'abstraction différents, elles partagent tout de même un socle commun de propriétés. Nous les présentons dans cette section.

2.1.3.1 Compatibilité

L'attribut principal des machines virtuelles est la compatibilité des logiciels. Une machine virtuelle doit se comporter comme la couche qui la supporte. De fait, toute

application développée pour être exécutée sur cette couche pourra être exécutée par la machine virtuelle. Par exemple, une machine virtuelle de niveau matériel pourra exécuter tous les logiciels et systèmes d'exploitation destinés à ce matériel.

2.1.3.2 Isolation

Une machine virtuelle offre également une isolation quant aux autres machines virtuelles ou réelles. En plus de permette un isolement des données, cette propriété protège les autres machines virtuelles et réelles des programmes malicieux pouvant être exécutés sur une machine virtuelle. La machine virtuelle se voit attribuer par le système d'exploitation une certaine quantité de ressources (temps CPU, mémoire, espace disque...) à son démarrage, et ne peut en aucun cas passer outre ces limites. Une machine virtuelle peut alors utiliser toutes les ressources qui lui sont attribuées sans dégrader les performances des autres applications concurrentes. L'utilisation de plusieurs machines virtuelles peut ainsi permettre de pallier l'iniquité dans l'attribution des ressources par le système d'exploitation.

2.1.3.3 Encapsulation

Une troisième propriété des machines virtuelles est l'encapsulation : on ajoute un niveau d'indirection dans la communication entre les couches. Cette indirection peut être utilisée à d'autres fins que la simple compatibilité. Elle permet à la machine virtuelle d'offrir un meilleur environnement d'exécution, comme une tolérance accrue aux fautes et erreurs de programmation : les machines virtuelles pour les langages de haut niveau permettent d'effectuer des vérifications lors de l'exécution et ainsi éliminer certains types d'erreurs (les erreurs de type, par exemple). De plus, les machines virtuelles modernes exploitent cette position afin de proposer de nouveaux services, tels que la prise en charge de la gestion de la mémoire (on pense notamment au ramasse-miettes Java). Cette indirection permet aussi la collecte d'informations, pouvant être exploitées pour effectuer une analyse de performance ou à des fins de débogage, par exemple.

2.1.3.4 Surcoûts

L'ajout d'une couche additionnelle peut dégrader les performances en introduisant des coûts supplémentaires. Cependant, dans certains cas, les bénéfices induits par l'utilisation de machines virtuelles comblent largement ces surcoûts. Par exemple, les applets Java d'une page internet peuvent être exécutées sur tout navigateur supportant ce type d'application, sans que le développeur n'ait à se soucier du type de machine utilisée lors de la visite de son site internet. De plus, dans le cadre du développement de logiciels, les machines virtuelles de niveau matériel permettent de faire cohabiter simultanément des systèmes d'exploitation différents. Les phases de développement, de test et de déploiement sont alors grandement facilitées.

2.1.4 SableVM : une machine virtuelle Java

Cette section est consacrée à SableVM[1], la machine virtuelle Java utilisée dans notre étude. SableVM est une machine virtuelle Java libre respectant les normes établies. Elle a été développée en langage C, suivant la norme ANSI, et minimise ainsi le volume de code dépendant de la machine hôte. Cette caractéristique permet un portage rapide à de nouvelles plateformes : SableVM a été portée sur une nouvelle plateforme en moins d'une heure.

Dans le reste de cette étude, de nombreuses références sont faites aux mécanismes et structures de données internes de SableVM. Il est donc important de familiariser le lecteur avec les spécificités de cette machine virtuelle.

Dans un premier temps, nous décrivons la place de la machine virtuelle Java au sein de l'architecture informatique. Ensuite, nous présentons les mécanismes internes de SableVM manipulés durant nos travaux : fenêtres d'exécution, procédé d'exécution du code octet et gestion des processus.

---

[1] http ://sablevm.org

2.1.4.1 La JVM : une machine virtuelle de haut niveau

Les machines virtuelles de haut niveau sont ainsi nommées de par leur position dans les couches supérieures de l'architecture (couches logicielles) : elles sont exécutées au-dessus du système d'exploitation, au même titre que les autres applications. Elles sont destinées à exécuter des langages de haut niveau (figure 2.2). Ces derniers sont souvent décrits comme facilement compréhensibles par l'utilisateur, dont les instructions sont indépendantes de la machine sur laquelle ils sont exécutés, et dont chaque instruction correspond généralement à plusieurs instructions en code machine.

Java et Smalltalk sont des exemples de langages exécutés sur des machines virtuelles de haut niveau.

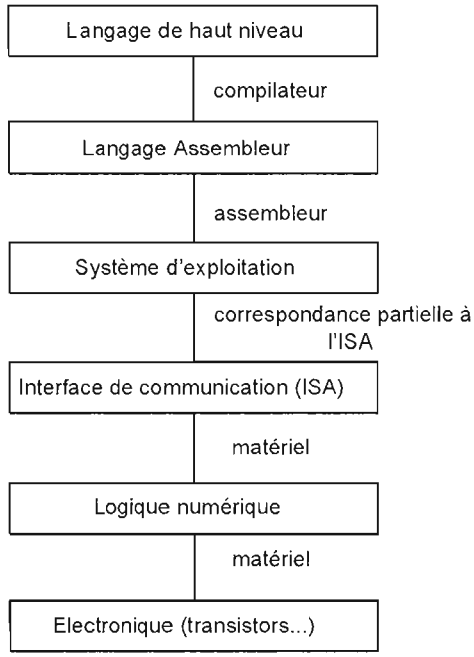


Fig. 2.2 Couches d'abstraction détaillées

Par opposition, les simulateurs sont des machines virtuelles de bas niveau, puisque ceux-ci sont destinés à résoudre le problème d'absence de matériel. Du point de vue

de la couche système d'exploitation, la machine virtuelle apparaît alors comme étant la couche matérielle. On observe aujourd'hui un regain de popularité de ce type de machines, avec l'apparition des produits de la game VMWare[2], présentés comme des logiciels d'infrastructure virtuelle.

La JVM, ou machine virtuelle Java, est soumise à certaines spécifications (Lindholm et Yellin, 1999) décrivant ses divers sous-systèmes. Cette description est fonctionnelle, et ne précise pas de quelle manière les fonctions décrites doivent être implémentées. Une certaine latitude est donc laissée aux concepteurs de machines virtuelles Java, allant du choix du langage de programmation aux types des structures de données internes utilisées. La machine virtuelle est donc une sorte de « boîte noire » exécutant le code Java, communicant avec son environnement par le biais d'interfaces normalisées.

Dans les sections suivantes, nous présentons les mécanismes internes de la machine virtuelle Java SableVM. Tout d'abord, nous définissons ce qu'est un contexte d'exécution. Puis, nous détaillons les mécanismes d'exécution du code octet. Enfin, nous présentons la gestion des processus au sein de SableVM.

2.1.4.2 Contexte d'exécution

Un contexte d'exécution (ou *frame*) est une structure de données destinée à accueillir, entre autres, des données et résultats partiels, à renvoyer les valeurs retournées lors de l'invocation d'une méthode, etc.

Chaque processus possède un pile dans laquelle sont conservées ces contextes d'exécution en cours de validité. A chaque appel de fonction, un contexte d'exécution est ajouté au-dessus de la pile des contextes des appels précédents. Celui-ci n'est supprimé que lors de l'achèvement de l'exécution de cette fonction, que ce soit de manière normale ou suite à la génération d'une exception.

Trois types de contexte d'exécution existent au sein de SableVM : les contextes

---

[2]http ://www.vmware.com

Java, les contextes natifs et les contextes internes. Chacun possède une structure radicalement différente des autres. Les contextes Java contiennent, par exemple, les paramètres d'appel de la méthode et ses variables locales. Les contextes natifs comportent, entre autres, un groupe de références natives. Les contextes Java et natifs sont créés respectivement lors de l'appel de méthodes en Java ou en code natif. Les contextes internes sont utilisés pour des besoins internes uniquement. Il en existe trois sous-types : le premier correspond à l'appel initial de la machine virtuelle. Le second est utilisé lorsque la machine virtuelle effectue des appels Java pour ses propres besoins. C'est par exemple le cas lors d'une initialisation static en Java. Enfin, le dernier type est intercalé entre les contextes Java et natifs[3].

2.1.4.3 Exécution du code octet

Lors de la compilation, le code Java est transformé en un ensemble d'instructions plus abstraites que le code machine et pouvant être exécutées par n'importe quelle machine virtuelle respectant les normes : le code octet (figure 2.3). Ce code est ainsi nommé car les instructions sont composées d'une opération longue d'un octet, suivie des paramètres. Le code octet est un langage intermédiaire destiné à réduire (voire supprimer) la dépendance au matériel et faciliter l'interprétation.

SableVM ne se contente pas de convertir le code octet en code machine afin de pouvoir l'exécuter. Une étape supplémentaire est introduite, la conversion du code octet en code spécifique à SableVM. Chaque instruction est alors placée dans un mot (dont la longueur dépend de la machine sur laquelle le code est exécuté), plutôt que dans un octet. Durant cette opération, SableVM réalise également quelques optimisations, réduisant les calculs effectués lors de l'exécution (Gagnon, 2002).

SableVM comprend trois méthodes d'interprétation du code octet, décrites sommairement ci-après.

---

[3]Pour plus de détails concernant les structures de données internes, consulter la documentation de SableVM, disponible sur http ://sablevm.org

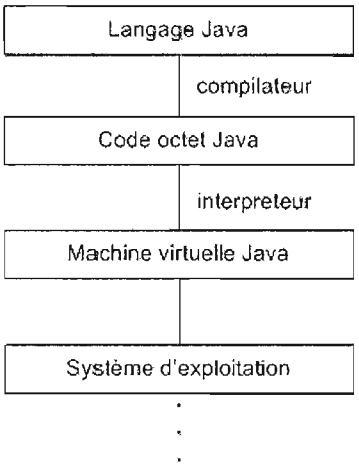


Fig. 2.3 Couches d'abstraction Java

L'interpréteur *Switch* charge le code octet et le stocke dans une structure de données (généralement un tableau) lui permettant d'accéder aux instructions de manière séquentielle. Puis il utilise une boucle de type switch en langage C (d'où son nom) afin de déterminer quelles actions effectuer pour chaque instruction.

Le *direct-threading* est une technique basée sur la précédente, insérant dans le code des instructions de saut vers les adresses des implémentations de chacune des instructions.

Enfin, la méthode *inline* (Piumarta et Riccardi, 1998) permet de remplacer certains appels de méthodes par une copie du corps de ladite méthode, réduisant le coût induit par les différents sauts.

2.1.4.4 Gestion des processus

SableVM utilise un ramasse-miettes[4] (ou *garbage collector*) afin de recycler les objets non utilisés. Certaines stratégies de récupération de la mémoire impliquent un

[4]Depuis notre étude, plusieurs types de ramasse-miettes ont été implantés au sein de SableVM.

déplacement des objets conservés. C'est par exemple le cas du ramasse-miettes copieur (ou *copying garbage collector*). Dans ce cas, les références vers les objets sont modifiées. C'est pourquoi on ne peut accomplir de récupération de mémoire durant l'exécution du code Java. La machine virtuelle doit alors être dotée d'un mécanisme de suspension de l'exécution Java afin de pouvoir effectuer cette opération de collecte d'objets inutilisés.

Au sein de SableVM, lorsque le ramasse-miettes veut commencer sa tâche, on doit s'assurer qu'aucun processus exécutant du code Java n'est actif. Un processus demande alors à tous les autres de suspendre leur exécution, et attend que sa demande soit prise en compte avant de se suspendre lui-même. Régulièrement, chaque processus vérifie si une requête de suspension a été faite. Si c'est le cas, il suspend son exécution et attend ensuite un ordre de reprise. En plus des mécanismes de notification, les processus sont dotés d'un indicateur d'état : lorsqu'un processus est suspendu, son statut est modifié. Le diagramme de la figure 2.4 résume les statuts possibles d'un processus au sein de SableVM.

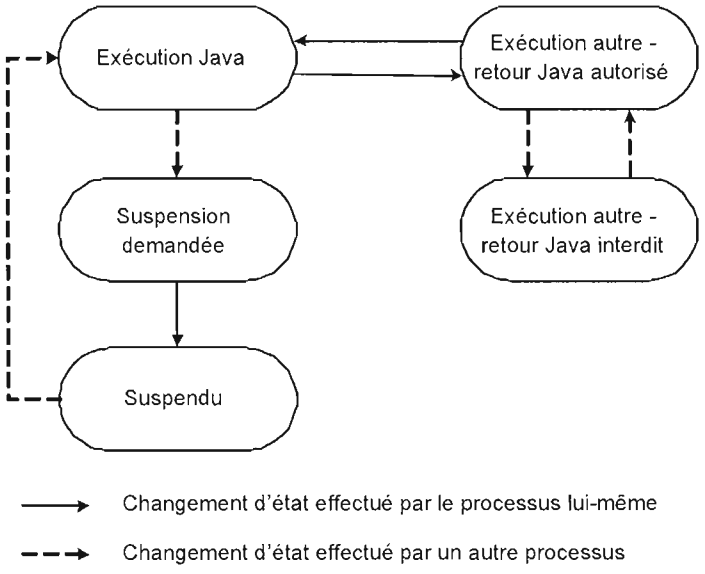


Fig. 2.4 États possibles des processus dans SableVM

Le procédé de demande de suspension et de reprise a été implanté dans SableVM

en utilisant des méthodes de notification générale. Plus précisément, ce système est basé sur les routines *pthread*, utilisant une variable de condition pour la transmission des diverses notifications[5].

2.2 Principes de débogage classique

Depuis l'introduction des premiers systèmes informatiques, le débogage, procédure consistant à identifier et éliminer les erreurs, a toujours représenté une partie importante du cycle de vie d'un logiciel. Ces erreurs sont décelées lorsqu'une différence apparaît entre le comportement observé et le comportement attendu d'un système. Bien entendu, pour ce faire, on considère que le comportement du logiciel est au préalable clairement défini.

De nombreux outils sont à la disposition des utilisateurs pour les assister dans cette recherche d'erreurs, des plus rudimentaires jusqu'aux systèmes dotés d'intelligence. Cette section présente le socle commun de fonctionnalités et principes des débogueurs. La première partie traite des principes fondamentaux des outils existants. La partie suivante décrit les fonctionnalités élémentaires présentes dans tous les outils. Pour finir, nous présentons les différents supports fournis par le matériel informatique afin d'améliorer les fonctions de débogage (certaines machines possèdent, par exemple, des registres dédiés au débogage).

2.2.1 Principes fondamentaux

Si aucune norme officielle ne gouverne les comportements des débogueurs, deux principes de « bon sens » existent. Ces principes sont relatifs d'une part à l'interaction entre l'outil et l'application déboguée, d'autre part au comportement du débogueur, indépendamment de l'application testée.

Le premier de ces principes est le principe d'Heisenberg, ainsi nommé car dérivé du principe d'incertitude d'Heisenberg (Heisenberg, 1930). Il régit l'interaction entre le

[5]Une variable de condition n'est pas verrou, mais est associée à un verrou. Elle est fournie par la bibliothèque standard et est utile pour la mise en place de la synchronisation de données.

débogueur est l'application examinée. Ce principe spécifie que le débogueur ne doit en aucun cas influer sur le comportement de l'application (exception faite de la suspension et de la reprise de l'exécution). L'application a un comportement similaire lors de la phase de débogage et lors d'une exécution autonome. Il serait en effet problématique de rechercher la cause d'un comportement inattendu en utilisant un outil modifiant le comportement de l'application. Le débogueur peut néanmoins communiquer avec le système exécutant l'application : il lui est possible, par exemple, de demander l'état d'une zone mémoire utilisée par l'application, afin de connaître la valeur d'une variable.

Le second principe suggère que les informations présentées par l'outil à l'utilisateur doivent être celles effectivement manipulées par l'application. Prenons le cas d'une valeur chargée et modifiée dans un registre, sans réécriture sur disque. L'application utilise alors la valeur en registre, alors que celle sur disque diffère. Il serait futile de présenter à l'utilisateur la valeur présente sur le disque, puisque cette valeur ne refléterait pas l'état du programme à cet instant. Le comportement de l'application pourrait alors apparaître absurde aux yeux de l'utilisateur, alors qu'il serait censé une fois la valeur modifiée connue.

Ces deux principes restent généraux, et, une fois encore, aucune norme ne vient contraindre le respect de ceux-ci. Il est toutefois difficilement pensable qu'un utilisateur puisse trouver un support quelconque dans l'utilisation d'outils ne satisfaisant pas ces deux règles.

2.2.2 Fonctionnalités élémentaires

Si la fonction première d'un débogueur est de faciliter la recherche d'erreurs dans un programme, certains de ces outils ont été conçus afin de répondre à un besoin spécifique, dans un contexte précis. Pour des langages de programmation déléguant à l'utilisateur la gestion de la mémoire, on a vu apparaître des outils permettant de localiser exclusivement les irrégularités d'accès à la mémoire, par exemple (Valgrind[6], pour n'en

[6] http ://valgrind.org

citer qu'un).

Les outils existants possèdent des fonctionnalités qui leur sont propres. Dans cette diversité de fonctionnalités, on retrouve toutefois des concepts récurrents. Ce sont ces différentes fonctions que nous présentons dans cette section.

2.2.2.1 Points d'arrêt

La difficulté de localisation d'une erreur est souvent proportionnelle au temps écoulé entre l'exécution d'un code erroné et l'observation d'un comportement anormal de l'application. C'est pourquoi il peut être utile de suspendre l'exécution d'un programme juste avant ou après l'observation d'un tel comportement. Pour ce faire, les débogueurs utilisent des points d'arrêt, définis par l'utilisateur. Il en existe trois types, *breakpoint*, *catchpoint* et *watchpoint*[7], que nous allons définir dans cette section.

Un *breakpoint* permet de suspendre l'exécution de l'application lorsqu'un certain emplacement dans le code est atteint. En général, on spécifie l'emplacement souhaité en indiquant le numéro de la ligne dans le code source, le nom de la fonction ou l'adresse mémoire de l'instruction exécutée. Il est également possible de coupler l'emplacement souhaité à une condition supplémentaire, afin d'avoir un contrôle plus fin sur l'instant de suspension désiré. Cette condition peut prendre des formes différentes, selon les capacités de l'outil utilisé. Par exemple, il est possible d'indiquer un arrêt lors de la nième exécution d'une instruction, utile lorsque le point d'arrêt se situe au cœur d'une boucle. La mise en place de breakpoints est souvent conditionnée à la présence d'informations relatives au code source de l'application, afin d'établir la correspondance entre code octet et ligne de code Java.

Un *catchpoint* permet la suspension de l'exécution d'un programme lorsqu'un événement survient. Par événement, on entend par exemple le soulèvement d'exceptions en Java ou C++. Dans ces événements, on ne considère pas les signaux envoyés par le

[7] Aucune traduction française convenable n'existe, nous utiliserons donc les termes anglais, également utilisés par les versions françaises des outils de débogage.

système d'exploitation. Dans ce cas, en effet, un *handler* est responsable des actions. Il serait donc inapproprié d'effectuer cette gestion deux fois.

Un *watchpoint* permet de spécifier l'adresse mémoire dont l'accès entraînera une suspension de l'exécution. Dans le cas de Java, on ne spécifie pas d'adresse mémoire, celle-ci pouvant changer suite à un recyclage de la mémoire (ou *garbage collection*). On indique plutôt le champ dont l'accès engendrera une suspension de l'exécution. Les *watchpoints* peuvent êtres utiles si le programme n'évolue pas de façon séquentielle, rendant la localisation temporelle de l'accès difficile. C'est le cas dans la programmation orientée objet, où l'exécution « saute » d'une méthode à une autre.

En plus des informations relatives au code source, la disposition des points d'arrêt peut être liée à l'exécution de l'application. Dépendamment des systèmes, certains outils ne permettent pas de poser un point d'arrêt avant l'exécution de toute ou partie de l'application. Ces contraintes sont généralement dues au chargement tardif des dépendances de l'application. Par exemple, le débogueur GDB sur les systèmes HP-UX, ne permet pas de disposer un point d'arrêt dans les bibliothèques partagées n'étant pas appelées directement par le programme, avant l'exécution de l'application (Stallman, Pesch et Shebsr, 2002).

2.2.2.2 Progression pas à pas

Lors de son investigation, l'utilisateur émet une hypothèse quant à l'emplacement de l'erreur et suspend, grâce aux différents moyens à sa portée, l'exécution du programme. Il est rare que la suspension permette de repérer exactement l'emplacement de la faute, aussi il peut être commode d'avoir des moyens permettant l'inspection des environs immédiats du point de suspension. Ainsi, la plupart des outils de débogage permettent une exécution du programme étape par étape, laissant à l'utilisateur le choix de la taille du pas.

La commande *step in* correspond à la granularité la plus fine de déplacement. Cette commande permet d'avancer jusqu'à l'instruction suivante. Dans le cas d'un ap-

pel à une fonction, cette instruction s'arrêtera après la première instruction de la fonction appelée. Selon les choix d'implémentation des outils, la notion d'« instruction suivante » peut varier. Dans le cas du langage Java, traduit en codes octets exécutables, cette notion peut correspondre à l'exécution d'une instruction en code octet, chaque instruction comprenant une commande et ses arguments. D'autres outils interprètent cette notion comme le passage à la ligne suivante, une ligne de code pouvant contenir plusieurs instructions. Dans tous les cas, step in permet l'avancée au pas le plus petit autorisé par le système.

La commande *step over*, aussi appelée next, permet de continuer l'exécution du programme jusqu'à la ligne suivante. Cette fonction est similaire à la précédente, à l'exception du comportement face à un appel de fonction : cette commande continue l'exécution jusqu'au retour de la fonction appelée, au lieu de s'arrêter dès la première instruction dans le corps de la fonction.

La commande *step out* continue l'exécution jusqu'à la sortie de la fonction courante. Cette fonction peut être utile dans le cas où l'exécution se trouve dans une fonction dite « feuille », c'est-à-dire n'effectuant aucun appel de fonction, ou dans le corps d'un fonction dont on sait pertinemment qu'elle ne modifie pas l'état du programme (fonction d'affichage, par exemple).

A l'instar des points d'arrêt, l'utilisation des différentes fonctions d'avancée étape par étape peut dépendre de la disponibilité d'informations concernant le code source de l'application. Là encore, des différences existent selon les choix d'implémentation et de conception des outils utilisés.

Lorsque l'inspection est terminée, la commande *continue* permet la reprise de l'exécution normale du programme, jusqu'au prochain point d'arrêt.

Ces commandes peuvent, selon les outils, arborer des noms différents, mais les principes restent identiques. De plus, d'autres fonctionnalités composées de celles présentées ici peuvent être offertes. Certains outils permettent également d'indiquer le

nombre de sauts désirés. Il devient possible de permettre l'exécution des n prochaines instructions d'un programme, par exemple. Combiné au type de pas souhaité, l'utilisateur a donc à sa disposition un large éventail de possibilités de déplacement.

2.2.2.3 Inspection des états d'un programme

Lorsque l'exécution du programme est suspendue, il peut être avantageux d'inspecter l'état du programme, ensemble des valeurs des différentes variables visibles à un moment donné. Les outils de débogage offrent alors plusieurs points de vue, permettant l'inspection de ces états et donc de déceler les anomalies.

Le code source du programme, lorsque disponible, est le meilleur point de vue afin de comprendre le comportement d'un programme. Rares sont les outils ne permettant pas une telle lecture. Un indicateur permet, dans la majeure partie des cas, de pointer la ligne d'exécution courante, permettant à l'utilisateur de faire la corrélation entre une ligne de code et le comportement de son application.

Le second point de vue consiste à examiner le contenu des variables à un instant donné. Selon le degré de perfectionnement du débogueur, cette affichage pourra prendre différentes formes, présentant les données telles que présentes dans la machine, ou de manière plus intelligible. Par exemple, l'affichage d'un entier peut-être fait de manière binaire ou hexadécimale, proche de la représentation machine, ou de manière décimale, plus compréhensible par l'utilisateur. DDD[8] (Data Display Debugger) permet, par exemple, l'affichage de structures de données complexes (figure 2.5).

L'impression de la pile de fenêtres d'exécution s'avère un moyen efficace de retracer l'exécution d'un programme. Cette liste d'appels prend la forme d'une liste de noms de fonctions, pouvant également afficher les valeurs des paramètres passés.

Plus rarement, certains outils permettent l'affichage des valeurs des différents registres du processeur. Ce procédé est utile dans le cas de la programmation système,

---

[8]http ://www.gnu.org/software/ddd/

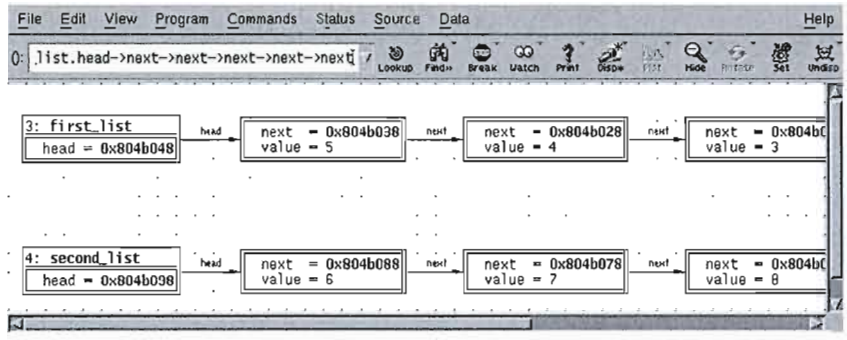


Fig. 2.5 Affichage d'une liste chaînée par DDD

ou de bas niveau. Cependant, cet affichage n'est pas des plus courants, puisqu'il engendre une dépendance forte entre l'outil de débogage et le matériel le supportant.

Dans tous les cas, la possibilité d'accès aux données, à un moment précis de l'exécution, représente un atout majeur dans la recherche d'erreur.

2.2.3 Support matériel au débogage

Toute application interagit de manière plus ou moins importante avec le système qui la supporte (système d'exploitation ou machine virtuelle). Afin de contrôler toutes ces interactions, l'outil de débogage doit lui aussi collaborer étroitement avec le système sous-jacent. Cette collaboration peut descendre jusqu'aux couches matérielles de la machine hôte. Ainsi, des composants matériels peuvent intervenir afin de faciliter la tâche au débogueur, ou simplement accélérer le traitement lors du débogage.

Dans cette section, nous présentons tout d'abord les moyens matériels existants afin de faciliter la mise en place de points d'arrêt. Puis, nous décrirons les mécanismes existants afin de supporter l'exécution d'une application pas à pas. Enfin, nous détaillons le support offert par la machine virtuelle Java.

Notez qu'il existe une alternative logicielle à tous les moyens matériels présentés dans cette section.

2.2.3.1 Support matériel aux points d'arrêt

Plusieurs solutions matérielles existent afin de mettre en place des points d'arrêt. La liste présentée dans cette section n'est pas exhaustive. Nous présentons ici la plus commune : l'utilisation de registres dédiés.

Le processeur possède des registres dédiés aux points d'arrêt. Ces registres conservent les adresses des instructions auxquelles l'exécution doit être suspendue. Lorsque le programme atteint une adresse correspondant à celle inscrite dans le registre, une exception est générée et reportée au débogueur. Ce dernier peut alors effecturer le traitement approprié (suspension, affichage de message...). Ces registres ne doivent pas être accédés directement par l'application, puisque seul l'utilisateur doit être en mesure de spécifier les points d'arrêt désirés. Un système de protection adéquat est alors nécessaire, afin de ne pas permettre la modification des valeurs contenues lors de l'exécution du programme.

Pour le cas particulier des *watchpoints*, l'utilisation de ces registres est impossible. Si la taille de la zone mémoire à surveiller est supérieure à celle disponible sur le processeur, on sera dans l'obligation d'avoir recours aux *watchpoints* logiciels. Dans une architecture x86, les huit registres $DR0$ à $DR7$ sont dédiés au débogage. Parmi eux, les registres $DR0 - 3$ sont réservés aux points d'arrêt, accessibles par l'utilisateur (Intel, 1995). Chaque registre pouvant contenir jusqu'à quatre octets de mémoire, la zone à surveiller ne peut dépasser 16 octets. L'utilisation de la version logicielle est également imposée lorsque l'expression à surveiller dépend d'une valeur contenue dans un autre registre.

Les registres dédiés au débogage sont partagés par toutes les fonctionnalités : il n'existe en général pas de registre exclusivement réservé aux *breakpoints*, ou exclusivement réservé au *watchpoints*. Les points d'arrêt sont dès lors souvent limités en nombre. Lors du dépassement du nombre de points d'arrêt autorisés, et lorsque le matériel informe les couches supérieures de ce dépassement, il revient au débogueur de prendre en

charge les points d'arrêt supplémentaires, définis alors de manière logicielle.

2.2.3.2 Support matériel à l'exécution étape par étape

La plupart des processeurs modernes disposent d'une section réservée aux indicateurs (ou *flags*). Un indicateur est alors réservé à l'exécution pas à pas. Lorsque cet indicateur est à 1, une interruption intervient après l'exécution de chaque instruction. Cette interruption est ensuite retransmise au débogueur qui décide des actions à mener. Ces indicateurs ne peuvent être modifiés que par des applications privilégiées, afin de protéger le système contre une utilisation intempestive de l'exécution pas à pas.

La hiérarchisation des interruptions permet au système d'arrêter l'exécution étape par étape lorsqu'une interruption différente intervient. Si deux interruptions interviennent simultanément, celle correspondant à l'avancée pas à pas est traitée d'abord. A la fin de ce traitement, l'indicateur est supprimé (passe à la valeur 0). On peut alors traiter la seconde interruption sans que ce traitement ne soit interrompu après chaque instruction.

2.2.3.3 Support offert par la machine virtuelle Java

La grande partie des systèmes d'exploitation existants sont dotés de mécanismes de communication, et d'interruptions. Ces mécanismes sont, entre autre, utilisés afin d'alerter le débogueur des interactions entre le système d'exploitation et l'application.

Dans le cas de débogage Java, l'application est exécutée sur une machine virtuelle. Elle n'interagit donc pas directement avec le système d'exploitation, où la couche inférieure de manière générale. La machine virtuelle, agissant en lieu et place du système d'exploitation, doit alors être dotée des mêmes mécanismes de support au débogage.

Dans l'architecture de débogage proposée par Sun Microsystems, les signaux et interruptions sont remplacés par un système d'événements, envoyés par la machine virtuelle au débogueur. Ces événements informent le débogueur de l'activité de l'application au sein de la machine virtuelle. Par exemple, l'événement JVMDI_EVENT_SINGLE_STEP

permet de notifier que l'application est exécutée pas à pas. Il est envoyé au débogueur après l'exécution par la machine virtuelle de chaque instruction du programme, lorsque l'avancée étape par étape est activée. Cet événement correspond à l'utilisation du signal PTRACE_SINGLESTEP, ayant le même but, émis par le système d'exploitation.

En plus du panel d'événements disponibles, la machine virtuelle doit également être dotée de fonctions suppléant les routines systèmes fournies par le système d'exploitation. Afin de définir un point d'arrêt, la machine virtuelle met à la disposition du débogueur la fonction SetBreakpoint(...), permettant de jouer le rôle de la routine PTRACE_SETREG accédant aux registres afin d'y stipuler l'adresse du point d'arrêt.

2.3 Conclusion

Dans ce chapitre, nous avons présenté les notions nécessaires à la bonne compréhension du reste de ce mémoire.

Dans un premier temps, nous avons décrit les machines virtuelles, leur historique et leurs propriétés, avant de préciser les caractéristiques de la machine virtuelle Java. Enfin, nous avons introduit les spécificités de SableVM, la machine virtuelle Java sur laquelle nos travaux sont basés.

Puis, nous avons introduit les différents principes du débogage classique, suivis de la présentation des fonctionnalités élémentaires des débogueurs. Enfin, nous avons décrit le support matériel offert au débogage, et son équivalent au sein de la machine virtuelle Java.

Chapitre III

ARCHITECTURE

Si la machine virtuelle Java est déjà un socle permettant l'exécution de programmes écrits en Java, il est possible de la doter de fonctionnalités lui permettant d'élargir son champ d'action. Ainsi, nous avons permis à SableVM, la machine virtuelle Java sur laquelle nous avons basé nos travaux, d'offrir un ensemble de fonctions relatives au débogage d'applications Java. Pour ce faire, nous avons implanté l'architecture proposée par Sun Microsystems au cœur de la machine virtuelle. Nous présentons dans ce chapitre l'architecture choisie, ainsi que les raisons ayant motivé notre décision.

La première partie de ce chapitre est consacrée à la présentation globale de l'architecture proposée par Sun Microsystems, offrant une vue d'ensemble de tous les éléments la constituant. La section suivante détaille la portion de cette architecture dédiée à la machine virtuelle Java. Les fonctionnalités offertes par le protocole de communication sont exposées dans la troisième partie. La quatrième partie est dédiée à la bibliothèque JDWP, chargée entre autre de la transcription des commandes envoyées et reçues par la machine virtuelle. Enfin, nous présentons, dans une cinquième partie, les avantages et inconvénients liés au choix de cette architecture.

3.1 Présentation de l'architecture de débogage Java

Le langage Java est un langage de haut niveau, destiné à être exécuté sur une machine virtuelle. Afin de garantir un comportement identique des applications Java,

indépendamment du système sous-jacent, les machines virtuelles, en charge de l'exécution du code Java, doivent respecter certaines spécifications (Lindholm et Yellin, 1999).

Dans le même souci d'offrir des services identiques en matière de débogage, Sun Microsystems a proposé une architecture standard : la Java Plateform Debugger Architecture (JPDA). Cette architecture compte deux fonctionnalités principales, l'une exécutant l'application Java, l'autre en charge du débogage. Ces deux entités échangent de l'information en respectant un protocole de communication, également décrit dans la JPDA.

Cette section propose tout d'abord une vue d'ensemble de la JPDA. Puis, les parties suivantes présentent les différents éléments de cette architecture.

3.1.1 Vue d'ensemble de la Java Platform Debugger Architecture

La Java Platform Debugger Architecture divise le débogage en Java en deux fonctionnalités basiques : d'une part l'exécution du programme Java, d'autre part l'analyse du programme exécuté. On peut dès lors considérer deux entités intervenant dans le débogage Java vu par Sun Microsystems : la machine virtuelle et le débogueur.

Aujourd'hui, de nombreuses machines virtuelles, libres ou commerciales, existent et le nombre de débogueurs sur le marché est encore plus important. Tous possèdent leurs spécificités, offrant un large choix d'outils répondant à chaque besoin en matière d'exécution ou investigation de code Java. Afin de tirer partie de cette diversité, la JPDA met en place des standards permettant une interopérabilité des outils respectant ses normes.

La JPDA est composée de deux interfaces de programmation (ou API), l'une destinée à la machine virtuelle, la Java Virtual Machine Debug Interface (JVMDI), l'autre destinée au débogueur, la Java Debug interface (JDI). Elle est également dotée d'un protocole de communication, le Java Debug Wire Protocol (JDWP), décrivant le format des échanges entre les deux entités. En plus de posséder les interfaces citées

précédemment, chacune des entités doit être dotée d'un mécanisme de transcription des informations en messages respectant le protocole. Ces deux entités sont présentées ici comme des bibliothèques (ensemble de fonctions). Finalement, on obtient un système tel que présenté sur la figure 3.1.

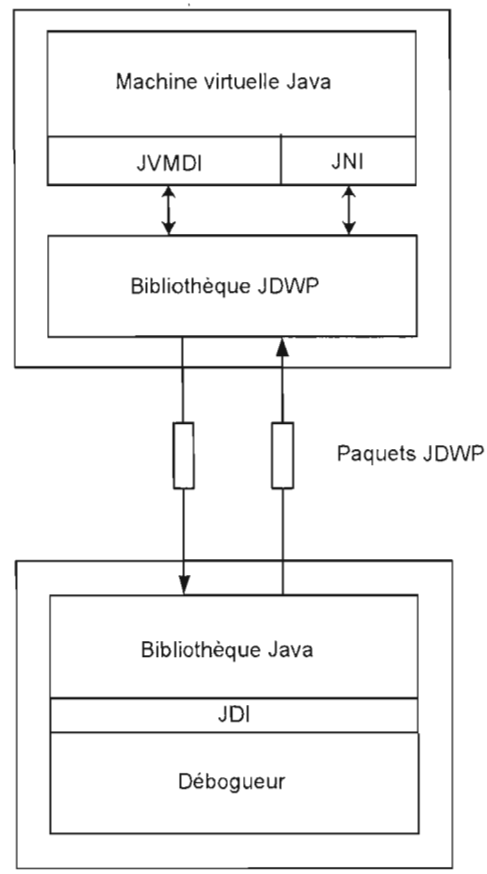


Fig. 3.1 La Java Platform Debugger Architecture

Cette architecture multi-niveaux permet aux développeurs de débogueurs d'opter pour la connexion répondant le mieux à leur besoin. Dans le cas de la figure 3.1, le débogueur est écrit en Java, utilise les bibliothèques Java pour la traduction de commandes en paquets JDWP et se connecte au plus haut niveau de l'architecture.

Il serait possible de connecter un débogueur écrit dans un langage autre que Java,

réceptionnant et décryptant les paquets JDWP en provenance de la machine virtuelle, et traduisant les requêtes de l'utilisateur en paquets du même protocole, comme sur l'exemple proposé sur la figure 3.2.

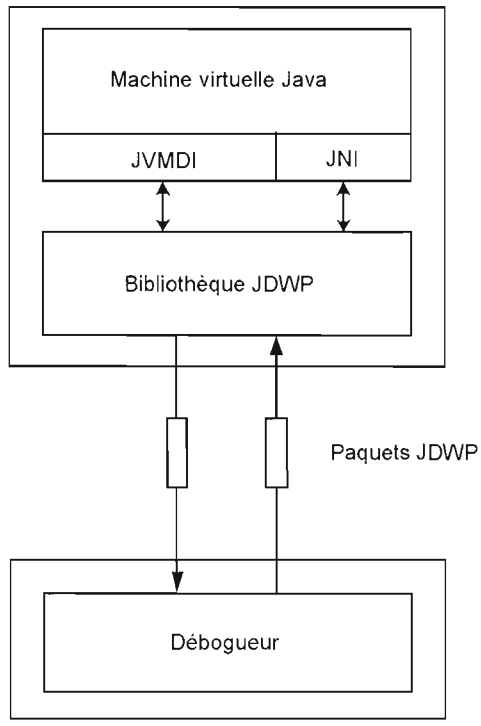


Fig. 3.2 Débogueur n'utilisant pas la bibliothèque Java

L'exemple de la figure 3.3 présente un débogueur écrit en C et connecté directement à la machine virtuelle en tant que client JVMDI. Ce débogueur peut alors collaborer avec les systèmes internes de la machine virtuelle, entre autres avec les mécanismes de gestion de la mémoire, et profiter des fonctionnalités offertes par la Java Native Interface (JNI).

En plus de spécifier les fonctions présentes dans les interfaces, la JPDA met également en place un système d'événements. Ceux-ci sont destinés à remplacer les signaux des systèmes d'exploitation dans l'architecture Java. Ils sont générés par la machine virtuelle et transmis, via la JVMDI, aux entités adjacentes de l'architecture alors informées

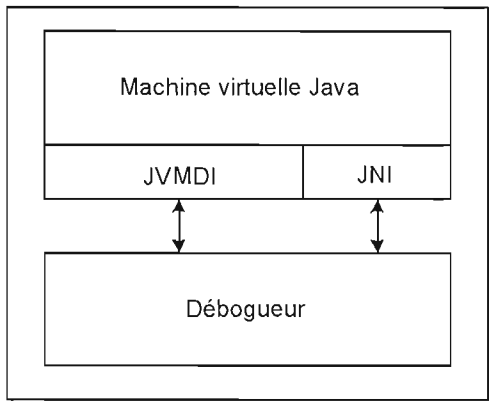


Fig. 3.3 Débogueur connecté comme client JVMDI

de l'état de l'exécution.

Intéressons-nous maintenant à chacune des parties composant cette architecture.

3.1.2 La Machine Virtuelle Java

A l'extrémité de l'architecture proposée par Sun Microsystems, la machine virtuelle prend en charge l'exécution de l'application Java. Du point de vue de l'application Java exécutée, elle agit en lieu et place du système d'exploitation, l'application interagissant alors avec son environnement via les services offerts par la machine virtuelle. Elle est dotée de plusieurs interfaces standard, permettant aux éléments extérieurs d'accéder à ses mécanismes internes. Nous ne traiterons dans ce document que de celles relatives au débogage, ou utilisée lors de l'implantation des fonctions de débogage.

Les machines virtuelles libres pullulent aujourd'hui, chacune possédant ses caractéristiques particulières : langage de programmation utilisé, algorithmes internes, etc. Nos travaux utilisent SableVM, une machine virtuelle Java née des travaux d'Étienne M. Gagnon (Gagnon, 2002). SableVM est développée en langage C, respectant les normes établies en matière de comportement de machines virtuelle Java (Lindholm et Yellin, 1999), mais ne possède pas d'attributs de débogage.

3.1.3 La Java Virtual Machine Debug Interface

La Java Virtual Machine Debug Interface (JVMDI) est une interface de programmation (API), composée de fonctions en langage C. Ces fonctions permettent d'interagir avec certains sous-systèmes de la machine virtuelle (gestion des processus, de l'espace mémoire...)et d'inspecter les états et contrôler l'exécution d'une application Java.

La JVMDI spécifie les comportements des fonctions fournies, de manière indépendante aux choix de développement faits au sein de la machine virtuelle. Si aucune indication n'est donnée quant à l'implantation de cette norme au sein de la machine virtuelle, ces fonctions doivent permettre d'obtenir l'information désirée sans interférer avec l'application, et donc sans modifier le comportement normal de celle-ci.

Cette interface n'est pas indispensable au bon fonctionnement d'une machine virtuelle. En effet, ses principales fonctions permettent de récupérer ou modifier ponctuellement des informations au cœur de la machine virtuelle, et ne modifient en rien l'exécution normale d'une application Java. L'ajout de cette interface peut s'avérer fastidieux au sein d'une machine virtuelle n'ayant pas été conçue avec un souci d'accès simplifié aux informations et de séparation des fonctionnalités. L'addition de certaines fonctions peut également amener à repenser certains algorithmes internes de la machine virtuelle. Aussi, cette interface n'est disponible qu'au sein de peu de machines virtuelles. Aujourd'hui, suite à nos travaux, SableVM est la seule machine virtuelle libre (c'est-à-dire non commerciale) proposant une interface standard de débogage.

On appelle client JVMDI tout outil utilisant l'interface JVMDI afin d'interagir avec la machine virtuelle.

3.1.4 La Java Native Interface

La Java Native Interface (JNI) est une interface de programmation également en langage C. Elle permet aux applications Java d'interagir avec des applications et bibliothèques développées en C, C++, Assembleur (Liang, 1999)...

Cette interface n'est pas spécifique au débogage et n'apparaît donc pas dans les spécifications de la JPDA. Nous ne la détaillerons donc pas dans la suite de cette étude. Cependant, nous avons fait appel, de manière sporadique, à des fonctions de cette interface durant le développement de certaines fonctionnalités relatives au débogage. Il nous est donc apparu pertinent de la mentionner brièvement ici.

3.1.5 La bibliothèque JDWP

La bibliothèque JDWP est l'entité en charge de l'encodage et du décodage des commandes envoyées et reçues par la machine virtuelle. Ces commandes sont traduites en paquets respectant le format décrit dans le Java Debug wire Protocol, d'où le nom que nous avons donné à cette bibliothèque.

La fonction de cette bibliothèque ne s'arrête pas à l'encodage. Elle contient également de nombreuses fonctions de gestion des objets et filtrage des événements générés par la machine virtuelle et transmis au débogueur.

Cette bibliothèque est un client JVMDI écrit en langage C, interagissant également avec la machine virtuelle via la JNI.

3.1.6 Le Java Debug Wire Protocol

Le Java Debug Wire Protocol (JDWP) est le protocole de communication qui définit le format d'information transitant entre la machine virtuelle et le débogueur. Le JDWP ne définit pas le mode de transport utilisé (*socket*, mémoire partagée...) pour la transmission des paquets.

Selon le JDWP, chaque requête transite dans un paquet : entête et corps du paquet doivent respecter des normes définissant l'ordre et la longueur des différents champs. On garantit ainsi un langage commun entre toute machine virtuelle et tout outil de débogage. Les informations contenues dans les paquets sont également soumises à des règles strictes. Chaque type de données (booléen, chaîne de caractères, entier...) doit

respecter scrupuleusement un codage défini par ce protocole.

En plus du format des paquets, le JDWP définit également l'initiation de la communication entre les outils, indépendamment du mode de transport utilisé.

L'utilisation de ce protocole est indépendante des choix des langages de programmation utilisés pour le développement de la machine virtuelle et du débogueur.

3.1.7 La Java Debug Interface et la bibliothèque Java

La Java Debug Interface est une interface de programmation composée de fonctions en Java uniquement. Elle permet la communication entre le code Java d'un débogueur et les bibliothèques standard Java dédiées au débogage.

La JDI a été mise en place afin de faciliter la création de débogueurs en Java, et d'offrir un niveau d'abstraction supplémentaire. Elle autorise ainsi les développeurs de débogueurs à ne se soucier que des fonctionnalités relatives au débogage offertes par leur outil, sans préoccupation de l'encodage des informations au format décrit par le JDWP.

La bibliothèque Java liée à l'interface JDI contient des fonctionnalités permettant de transcrire les commandes reçues et envoyées par le débogueur en paquets respectant les spécifications JDWP.

Des réalisations de ces interfaces et bibliothèques relatives au débogage sont déjà accessibles librement. Les débogueurs utilisés dans le cadre de cette étude utilisant des kits de développement déjà dotés des mécanismes nécessaires à l'encodage, nous ne les détaillerons donc pas dans la suite de ce document.

3.1.8 Le débogueur

Le débogueur est l'outil permettant à l'utilisateur de rechercher les erreurs dans son programme. A l'autre extrémité de l'architecture de débogage, il répond généra-

lement à un besoin précis en matière de développement d'application, et présente des fonctionnalités relatives à un paradigme de débogage (ces fonctionnalités diffèrent selon le paradigme). Si certains systèmes possèdent des capacités de débogage intégrées, l'architecture étudiée ici sous-entend une séparation claire entre exécution et débogage : débogueur et machine virtuelle sont alors développés de manière indépendante.

Il existe aujourd'hui une multitude de débogueurs, couramment intégrés à des environnements de développement complets. Ces débogueurs sont souvent spécifiques à un langage de programmation. Nous précisons, dans les sections appropriées, les spécificités des débogueurs utilisés au cours de notre étude.

3.2 La Java Virtual Machine Debug Interface

3.2.1 Fonctionnalités offertes

Après avoir introduit la position de cette interface au sein de l'architecture, intéressons nous de plus près à l'ensemble de fonctions disponibles. Nous ne listons pas toutes les fonctions accessibles par un client JVMDI. Nous présentons simplement un aperçu général des catégories de fonctions.

La JPDA dote la machine virtuelle Java d'un mécanisme de génération d'événements. Certaines fonctions de la JVMDI permettent de définir les actions générant des événements. Un client JVMDI peut interagir avec une application suite à la génération d'événements, en stipulant, via l'interface, quelle fonction est en charge du traitement des événements. Il lui est également possible de spécifier quels événements lui faire parvenir ou non.

Lors du débogage, la suspension et la reprise de l'exécution d'une application sont choses courantes. Aussi, cette interface propose des fonctions permettant d'avoir un contrôle fin sur l'exécution des processus légers (ou *threads*). Ces fonctions permettent également d'obtenir de l'information (verrous acquis, état...) sur un processus léger. Les processus légers pouvant être organisés en groupes, il est également possible d'obtenir

de l'information sur les groupes existants.

Également indispensables lors du débogage, des fonctions permettent la gestion des points d'arrêt (*breakpoints* et *catchpoints*), ainsi que la récupération et la modification de la valeur d'une variable. On ne pourrait envisager une session de débogage sans ces fonctionnalités.

Dans la programmation orientée objet, on a introduit les concepts de classes, contenant, entre autre, champs et méthodes. Un éventail de fonctions permet l'accès aux informations qui leur sont reliées. Pour les classes, il est possible d'obtenir des informations sur sa nature (c'est-à-dire si c'est un tableau ou une interface) ainsi que son contenu : champs et méthodes présents au sein de la classe. Pour les méthodes, on peut obtenir les informations sur les lignes de code, afin notamment de pouvoir déterminer l'emplacement d'un point d'arrêt. On peut également avoir une liste des variables visibles dans le corps de la méthode, ou celle des exceptions générées par cette méthode. Pour les champs, méthodes et classes, on peut obtenir les noms et signatures, ainsi que les droits d'accès de l'entité (en Java, public, protected, private...).

En Java, l'exécution est décomposée en fenêtres (ou frames), chaque fenêtre correspondant à l'appel à une fonction. On ne peut envisager un contrôle de l'exécution sans accès à ces fenêtres, ne serait-ce que pour l'impression de la pile d'appels. Certaines fonctions de la JVMDI autorisent ce type d'accès.

Si, en Java, l'utilisateur n'a pas d'accès direct à la mémoire, la JVMDI permet de préciser à la machine virtuelle quels mécanismes utiliser pour l'allocation et la libération d'espace mémoire. Le client JVMDI ayant spécifier les fonctions de son choix, il lui est possible d'exercer un contrôle fin sur la gestion de la mémoire par la machine virtuelle dans le cadre du débogage.

Enfin, les normes et standards évoluent rapidement en Java, n'assurant pas toujours une compatibilité avec les outils aux normes précédentes. Aussi, des fonctions permettent d'interroger la machine virtuelle sur les normes respectées et sur les possi-

bilités offertes. Certaines fonctions de la JVMDI étant définies comme optionnelles, les possibilités d'une machine virtuelle représentent l'ensemble des fonctions optionnelles effectivement disponibles.

3.2.2 Communications avec les éléments de l'architecture

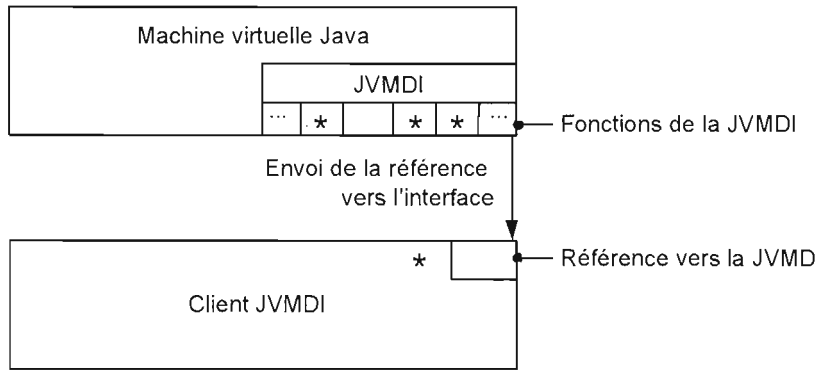
Un client JVMDI communique avec la machine virtuelle via l'interface JVMDI. Ce client peut être soit le débogueur directement, s'il est écrit en langage C et qu'il appartient au même processus que la machine virtuelle, soit une entité indépendante (c'est le cas de notre bibliothèque JDWP). Du point de vue de la machine virtuelle, aucune différence n'existe entre les types de clients.

Les fonctions de la JVMDI sont accessibles via une table de fonctions transmises, sur demande, par la machine virtuelle Java. Le client JVMDI (bibliothèque JDWP dans notre cas) appelle ensuite les fonctions dont les références lui ont été passées via cette table (figure 3.4).

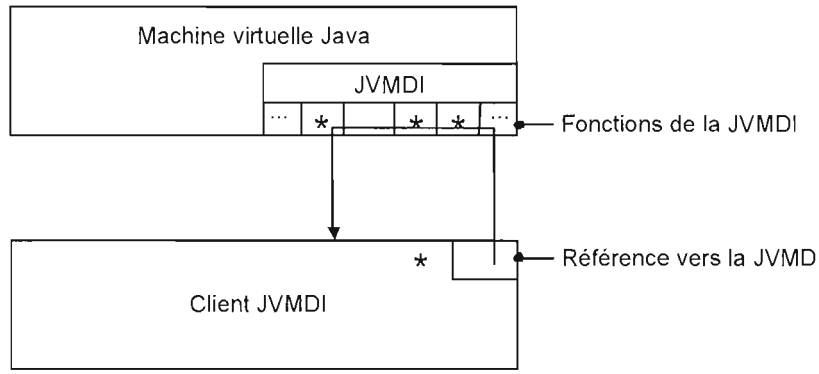
Les fonctions appartenant à cette interface ont toutes le même format : les arguments et valeurs retournées sont passés en paramètres (en utilisant des pointeurs, si besoin), la fonction ne renvoyant qu'un code d'erreur. Si ces fonctions manipulent des objets, on utilise alors des références natives (Liang, 1999).

Toutes les fonctions de cette interface sont implantées dans la machine virtuelle, à l'exception de quelques unes : celles responsables de la gestion de la mémoire et celle responsable de la gestion des événements. Le client JVMDI doit spécifier quelles fonctions utiliser en ajoutant la référence au sein de la JVMDI (figure 3.5). Ces fonctions seront ensuite appelées par la machine virtuelle durant l'exécution de l'application.

Ces mécanismes de communication utilisant le passage de références, cela justifie le fait qu'un client JVMDI doivent impérativement être exécuté au sein du même processus que la machine virtuelle.



(a) Initialisation : envoi de l'ensemble des références vers les fonctions de l'interface



(b) Appel à une fonction de l'interface JVMDI

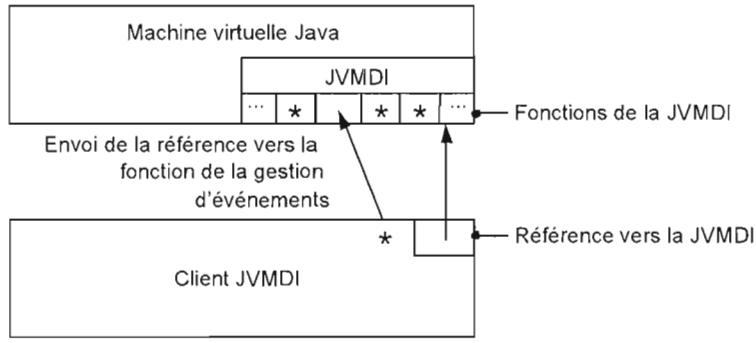
Fig. 3.4 Communication entre la machine virtuelle et le client JVMDI

3.3 Le Java Debug Wire Protocol

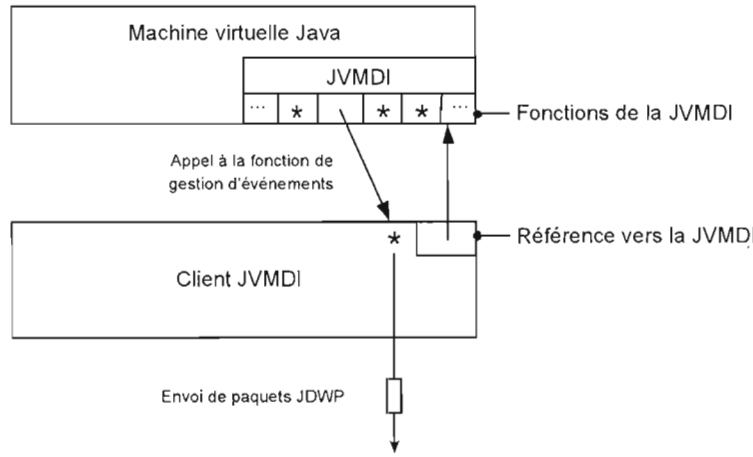
3.3.1 Fonctionnalités offertes

Le protocole de communication sert de jonction entre les deux interfaces à l'extrémité de l'architecture de débogage. Les commandes sont regroupées en dix-huit groupes dans la version utilisée lors de notre étude. Ici également, nous ne listons pas toutes les possibilités offertes par les commandes, simplement les fonctionnalités principales des groupes existants.

Un groupe permet d'obtenir des informations d'ordre général sur les capacités de



(a) Initialisation : ajout de la fonction de gestion d'événements



(b) Appel à la fonction de gestion d'événements

Fig. 3.5 Communication lors de la génération d'un événement

la machine virtuelle. On peut renseigner le débogueur sur l'état de la machine virtuelle : il est possible d'obtenir les versions des normes respectées (Java, JDWP...), ainsi que les choix faits lors du développement de ces normes (format des identifiants des objets...). De plus, on peut renvoyer les listes de toutes les classes chargées par la machine virtuelle, tous les processus légers créés, etc. Ce groupe de commandes permet également la suspension, la reprise et l'arrêt de l'exécution de la machine virtuelle ou la gestion de l'envoi d'événements par celle-ci.

Le groupe des références concerne les informations relatives aux types de référence

des objets utilisés. Par exemple, chaque chaîne de caractères est une instance de la classe Java.lang.String, qui est alors le type de référence de chacune de ces chaînes. Les commandes fournies dans ce groupe permettent l'obtention de la signature de ce type (selon les spécifications de la Java Native Interface), les droits d'accès, les noms et signatures des champs et méthodes que contient ce type, ainsi que la valeur des champs existants.

Concernant les objets, des commandes permettent d'autoriser ou non la collecte de ceux-ci par le ramasse-miettes. Il est également possible d'accéder aux champs de cet objet, afin d'en lire ou modifier les valeurs. On peut de plus invoquer une méthode de cet objet (par réflexion) ou obtenir l'identifiant de son type de reference.

Le groupe des classes permet principalement d'invoquer les méthodes, de créer une nouvelle instance ou de modifier les champs statiques d'une classe. Il est également possible de savoir de quelle classe dérive la classe proposée.

Un groupe permet de recueillir des informations concernant une méthode. Ainsi, on peut obtenir les tables de correspondance entre les lignes de code Java et les instructions en code octet, ainsi que le positionnement des variables d'une méthode. On peut ainsi obtenir la valeur d'une variable parmi le groupe de variables locales de cette méthode. Il est également possible d'obtenir le code octet d'une méthode ou de savoir si cette méthode est obsolète.

L'exécution des processus légers peut être contrôlée grâce à un groupe de commandes en permettant la suspension et la reprise. Il est possible de connaître le nom, le status ou le groupe d'un processus léger. Chaque processus léger possède une pile de fenêtres d'exécution, chacune correspondant à l'appel d'une méthode. Des commandes permettent d'en déterminer le nombre et la séquence.

Ces fenêtres d'exécution peuvent être interrogées via un autre groupe de commandes. On peut déterminer l'identifiant de l'objet ayant fait l'appel à la méthode liée à la fenêtre, ou accéder à la valeur d'une variable de cette fenêtre.

D'autres groupes permettent la création d'une nouvelle instance d'une chaîne de caractères ou d'un tableau, ou de lister les classes visibles par un chargeur de classes (ou *class loader*) précis.

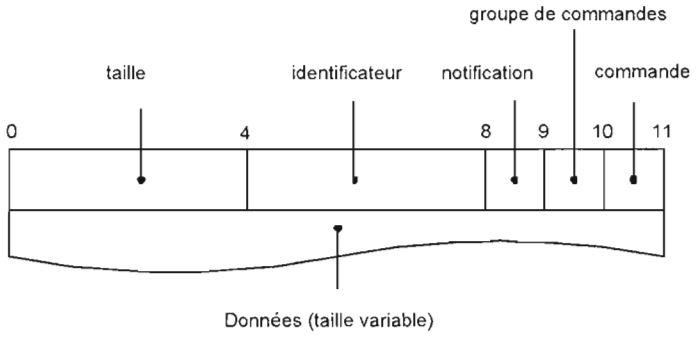
Enfin, il existe deux ensembles de commandes liées à la gestion des événements. Le premier permet de définir quels événements doivent, ou non, être transmis. Le second permet justement de transcrire ces événements.

3.3.2 Format des paquets JDWP

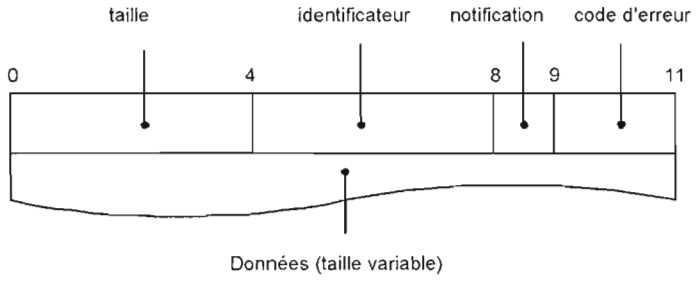
Le protocole tient une position centrale dans l'architecture Java. Il est utilisé d'une part par la machine virtuelle, d'autre part par le débogueur. Le débogueur n'est pas forcément exécuté sur le même processus que l'application, voire sur la même machine physique. Le mécanisme proposé doit donc permettre d'envoyer des instructions à distance. C'est ce que propose JDWP, en utilisant des paquets pour acheminer l'information.

Ce protocole spécifie le format des paquets transportant les commandes, destinés à voyager sur le réseau ou au sein de la même machine. Le format proposé reste indépendant de la méthode de transport, et la transmission ne commence qu'une fois la connexion entre les machines hôtes établie. La communication commence par un échange de paquets de synchronisation de début, puis les deux entités communiquent librement.

Chaque paquet contient deux parties : entête et corps. Deux types de paquets existent : les commandes et les réponses (figure 3.6). Les entêtes des commandes et réponses sont grandement similaires : toutes deux contiennent la longueur totale du paquet envoyé, l'identifiant de la requête, ainsi qu'un champ de notification permettant de différencier une commande d'une réponse. Dans le cas d'une commande, l'entête contient également le numéro du groupe et de la commande à exécuter. Le corps des paquets contient les paramètres des commandes, si besoin. Dans le cas d'une réponse, l'entête contient les codes d'erreur retournés par l'exécution de la commande correspondante.



(a) Paquet de commande



(b) paquet de réponse

Fig. 3.6 Format des paquets JDWP

Les paquets contenant des notifications d'événements sont considérés comme des commandes, et le corps de ces paquets contient l'information relative à l'événement généré.

3.3.3 Propriétés du protocole

La propriété la plus remarquable de ce protocole est les fait qu'il soit asynchrone : les réponses peuvent ne pas être envoyées dans le même ordre que l'exécution ou la réception des paquets de commande. C'est pourquoi les paquets contenant les réponses comprennent l'identifiant de la requête à laquelle ils font écho.

Certaines commandes ne requièrent cependant pas de réponse, et la spécification reste muette sur l'attitude à adopter dans ce cas. Cependant, notre expérience nous a

démontré que les outils utilisés attendaient un paquet en retour de ce type de commandes. Ainsi, dans notre réalisation, les commandes ne demandant pas de notification particulière se voient renvoyer une réponse contenant les identifiants de la commande associée, et un corps vide. Ces réponses jouent, en quelque sorte, le rôle d'accusés de réception.

Bien que ce protocole soit asynchrone, lorsqu'un événement devant être transmis au débogueur est généré par la machine virtuelle, le paquet associé doit être envoyé directement, avant tout autre paquet, sans processus de stockage ou de file d'attente.

Une autre spécificité de ce protocole est qu'il ne permet pas de communiquer avec une machine virtuelle totalement inconnue, certaines informations devant être recueillies avant le démarrage de cette dernière. Il est nécessaire de connaître les possibilités offertes par celle-ci en matière de transport des paquets, afin d'utiliser une méthode supportée par la machine utilisée. Le choix du mode de transport des paquets JDWP est indiqué via une option de la ligne de commande permettant de lancer la machine virtuelle (figure 3.7). Ce choix est définitif et ne peut être modifié en cours d'utilisation, même partiellement (par exemple simplement le numéro du port d'écoute). Les différentes options, ainsi que les valeurs possibles, sont indiquées dans la spécification.

$-Xrunjdwp : transport = dt\_socket, server = y, address = localhost : 12345$

-Xrunjdwp indique que les options suivantes sont destinées à la communication avec le débogueur.

transport indique le mode de transport utilisé, ici une *socket.*

server permet de savoir qui, de la machine virtuelle ou du débogueur, décide du port d'écoute. La valeur y indique que le choix est laissé au débogueur.

address indique l'adresse de la machine exécutant le débogueur, suivie du port d'écoute.

Fig. 3.7 Exemple d'options d'une ligne de commande de lancement de la machine virtuelle Java

3.4 La Bibliothèque JDWP

La machine virtuelle Java propose essentiellement des fonctions de contrôle et d'exécution des applications Java. Ces fonctions sont étendues afin de répondre aux besoins de communication avec les outils de débogage. Le débogueur, lui, représente à l'utilisateur les états du système et lui permet d'interagir avec l'application. Entre ces entités, reposent deux bibliothèques permettant de traduire les commandes et réponses en paquets JDWP : la bibliothèque Java pour le débogueur, la bibliothèque JDWP pour la machine virtuelle.

La JVMDI propose des fonctions de bas niveau, permettant, par exemple, de modifier la valeur d'une variable à une position précise dans le groupe de variables visibles au sein d'une méthode. Le débogueur permet, entre autre, la modification d'une variable désignée par le nom qui lui a été attribué dans le code source. Lorsque l'utilisateur désire modifier la valeur d'une variable, une requête contenant l'emplacement (classe et méthode contenant cette variable), le nom et la nouvelle valeur de cette variable parvient à la bibliothèque JDWP. C'est cette dernière qui devra mettre en œuvre les fonctions à sa disposition afin de faire correspondre le nom de la variable et sa position dans le groupe. Puis elle appellera la fonction de modification avec les paramètres adéquats.

Cette exemple illustre le fait que la bibliothèque JDWP n'est pas simplement utilisée pour la traduction de paquets, mais possède également une certaine intelligence. Voyons maintenant, dans les sections suivantes, l'ensemble des tâches déléguées à cette bibliothèque.

3.4.1 Encodage et décodage des paquets JDWP

La bibliothèque JDWP reçoit les paquets en provenance du débogueur. Avant de pouvoir exécuter les fonctions demandées, la bibliothèque JDWP est dotée d'un mécanisme de décodage des paquets, afin de récupérer les valeurs reçues dans ce paquet.

De la même manière, lors d'appels aux fonctions des interfaces de la machine

virtuelle, des valeurs sont retournées et doivent être transmises au débogueur. Cette bibliothèque permet alors la construction de paquets au format décrit par le protocole JDWP, contenant les valeurs retournées par les fonctions appelées.

Lors de l'exécution d'une application, lorsqu'un événement doit être transmis au débogueur, la bibliothèque JDWP est responsable d'une part de la réception de l'événement, d'autre part du remplissage des champs du paquet avec les valeurs correspondant à l'événement généré. Une fois les paquets convenablement construits, la bibliothèque JDWP les transmet au débogueur, en utilisant le moyen de transport spécifié lors du démarrage de la session de débogage.

Les possibilités en matière de transport de l'information sont propres à chaque bibliothèque et ne dépendent pas des spécifications de la JPDA. Celle-ci n'offre en effet qu'une méthode standard de communiquer quel moyen de transport est utilisé.

3.4.2 Attribution d'identifiants

Les objets ne pouvant pas transiter le réseau, le protocole JDWP utilise des identifiants, propres à chaque objet (le terme objet est utilisé ici au sens large, incluant la définition propre à la programmation orientée objet). La bibliothèque JDWP est idéalement placée pour effectuer l'attribution et la gestion des identifiants : elle peut obtenir et manipuler les objets via les interfaces de la machine virtuelle, puis transmettre leurs identifiants sur le réseau au débogueur.

Elle attribue un identifiant à toute entité pouvant être manipulée à distance, telle une méthode, une fenêtre d'exécution, un objet au sens Java, etc. Cet identifiant est attribué la première fois que la bibliothèque JDWP rencontre un objet, et n'est jamais modifié durant toute la durée de vie de cet objet. Il est ensuite communiqué au débogueur, qui fera, dans les requêtes postérieures, référence à cet objet grâce à cet identifiant.

Certains identifiants sont globaux et identifient uniquement une entité au sein de

la machine virtuelle. D'autres, tels les identifiants des champs d'une classe (ou field), sont locaux et les identifient au sein d'une classe. Cette propriétés est définie pour chaque type d'identifiant par la spécification de la JPDA.

3.4.3 Gestion des objets

La bibliothèque JDWP attribue à chaque objet un identifiant lors de la première utilisation de celui-ci. Cependant, afin de déterminer si un objet n'a pas été utilisé précédemment, cette bibliothèque doit conserver une trace des objets connus. Les structures de données et algorithmes utilisés pour la gestion et l'accès aux objets connus sont propres à chaque réalisation et ne font l'objet d'aucune spécification. Les objets étant utilisés dans la quasi totalité des requêtes provenant du débogueur, on comprendra l'intérêt d'opter pour des structures de données et algorithmes relativement efficaces.

Ainsi, la bibliothèque JDWP est responsable de la gestion des entités pouvant être manipulées par le débogueur.

3.4.4 Gestion de la mémoire

Les références vers les objets connus sont conservées au sein de la bibliothèque JDWP. Afin de pouvoir transmettre ces références, la machine virtuelle alloue la mémoire destinée à conserver cet objet, et le préserver du ramasse-miettes Java (garbage collection). Ce mécanisme d'allocation doit être spécifié par la bibliothèque JDWP : celle-ci doit envoyer à la machine virtuelle les références vers les fonctions d'allocation et libération de mémoire utilisées par les fonctions de la JVMDI. Elle peut ainsi exercer un contrôle fin sur l'utilisation de la mémoire par la machine virtuelle.

3.4.5 Gestion des événements

Le nombre d'événements générés est de l'ordre du nombre de codes octets qui composent l'application Java à déboguer. On comprend donc la nécessité de mettre en place des mécanismes de filtrage des événements. Il est possible d'activer la génération

d'événements globalement ou par processus, au sein de la machine virtuelle. Cette possibilité représente un premier moyen de filtrer les événements à transmettre aux autres éléments de l'architecture.

Les événements reçus par la bibliothèque JDWP sont traités, puis filtrés afin de définir s'ils seront transmis ou non au débogueur. Ce second filtre, au sein de la bibliothèque JDWP, se doit d'être performant, puisque les événements qui le passeront seront transformés en paquets JDWP et transiteront sur le réseau. Ce second filtre évite une surcharge du réseau due à la transmission d'événements non désirés, le débit des communication n'étant pas géré par la JPDA.

Le protocole JDWP permet au débogueur de spécifier à la bibliothèque JDWP des tests que les événements générés doivent réussir avant de pouvoir lui être transmis. Par exemple, il est possible de disposer plusieurs points d'arrêt dans le code d'une même méthode. Cependant, l'utilisateur peut activer ou désactiver certains points d'arrêt sans pour autant les supprimer. L'exécution ne s'arrêtera donc pas sur les points désactivés. Pour ce faire, le débogueur spécifie à la machine virtuelles les emplacements devant générer une suspension de l'exécution. Les points d'arrêt désactivés généreront des événements qui seront transmis par la machine virtuelle au client JVMDI (le premier filtre est passé), mais les emplacements ne correspondant pas au tests mis en place, ils ne seront pas transmis au débogueur (arrêtés par le second filtre).

Le traitement d'un événement peut, par exemple, consister à invalider tous les identifiants correspondant aux fenêtres d'exécution d'un processus, lorsque l'événement de fin d'exécution de ce processus est reçu. Si aucune requête du débogueur ne demande une notification lors de l'arrêt d'un processus, cet événement ne sera pas transmis après le traitement, bien qu'envoyé de la machine virtuelle à la bibliothèque JDWP.

La bibliothèque JDWP est donc capable de traiter, filtrer et transmettre les événements.

3.4.6 Gestion des processus

Lors du débogage d'une application Java, il est utile de disposer de mécanismes de suspension et reprise de l'exécution afin d'inspecter l'état d'un programme à un instant donné. L'interface JVMDI propose des fonctions permettant de contrôler l'exécution d'un processus. Il est donc possible à la bibliothèque JDWP d'y faire appel.

Lorsque le débogueur spécifie des filtres additionnels à appliquer lors de la génération d'événements, il peut également indiquer la politique de suspension voulue : suspension de tous les processus de la machine virtuelle, uniquement du processus ayant généré l'événement ou d'aucun processus. C'est alors à la bibliothèque JDWP, en charge du traitement des événements, de procéder à la suspension des processus indiqués. Elle utilise, pour ce faire, les fonctions disponibles au sein de la JVMDI. Puis, sur demande du débogueur, cette même bibliothèque demande à la machine virtuelle la reprise de ces processus.

En plus de la gestion des processus de l'application à déboguer, la bibliothèque maintient constamment un canal de communication avec le débogueur. Ce canal prend la forme d'un processus indépendant de l'application. Ce processus est invisible au débogueur et est utilisé pour transmettre et recevoir des paquets JDWP aux autres entités. Il est créé, géré et détruit par la bibliothèque JDWP.

3.5 Évaluation de l'architecture

La JPDA est une architecture proposée par Sun Microsystems. Il nous aurait été possible d'opter pour une autre architecture, voire d'en concevoir une. Cette section décrit les avantages et inconvénients de la JPDA, avant de conclure sur les raisons de notre choix de la JPDA.

3.5.1 Points forts de la JPDA

3.5.1.1 Modularité

Le principal avantage de cette architecture réside, selon nous, dans la modularité offerte. En effet, si aucune dépendance n'est introduite lors du développement d'un module, l'utilisateur a alors la possibilité d'utiliser les réalisations de son choix à chacun des niveaux de cette architecture. Ainsi, si le choix des structures de données utilisées, ou si les algorithmes proposés pour la gestion des objets n'étaient pas optimaux pour une utilisation particulière, il serait possible de remplacer la bibliothèque JDWP par une réalisation du choix de l'utilisateur, sans avoir à modifier les autres éléments de l'architecture (machine virtuelle ou débogueur). Cette modularité a été largement utilisée lors des procédures de tests et de validation de nos travaux, décrits dans la suite de ce document.

3.5.1.2 Séparation des fonctionnalités

La vocation d'une machine virtuelle est l'exécution du code Java. Il serait possible d'implanter certaines des fonctionnalités de débogage au cœur même de la machine virtuelle, qui deviendrait alors un unique outil en charge de l'exécution et du développement des applications. Mais plusieurs inconvénients entravent cette optique. Les paradigmes de débogage existants ne sont pas tous appropriés à toutes les applications développées. Offrir le support de l'ensemble de ces concepts au sein d'un même outil ne serait donc pas pertinent dans tous les cas. De plus, les machines virtuelles Java seraient alors plus gourmandes en ressources , essentiellement en espace mémoire. Les systèmes sur lesquels seule l'exécution est requise et ne disposant que de ressources limitées n'opteraient donc pas pour une architecture Java. C'est le cas, par exemple, pour les appareils électroniques portables, tels les téléphones cellulaires.

L'architecture suggérée sépare clairement les fonctionnalités, les répartissant aux différentes entités. La filtration des événements déléguée à la bibliothèque JDWP réduit

l'échange de messages sur le réseau, et évite à la machine virtuelle d'effectuer cette gestion, hors de ses attributions. L'implantation de cette architecture ne demande donc pas de modification majeure des mécanismes internes de la machine virtuelle.

3.5.1.3 Compatibilité

Sun Microsystems rédige et publie la plupart des spécifications reliées à Java, jouissant d'une visibilité internationale. La quasi totalité des outils commerciaux Java respectent ces normes. En optant pour l'architecture proposée ici, nous assurons à SableVM une compatibilité avec la majorité des outils commerciaux de débogage Java, ainsi qu'avec un nombre importants d'outils libres. Implanter cette architecture permet de ne pas restreindre le choix d'outils de débogage utilisables de concert avec SableVM. La communauté d'utilisateurs peut alors tirer partie des puissants outils déjà existants et répondant aux normes.

3.5.2 Lacunes de l'architecture

3.5.2.1 Absence de valeurs des constantes

Si l'architecture nous paraît convenir à l'élaboration d'une interface de débogage pour machine virtuelle Java, certains détails peuvent cependant représenter un frein au choix de la JPDA. Ainsi, les valeurs des constantes utilisées dans les fonctions proposées ne sont pas définies. Le programmeur a alors libre choix des valeurs utilisées. Cette latitude représente cependant une contrainte à la modularité. En effet, cette politique impose soit une publication des valeurs choisies, soit une compilation conjointe des éléments de la partie de cette architecture liée à la machine virtuelle.

3.5.2.2 Failles de sécurité

Certaines des fonctionnalités offertes par les interfaces peuvent présenter des failles de sécurité. Ces failles seront exhibées lorsque les modules les contenant seront présen-

tés plus précisément, dans les chapitres suivants. Elles ne remettent toutefois pas en cause l'architecture, mais soulèvent tout de même certains points à modifier dans les prochaines révisions de cette spécification.

3.5.3 Conclusion

Finalement, bien que présentant certains défauts mineurs à nos yeux, l'architecture proposée par Sun Microsystems nous semble convenir à l'élaboration d'une interface de débogage pour machine virtuelle Java. Nous développons donc, dans la suite de ce document, les différents éléments mis en place afin de permettre à SableVM de communiquer avec les débogueurs.

Chapitre IV

IMPLANTATION DE L'INTERFACE DE DÉBOGAGE

La Java Virtual Machine Debug Interface est un ensemble de fonctions en langage C permettant d'interagir directement avec les mécanismes internes de la machine virtuelle Java. Chacune des fonctions implantées au sein de la machine virtuelle doit respecter la norme décrivant son comportement.

Ce chapitre détaille l'implantation de la JVMDI au sein de SableVM. La première partie présente la structure de l'interface de débogage. Dans la seconde partie, l'implantation du mécanisme de génération d'événements proposée par la JPDA est détaillée. La troisième partie est consacrée à l'implantation du support aux points d'arrêts dans SableVM. La partie suivante traite de l'implantation des fonctions de contrôle des processus. Puis, nous expliquons comment le chargement des classes est effectuée au cœur de SableVM. Enfin, la dernière partie présente les fonctionnalités de débogage pour lesquelles l'implantation a demandé peu ou pas de changements dans la structure de SableVM.

4.1 Structure de l'interface de débogage

La JVMDI est la seule interface permettant à des entités extérieures à la machine virtuelle Java de spécifier des fonctions qui seront appelées par les mécanismes internes de cette dernière. Cette section présente donc la structure de données interne à SableVM destinée à accueillir cette interface.

La JVMDI est, au même titre que la JNI, un tableau de fonctions au sein de la machine virtuelle. Le format de ce tableau est spécifié par la JPDA. Une référence vers ce tableau est envoyée aux entités en faisant la demande, alors connues sous le nom de clients JVMDI.

Les spécifications imposent les structures de données destinées à contenir ces interfaces, afin que les clients JVMDI puissent y accéder de manière standard. Ces structures de données faisant partie des mécanismes internes de la machine virtuelle, il est toutefois possible de les englober dans des structures propres à SableVM, afin d'y entreposer également d'autres éléments, à des fins internes. C'est le cas des fonctions passées par le client JVMDI, fonctions de gestion de la mémoire et de traitement des événements, stockées dans cette structure interne. Nous avons utilisé une structure similaire à celle présentée en figure 4.1 pour contenir l'interface et les attributs de débogage.

L'utilisation d'un verrou protège le canal de communication contre les accès concurrents. En effet, ce canal est utilisé d'un part pour l'envoi et la réception de commandes, d'autre part pour l'envoi de paquets d'événements. Ces deux fonctionnalités étant distinctes et partageant le médium de communication, un verrou est nécessaire.

4.2 Génération d'événements

Afin de pallier l'impossibilité d'utiliser les mécanismes d'interruptions et signaux disponibles au niveau du système d'exploitation, la JPDA met en place un jeu d'événements, générés par la machine virtuelle.

Cette section détaille l'implantation de ce mécanisme au sein de SableVM. Tout d'abord, nous présentons le processus de génération d'événement. La partie suivante décrit le choix de l'emplacement de la génération de chaque type d'événement. Enfin, la dernière partie détaille les modifications apportées à SableVM, afin de permettre le support de ce mécanisme d'événements.

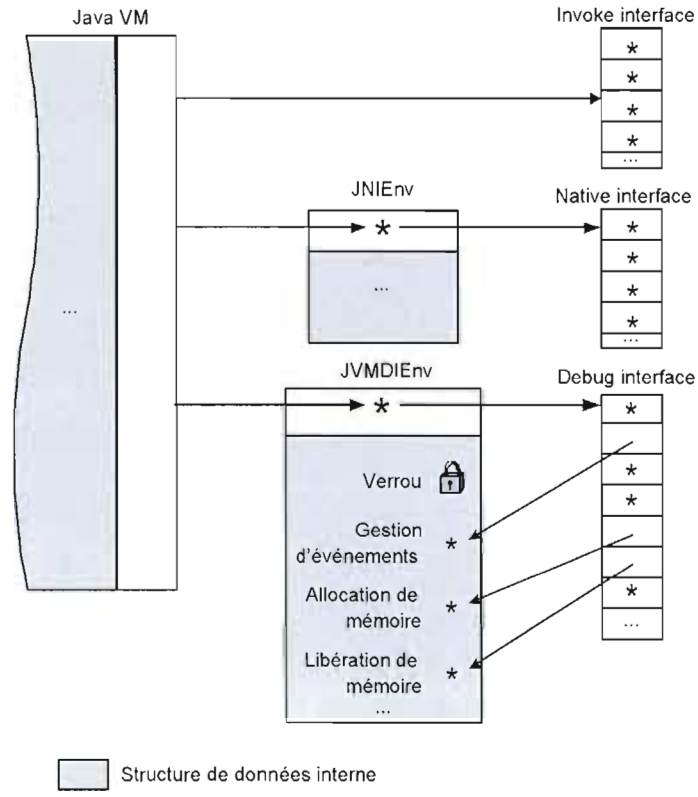


Fig. 4.1 Structure partielle de la JVMDI dans SableVM

4.2.1 Processus de génération d'événements

La machine virtuelle possède une série d'indicateurs permettant de déterminer si chaque type d'événement doit être transmis aux autres entités de la JPDA. Ces indicateurs sont un ensemble de bits (un bit par typed'événement), dont la valeur 1 indique l'autorisation d'envoi d'un événement du type correspondant, et 0 correspond à l'interdiction. Ces indicateurs sont globaux, et donc accessibles par chaque processus. Chaque processus possède également sa série d'indicateurs, permettant d'autoriser localement la génération de certains types d'événements. Lors de la création d'un processus, ses indicateurs prendront comme valeurs initiales les valeurs courantes des indicateurs de la machine virtuelle. La JPDA spécifie les valeurs initiales des indicateurs globaux. Une

fois créées, ces deux séries d'indicateurs sont totalement indépendantes les unes des autres : les modifications apportées aux uns n'altèrent pas les autres. La modification de ces valeurs, globalement ou localement, se fait via une fonction de la JVMDI.

Lors de la création d'un événement, pour procéder à la notification, il faut qu'au moins une des séries d'indicateurs, soit celle du processus l'ayant généré, soit celle de la machine virtuelle, autorise son envoi. On peut ainsi avoir un contrôle plus fin sur la quantité d'événements générés par la machine virtuelle. Par exemple, lors d'une session de débogage, on pourra spécifier qu'un processus nous notifie à chaque exécution d'un code octet. Pour ce faire, on interdira la génération de cet événement de manière globale et on l'autorisera au sein du processus voulu. Cette notification sera la base d'une exécution pas à pas. Il serait impensable d'autoriser la machine virtuelle, et donc l'ensemble des processus, à envoyer ce type d'événements, étant donné le grand nombre d'événements générés et le ralentissement sensible de l'exécution induit.

Afin de faciliter la maintenance de SableVM, la génération d'événements est effectuée dans des blocs de code clairement séparés du reste. Ces blocs de code sont insérés au cœur de SableVM, aux emplacements appropriés à chaque type d'événement, et suivent la même construction à chaque fois : tout d'abord, on s'assure que les indicateurs locaux ou globaux permettent la génération de ce type d'événement. Si c'est le cas, on collecte alors les informations permettant de construire l'événement. Puis on appelle la fonction de gestion des événements, spécifiée par le client JVMDI.

4.2.2 Types d'événements

Les structures de données représentant chaque événement sont spécifiées par la JPDA, facilitant, une fois de plus, la modularité de l'architecture. Toutes ces structures contiennent les informations concernant la localisation de l'événement (le processus ayant généré cet événement et, dans certains cas, la classe et la méthode), ainsi que des données spécifiques à chaque événement.

La construction de l'événement est souvent triviale une fois l'emplacement adé-

quat de génération choisi : les informations nécessaires à cette construction sont alors aisément accessibles. Pour les événements ayant demandé une mise en œuvre particulière, nous détaillerons, dans cette partie, le choix de l'emplacement de génération. La dernière section présente brièvement l'implantation des autres événements au sein de SableVM.

4.2.2.1 Événements liés aux exceptions

Le mécanisme permettant à SableVM de lever des exceptions se situe au cœur de l'interpréteur de code octet. Lorsqu'une exception est levée, on peut soit mettre un terme à l'exécution, soit rechercher une clause de traitement de cette exception (clause *catch*) dans les méthodes appelantes. Un événement relatif à cette exception est alors généré après la création de l'objet exception, et juste avant la recherche de la clause de traitement.

Lors de la recherche de la clause de traitement de l'exception, on peut rencontrer plusieurs autres clauses. On vérifie alors que le type de l'exception gérée par ces clauses correspond à celui de l'exception levée. Pour ce faire, on peut être amené à charger et préparer une classe, et donc lever potentiellement une nouvelle exception (si la classe à charger n'est pas trouvée, par exemple). Cette nouvelle exception écrase alors la précédente. Une fois cette novelle exception créée, on peut soit arrêter l'exécution, soit rechercher la clause de traitement de cette nouvelle exception, et ainsi de suite.

L'événement généré suite à une exception doit contenir non seulement les informations relatives à l'exception, mais également à la localisation de la clause de traitement de celle-ci, si elle existe. Étant donnée la génération précoce de cet événement, les données relatives à la clause de traitement ne sont pas encore disponibles. Nous avons donc du dupliquer la recherche de la clause *catch*, en prenant soin de supprimer les actions pouvant générer une nouvelle exception (chargement d'une nouvelle classe, par exemple). Lors de la recherche de cette clause, si nous rencontrons le besoin de charger ou préparer une nouvelle classe, nous abandonnons la recherche, considérant que la clause est

momentanément inatteignable. Les champs de l'événement indiquant l'emplacement de la clause sont alors à 0.

Un second événement relatif aux exceptions existe. Celui-ci est généré lorsque l'exception levée atteint sa clause de traitement. On doit alors indiquer l'emplacement réel dans le code, ainsi que la nature de l'exception traitée. Cependant, lors du traitement d'une exception, cette dernière n'est pas toujours disponible. Le traitement de l'exception peut avoir eu lieu lors d'un appel à du code natif, via la JNI. L'événement correspondant n'est généré qu'au retour à l'exécution du code Java, une fois l'exception traitée. On a donc inséré, au niveau de chaque processus, un indicateur signalant si le traitement d'une exception a été effectué dans le code natif. Cet indicateur est enclenché dans le corps de la fonction JNI permettant le traitement d'une exception, la fonction ExceptionClear. Si l'indicateur est à 1 au retour au code Java, on procède à la génération de l'événement correspondant : on mentionne la position courante dans le code, mais aucune information concernant la nature de l'exception. Si le traitement a lieu en Java, on génère un événement contenant des données relatives à la position et à la nature de l'exception.

4.2.2.2 Événements relatifs à l'exécution du code octet

Comme présenté en section 2.1.4.3, SableVM propose trois types d'interprétation du code octet. Le support au débogage n'est toutefois disponible qu'avec l'interpréteur *switch*. Ce type d'interprétation facilite l'implantation de deux types d'événements : ceux générés suite à la rencontre d'un *breakpoint* et ceux indiquant une exécution pas à pas. Ces deux types d'événements sont insérés dans la boucle d'interprétation du code octet.

Les *breakpoints* sont insérés en modifiant la valeur du code octet correspondant : le premier bit passe de la valeur 0 à 1. La nouvelle valeur du code octet n'équivaut alors à aucun code connu par la machine virtuelle. La génération de l'événement est donc placée dans la clause de gestion des erreurs de la machine virtuelle. On vérifie que le code octet

comporte bien un indicateur de *breakpoint* et, le cas échéant, on génère l'événement. Sinon, on procède normalement au traitement de l'erreur. L'événement contient des informations sur l'emplacement dans le code, information dont l'accès est aisé une fois dans la fonction d'interprétation. La construction de l'objet est alors élémentaire.

Les événements relatifs à l'exécution pas à pas permettent de suivre l'exécution du code à la granularité la plus fine possible au sein de la machine virtuelle. Dans le cas de SableVM, cette granularité correspond à l'exécution d'une instruction code octet (opération et paramètres). Chaque instruction est suivie d'un saut au début de la boucle switch d'interprétation, afin de permettre l'exécution du code suivant. On ne peut donc pas insérer la génération après l'exécution de chaque instruction. On a donc inséré la génération juste avant la boucle switch. Ce choix d'emplacement induit la génération d'un événement juste avant l'exécution du premier code octet, mais ce comportement n'entre pas en conflit avec les spécifications. Celles-ci indiquent simplement qu'un événement doit être généré dès qu'on atteint un nouvel emplacement dans l'exécution.

De la même manière que pour les *breakpoints*, l'objet créé pour l'événement d'exécution pas à pas ne doit contenir que des informations sur son emplacement dans le cours de l'exécution. Sa construction est aisée.

4.2.2.3 Événements liés aux contextes d'exécution

Les événements relatifs au début et à la fin de l'exécution de méthode sont ceux ayant demandé le plus d'insertions de code de générations. En effet, les méthodes peuvent être invoquées depuis de nombreux endroits dans SableVM. Par souci de maintenance de code, plutôt que d'insérer la génération à tous ces emplacements, nous avons opté pour la centralisation de la génération au sein d'une fonction séparée, appelée avant chaque entrée ou sortie de méthode. Nous avons donc inséré des appels à la génération d'événements à la fin de toutes les opérations des codes octet d'invocation de méthodes (invokestatic, invokevirtual, etc), ainsi que dans les fonctions de la JNI de type Call<type>Method, où <type> représente le type retourné par la méthode Java.

Les événements relatifs à la sortie ont été insérés avant la fin de ces méthodes natives d'invocation.

L'appel à la fonction de génération s'effectue avec comme paramètre la structure de données interne représentant l'environnement d'exécution. Cette structure contient le processus courant et permet un accès rapide aux informations nécessaires à la construction de l'événement.

4.2.2.4 Événements de création et arrêt de la machine virtuelle

L'événement de création de machine virtuelle indique aux entités susceptibles d'interagir avec SableVM que celle-ci est dorénavant en fonction. La spécification n'impose pas que cet événement soit le premier à être transmis. Cependant, tout événement transmis précédemment doit être traité avec précaution, les mécanismes internes de la machine virtuelle pouvant ne pas être totalement fonctionnels.

Plutôt que d'indiquer que la machine virtuelle est prête à l'utilisation, nous avons choisi d'attendre que les mécanismes de communication (mise en place des connexions réseau et de la fonction de gestion d'événements) soient disponibles. Avant ça, les événements ne pourraient être transmis aux autres entités, leur génération serait alors inutile.

Le second événement indique la fin d'activité de la machine virtuelle. La JPDA indique que cet événement ne doit jamais transiter sur le réseau. Il doit simplement être transmis au client JVMDI, qui pourra alors effectuer les actions appropriées (libération de l'espace mémoire, coupure des connexions réseau...). SableVM étant développée en C, plusieurs causes peuvent engendrer l'arrêt l'exécution. Par exemple, une fonction défectueuse pourrait causer une corruption de mémoire et mettre fin à l'exécution de la machine virtuelle. C'est pourquoi, l'envoi de cet événement n'est pas assuré pour chaque cause potentielle de fin d'exécution. Nous conseillons au client JVMDI de se baser, à l'instar de l'événement de création de la machine virtuelle, sur l'état des connexions réseau.

4.2.2.5 Autres événements implantés

L'implantation d'autres types d'événements a été effectuée au sein de SableVM. Les événements ne demandant pas de modifications majeures, simplement l'insertion de la génération, sont décrits dans cette section.

Lors de l'appel à la fonction de création d'un processus, l'un des paramètres représente la première fonction que le processus devra exécuter, une fois créé. La fonction de création est responsable de la création et l'initialisation de la structure de données représentant le processus, puis de l'appel de la fonction indiquée. Au retour de celle-ci, le processus a terminé son exécution, on procède alors la destruction ou au recyclage de la structure de données.

Les événements indiquant le début et la fin de l'exécution d'un processus ne contiennent que l'objet représentant le processus créé ou détruit. La génération de l'événement de création de processus a été insérée dès la fin de la création de l'objet, avant l'appel à la fonction passée en paramètre. La génération de l'événement de fin de processus est insérée au retour de la fonction devant être exécutée par le processus. Ce procédé nous assure de deux choses : (1) l'initialisation de la structure de données représentant le processus est achevée avant de générer l'événement de début de processus, et (2) cette structure n'est pas encore détruite ou recyclée lors de la génération de l'événement de fin d'exécution.

Un événement doit également être généré lors de la préparation d'une classe. La fonction de préparation de classe étant clairement identifiée et séparée des autres modules de SableVM, l'insertion de l'événement correspondant a été effectuée à la fin de celle-ci. On attend que la classe soit pleinement préparée, que les listes des interfaces implémentées, des méthodes et champs soient disponibles, mais aucun code de cette classe n'a été exécuté. On peut ainsi récupérer les informations nécessaires à la construction de l'événement, avant que la fonction de préparation ne rende la main au reste de l'exécution.

4.2.3 Modifications apportées à SableVM

Dans cette section, nous décrivons les modifications apportées à SableVM afin de supporter le mécanisme d'événements proposé par la JPDA.

4.2.3.1 Ajout de Contextes d'exécution

Les événements ne sont générés que durant l'exécution du code Java. La fenêtre d'exécution au sommet de la pile du processus courant est alors de type Java. Une fois l'événement généré, on le soumet à la fonction de traitement des événements fournie par le client JVMDI. Cette fonction peut être amenée à effectuer des appels à des fonctions JNI ou JVMDI. Ces appels peuvent requérir la manipulation de références natives. Cependant, les fenêtres d'exécution Java ne sont pas munies de telles références. L'utilisation de fonction JNI ou JVMDI au sein de la fonction de traitement des événements n'est donc pas possible.

Nous avons ajouté aux types de fenêtres d'exécution un nouveau type de fenêtres internes. Celles-ci sont munies d'un jeu de références natives, permettant d'effectuer un appel aux fonctions JNI et JVMDI. Cette fenêtre est empilée dès l'appel de la fonction de traitement des événements, puis supprimée au retour de cette fonction.

4.2.3.2 Localisation du code octet

Tous les événements doivent contenir des informations contenant leur emplacement dans le code. SableVM modifie le code octet reçu en un code propre à SableVM. Lorsqu'une instruction est exécutée, cette modification induit un décalage entre sa position dans le code octet original et celle dans celui spécifique à SableVM. L'implantation des événements, et la nécessité d'indiquer la position dans le code octet original nous a poussé à créer un ensemble de fonctions internes à SableVM permettant de faire la conversion entre la valeur interne et la valeur dans le code original. Cette conversion permet aux outils recevant les événements de pouvoir lier la position dans le code octet

et la ligne de code Java, lorsque l'information est disponible (Eclipse surligne les lignes de code concernées, comme montré en figure 4.2).

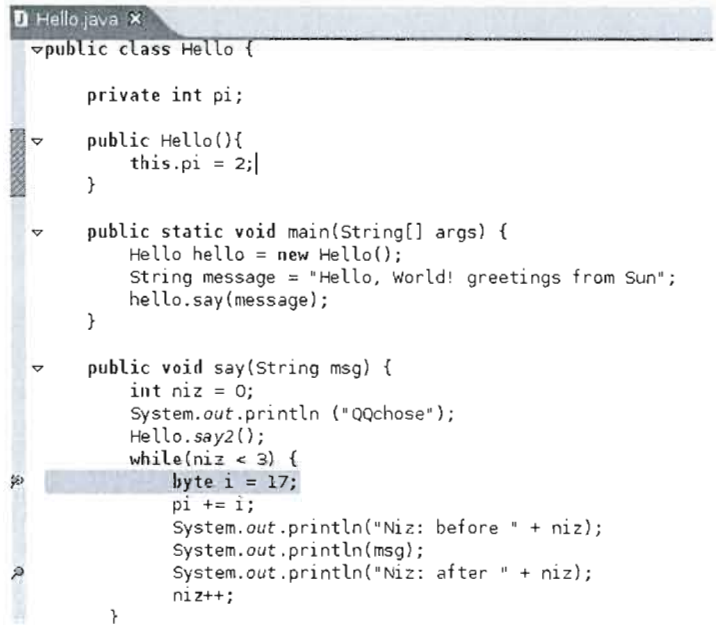


Fig. 4.2 Indication de la ligne contenant le code exécuté par Eclipse sous SableVM

4.3 Points d'arrêt

Un point d'arrêt permet à l'utilisateur de suspendre l'exécution de l'application à l'emplacement de son choix. Trois types de points d'arrêts existent : les *breakpoints*, les *catchpoints* et les *watchpoints*, définis à la section 2.2.2.1.

Dans le cadre de notre étude, seuls les deux premiers types de points d'arrêt ont été implantés, les fonctionnalités relatives aux *watchpoints* étant présentées comme optionnelles dans la JPDA. Les mises en œuvre des points d'arrêt implantés étant totalement différentes, nous présentons tout d'abord comment nous avons ajouté le support des *breakpoints* au sein de la machine virtuelle, avant de décrire les changements apportés aux mécanismes internes de SableVM. Nous suivrons ensuite la même démarche pour les *catchpoints*.

4.3.1 Implantation des *breakpoints*

4.3.1.1 Support des *breakpoints* dans SableVM

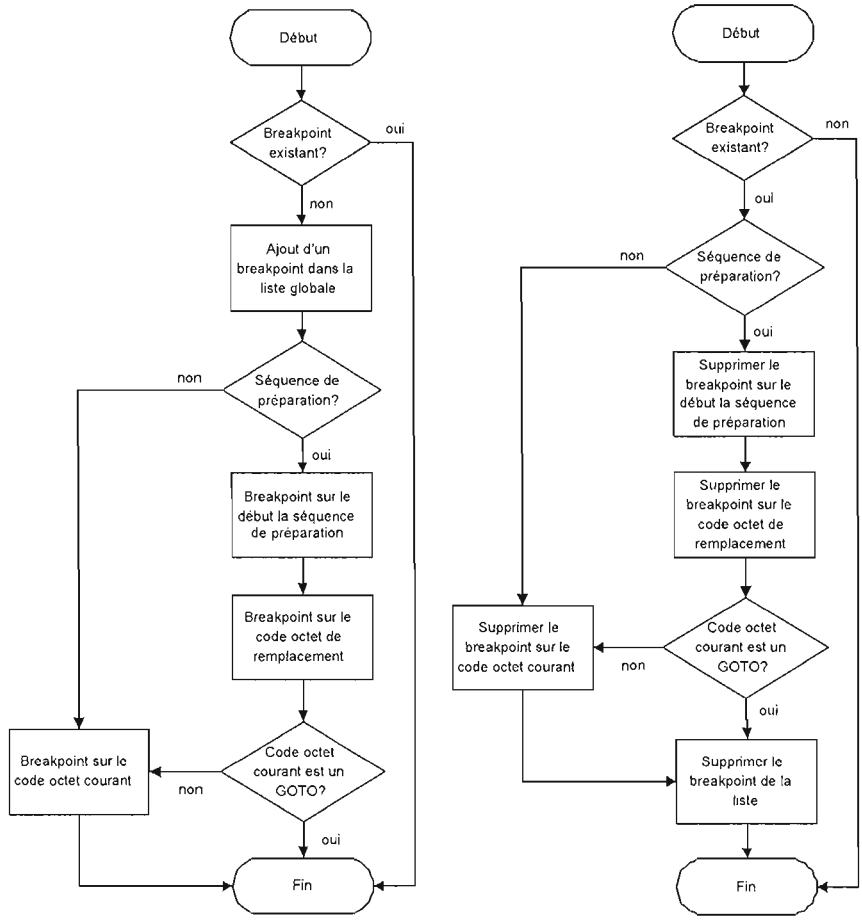
SableVM transforme le code octet en un code intermédiaire avant l'exécution (transformation présentée en section 2.1.4.3). Chaque octet de code est alors retranscrit en mot de code. La longueur d'un mot étant supérieure ou égale à celle d'un octet, cette opération ajoute à l'instruction un ensemble de bits inutilisés. Cet ensemble peut alors être exploité afin d'élargir le jeu d'opérations possibles, ou être employé à d'autres fins par les mécanismes internes de la machine virtuelle. Ainsi, nous avons utilisé le bit de poids fort du premier mot de l'instruction, correspondant au mot d'opération, afin d'indiquer la présence d'un *breakpoint*.

Lors de l'interprétation de cette instruction, le code ne correspondra à aucune instruction reconnue par SableVM. On se dirigera alors vers le code de gestion d'erreur. C'est au début de cette gestion d'erreur que l'on effectue un test sur le bit de poids fort de cette instruction. Si ce bit est à 1, on générera un événement indiquant la rencontre d'un *breakpoint*. Puis, au retour de cette génération, on exécutera l'instruction dont on récupérera le code original simplement en modifiant la valeur du premier bit. Si le bit rencontré est à 0, on exécutera le code de gestion d'erreur habituel.

La spécification Java autorise une préparation tardive : on peut charger et préparer classes et méthodes lors de la première utilisation de celles-ci. Au sein de SableVM, on insère à la place du premier appel à une méthode un appel à une séquence de préparation (Gagnon et Hendren, 2003). A l'issue de cette préparation, une instruction indiquant l'appel à la méthode préparée est substituée à l'appel à la préparation. Ce remplacement pourrait alors écraser une valeur contenant un point d'arrêt. Avant d'insérer un *breakpoint*, on vérifie donc que l'instruction modifiée ne corresponde pas à une séquence de préparation. Si c'est le cas, nous insérons également un *breakpoint* dans l'instruction qui écrasera l'appel à la préparation, à l'issue de celle-ci.

Finalement, le support aux *breakpoints* au sein de SableVM a été mis en place

grâce aux structures de données appropriées et à l'utilisation des algorithmes présentés en figure 4.3.



(a) Ajout d'un breakpoint (b) Suppression d'un breakpoint

Fig. 4.3 Algorithmes relatifs aux breakpoints

4.3.1.2 Modifications apportées à SableVM

Certains débogueurs permettent à l'utilisateur d'indiquer l'emplacement d'un point d'arrêt avant même d'avoir débuté l'exécution de l'application. Dans ce cas, il est possible que la méthode contenant la ligne de code indiquée n'aient été ni chargée, ni préparée. Le code octet correspondant n'est alors pas encore disponible. La session

de débogage ne devant pas modifier le comportement normal d'une application, nous avons dès lors éliminé l'option consistant à forcer le chargement et la préparation d'une méthode dès la mise en place d'un point d'arrêt.

SableVM étant tenue de mettre en place les points d'arrêt dans cette situation, une liste de points d'arrêt est maintenue au niveau de la machine virtuelle, afin d'être accessible par tous les processus. Dès que l'utilisateur indique la position d'un point d'arrêt, celui-ci est ajouté à cette liste. Si la méthode est déjà préparée, on l'insère à l'instruction correspondante. La préparation d'une méthode a donc été légèrement modifiée : à la fin de la préparation, nous parcourons la liste de points d'arrêts, afin de déterminer si certains d'entre eux doivent être mis en place au sein de la méthode venant d'être préparée. Si c'est le cas, on l'insère à l'emplacement indiqué. On s'assure ainsi que le point d'arrêt est installé avant l'exécution de la méthode.

4.3.1.3 Évaluation de la méthode choisie

Le choix de la signalisation sur le bit de poids fort n'est pas fortuit. Si l'ensemble d'instructions croît en utilisant les bits de manière linéaire, l'utilisation de ce bit limite au minimum l'accroissement possible du jeu d'instructions. Le mot d'opération plutôt que les mots de paramètres est, de plus, plus approprié à cette opération. Supposons qu'une opération prenne comme paramètre une adresse en mémoire. Les adresses ayant pour longueur un mot, si on utilisait un bit d'un mot de paramètre, on serait incapable de savoir si ce bit fait partie de l'adresse mémoire ou si un point d'arrêt a été mis en place. Ainsi, nous avons ajouté la contrainte qu'un *breakpoint* ne peut être disposé que sur le mot d'opération, et non sur celui de paramètre.

La méthode choisie présente comme avantage l'extrême simplicité d'implantation et de maintenance. En effet, elle est indépendante de l'instruction sur laquelle le *breakpoint* est posé, et ne nécessite donc pas de traitement, ou mémorisation particulière. De plus elle n'engendre qu'un volume minime de modification à l'interprétation de code octet.

Cependant, cette méthode ne peut être utilisée qu'avec l'interpréteur Switch, présent dans SableVM, ce qui peut nuire aux performances de la machine virtuelle. Elle n'est toutefois pas incompatible avec l'implantation de *breakpoints* propres à chaque type d'interprétation, mais la maintenance du code serait alors sensiblement complexifiée.

Notez qu'il est rare que l'on se soucie de la performance lors des opérations de débogage, l'exécution étant à maintes reprises suspendue puis relancée.

4.3.2 Implantation des *catchpoints*

4.3.2.1 Mécanisme d'exceptions

Lors de l'exécution d'un programme, certaines erreurs, dites exceptions, peuvent survenir alors qu'aucun bogue n'est présent dans le code. Par exemple, on peut tenter d'accéder à une ressource inaccessible au moment de l'exécution.

En Java, deux sortes d'exceptions peuvent être levées lors de l'exécution du code : les exceptions déclarées (ou checked exceptions), et les exceptions à l'exécution (ou runtime exceptions). Les exceptions déclarées sont ainsi nommées car le compilateur vérifie que (1) les méthodes levant des exceptions les déclarent bien dans leur définition et que (2) chaque méthode appelant une méthode levant une exception utilise bien un bloc protégé (une clause try, en Java) ou déclare cette exception dans sa définition. Les exceptions à l'exécution correspondent à des événements impossibles à localiser à la compilation (exceptions asynchrones), ou pouvant survenir à tout moment dans l'exécution du programme, comme les problèmes d'allocation de mémoire[1].

Le signalement d'une exception peut être automatique, s'il correspond à une exception définie dans le langage de programmation ou une bibliothèque fournie. Il peut également être déclenché par l'utilisation d'une primitive de signalement. Dans les deux cas, lorsqu'une exception est levée, deux comportements sont possibles. Si l'exception

---

[1]Source : http ://www.wikipedia.org

survient tôt dans la mise eu route de la machine virtuelle, ou si elle entraînerait une génération d'exceptions en boucle infinie, le cours normal de l'exécution est interrompu. Sinon, on remonte la chaîne d'appels jusqu'à trouver un traitement de cette exception (clause catch).

4.3.2.2 Support des *Catchpoints* dans SableVM

Bien que certains outils appellent également les *catchpoints* « breakpoints sur exceptions », les mécanismes intervenant dans la pose d'un *catchpoint* sont totalement différents de ceux utilisés pour les *breakpoints*. Cette différence est en partie due à la méthode de localisation du point d'arrêt. Pour les *breakpoints*, on spécifie la ligne de code avant laquelle suspendre l'exécution. Il n'est pas toujours possible de connaître à priori l'instruction (et donc la ligne de code) pouvant lever une exception, notamment dans le cas des exceptions à l'exécution. C'est pourquoi, plutôt que de spécifier l'emplacement dans le code, on indique le type d'exception qui suspendra l'exécution avant d'être levée.

Afin de supporter cette fonctionnalité au sein de SableVM, nous utilisons le mécanisme de génération d'événements mis en place. Ce mécanisme permettant le choix de la politique de suspension de l'exécution lors de la génération d'un événement, il est tout à fait approprié à la mise en place de *catchpoints* : on suspend l'exécution de l'application (de manière globale ou simplement le processus ayant généré l'exception) entre la création de l'objet exception et avant le traitement de celle-ci par la machine virtuelle. Si l'utilisateur décide de reprendre le cours de l'exécution, il pourra alors examiner le processus de traitement de l'exécution levée.

4.3.2.3 Évaluation de la méthode choisie

L'utilisation du mécanisme de génération d'événement pour les *catchpoints* étant dictée par les outils de débogage, les choix évalués ici sont principalement ceux mis en œuvre dans la génération des événements relatifs aux exceptions.

Lorsqu'un événement relatif à une exception est généré, il doit contenir, lorsque

possible, la position de la clause de traitement (clause *catch*, en Java). Le choix de renvoyer des valeurs nulles dans les champs correspondant lorsqu'une erreur intervient dans la recherche possède, à nos yeux, plusieurs avantages. D'une part, le mécanisme de recherche de cette clause au sein de l'événement étant similaire au mécanisme mis en œuvre par la machine virtuelle, on évite la duplication de la gestion des erreurs, peut recommandable pour la maintenance du code. D'autre part, si on se décidait à gérer les erreurs pouvant survenir, le recherche subséquente de la clause s'effectuerait sur la dernière exception levée, et non sur celle indiquée dans les champs de l'événement. Si cette clause est trouvée, les valeurs affichées paraîtraient incohérentes à l'utilisateur (d'une part les valeurs correspondant à une exception, d'autre part une clause de traitement correspondant à une autre exception).

L'inconvénient de cette méthode est qu'elle peut indiquer à l'utilisateur qu'aucune clause de traitement n'existe alors qu'une telle clause peut effectivement apparaître dans le code.

Si une exception est levée lors de la recherche, il serait possible de générer un nouvel événement pour cette nouvelle exception. Deux principales raisons nous ont conduit à ignorer cette option. D'une part, l'utilisateur ayant posé un *catchpoint* sur un certain type d'événement ne comprendrait pas pourquoi l'exécution est suspendue alors qu'on lui présente une exception d'un type différent (le type de l'exception levée durant la recherche). D'autre part, la spécification de la JPDA indique clairement que les événements doivent être transmis dès qu'ils surviennent et qu'aucun mécanisme de queue ne doit être mis en place. On ne peut donc pas transmettre le nouvel événement, relatif à la dernière exception levée, avant l'envoi de l'événement de l'exception initiale.

4.4 Contrôle des processus

Durant une session de débogage, il peut être utile à l'utilisateur de contrôler l'état des processus. De plus, suite à la génération d'événements, une politique de suspension peut être spécifiée, permettant de suspendre uniquement le processus ayant généré l'évé-

nement, tous les processus de la machine virtuelle ou aucun processus. Pour cela, un ensemble de fonctions est disponible au sein de la JVMDI.

Dans cette partie, nous présentons tout d'abord le support des fonctionnalités de débogage relatives à suspension/reprise d'exécution d'un processus. Ensuite, nous détaillons la création d'un processus de débogage. Enfin, nous exposons les modifications apportées aux structures de notre machine virtuelle, avant d'évaluer les choix faits.

4.4.1 Suspension et reprise de l'exécution d'un processus

Le contrôle de l'exécution des processus au sein de SableVM, décrit en section 2.1.4.4, a été implanté en utilisant des routines de notifications globales.

L'ajout de la possibilité de suspendre individuellement un processus engendre la nécessite de la conservation de la raison de la suspension d'un processus. La raison de la suspension d'un processus influe en effet sur les conditions nécessaires à la reprise de son activité : lors de la notification générale de reprise, on vérifie que chaque processus était en activité avant la suspension. On ne demandera donc pas aux processus suspendus individuellement avant la suspension générale de reprendre leur activité. De la même manière, lorsque l'on demande à un processus de reprendre son exécution, on vérifie que ce dernier n'est pas actuellement suspendu globalement. On a donc ajouté un indicateur, au sein de chaque processus, informant de la ou des raisons ayant mené à la suspension.

Lorsqu'un processus demande à un autre processus de suspendre son exécution, le demandeur attend que sa requête soit satisfaite avant de poursuivre son exécution. Le processus suspendu doit alors alerter le demandeur de sa mise en sommeil. Cette notification se fait via une variable de condition. La seule variable de condition alors disponible au sein de la structure de données représentant un processus étant utilisée pour l'envoi des requêtes de reprise d'exécution, nous en avons ajouté une, destinée à ce type de notification. Les demandes de reprise globale ou individuelle utilisent toujours l'ancienne variable de condition, associée aux routines pthread. Cependant, dans le cas d'une demande de reprise générale, la routine de diffusion générale était utilisée

(pthread_cond_broadcast). On utilise désormais la routine de notification individuelle, en parcourant tous les processus existants (pthread_cond_signal).

Il est également possible que plusieurs processus émettent une requête de suspension d'un même processus. Le processus visé doit alors prévenir tous les demandeurs de sa suspension. Nous avons alors doté la structure de données représentant un processus d'une liste conservant des références vers tous les processus ayant effectué une telle requête. Lors de la mise en sommeil d'un processus, on parcourt cette liste afin de prévenir tous les processus y apparaissant.

4.4.2 Création de processus de débogage

L'utilisation d'un outil de débogage ne doit pas modifier le comportement de l'application surveillée, comme indiqué dans la section 2.2. Aussi, il est déconseillé d'appeler des méthodes Java à des fins de débogage, une fois que l'application à déboguer est lancée. La JVMDI met alors à disposition des débogueurs une fonction permettant le lancement d'un nouveau processus, sans avoir à charger des sous-classes de java.lang.Thread ou des implantations de java.lang.Runnable. Ce nouveau processus, créé de manière native, requiert toutefois une instance de java.lang.Thread à laquelle il sera attaché. Il peut être utilisé pour la gestion des événements, pour la communication avec d'autres processus durant la phase de débogage ou à toute fin autre que l'exécution de l'application à déboguer. On s'assure ainsi que les mécanismes de débogage n'interfèrent pas avec l'exécution du code Java.

SableVM possédait déjà une fonction native de création de processus. La création d'un tel processus a nécessité la scission de cette fonction en deux : dès la fin de la création du nouvel objet, la première permet l'exécution d'une méthode Java, la seconde le lancement d'une fonction native définie par le débogueur. Dans tous les cas, des événements de création et de fin de processus sont générés. Cependant, le processus de débogage étant créé avant la mise en place des mécanismes de communication avec le débogueur, ils ne lui sont pas transmis, et ce processus reste invisible à l'utilisateur.

4.4.3 Modifications apportées à SableVM

Plus que l'ajout des structures de données utilisées pour supporter les nouvelles fonctionnalités, les principales modifications ont été apportées au niveau des algorithmes internes de SableVM. Les algorithmes de suspension et reprise globales de l'exécution sont particulièrement sensibles, du fait de leur corrélation au recyclage de la mémoire. en multipliant les raisons de suspension des processus, on ne doit pas altérer le recyclage de la mémoire, par exemple en diminuant la fréquence des appels au ramasse-miettes.

Il est également important de s'assurer que les transformations effectuées n'autorisent pas d'interblocage des processus (ou deadlock). La validation des modifications apportées a été effectuée lors des phases de test, en exécutant des applications lourdes, faisant appel à la gestion des processus.

Notez que les modifications que nous avons soumises à ces algorithmes ont été reprises par la communauté d'utilisateurs. Les algorithmes actuellement utilisés par SableVM ne sont plus ceux que nous avons mis en place, mais tiennent toutefois compte des changements que nous avons apportés aux statuts des processus. Ces changements résultent de l'évolution normale d'un logiciel libre, et offrent à SableVM une fiabilité accrue du fait de la contribution d'autres auteurs, peut-être plus compétents dans la gestion de processus.

4.4.3.1 Évaluation de la méthode choisie

Lors de l'implantation de la suspension individuelle d'un processus, nous avons opté pour la conservation du mécanisme de points de contrôle, permettant à chaque processus de vérifier régulièrement si une demande de suspension a été effectuée. On s'assure ainsi qu'un processus n'a pas été suspendu durant la manipulation d'un objet Java. En effet, une fois suspendu, un recyclage de la mémoire peut avoir lieu, et la référence utilisée avant la suspension peut devenir invalide à la reprise de l'exécution.

La conservation de la raison de la suspension plutôt que l'addition d'un nouveau

statut a été partiellement motivée par la quantité moindre de modifications nécessaires. Mais le critère principal a été l'absence de dégradation des performances, aussi bien pour le recyclage de la mémoire que pour la prévention d'interblocage. En effet, la création d'un nouveau statut pour chaque raison de suspension induit des modifications lourdes aux algorithmes déjà performants, entraînant l'addition de nombreux tests supplémentaires. Cette refonte des algorithmes devrait être effectuée à chaque addition d'une nouvelle cause de suspension, et la maintenance évolutive de SableVM en aurait pâti. La conservation de la raison de la suspension induit simplement l'ajout de nouvelles conditions de réveil des processus.

4.5 Gestion des classes chargées

4.5.1 Chargement de classes

Le chargement (ou loading) est le processus qui consiste à trouver la représentation binaire d'une classe ou interface, et de construire, à partir de cette représentation, un objet représentant cette classe ou interface (Lindholm et Yellin, 1999). Cette récupération est effectuée par un chargeur de classe (ou class loader). Il peut exister plusieurs types de chargeur au sein de la machine virtuelle : le chargeur d'amorce (bootstrap class loader), le chargeur du système (system class loader), et éventuellement un ou plusieurs chargeurs définis par l'utilisateur.

Un modèle de délégation a été introduit dans Java 2, organisant les chargeurs en hiérarchie ou arbre, plaçant le chargeur d'amorce au sommet. Lorsqu'une requête de chargement parvient à un chargeur, il peut soit effectuer ce chargement, soit le déléguer à un autre chargeur, qui se retrouve alors avec le même choix. Le chargeur ayant délégué le chargement est alors connu comme celui ayant initié le chargement, le chargeur ayant effectivement construit la class à partir du code est celui ayant défini le type.

La JVMDI propose des fonctionnalités permettant la récupération d'une part de toutes les classes chargées au sein de la machine virtuelle, d'autre part tous les types

initiés par un chargeur donné. SableVM conserve déjà une liste de tous les chargeurs de classes. De plus, chaque chargeur possède une liste de tous les types initiés. L'ensemble des classes chargées est alors un sous ensemble des types initiés. On peut donc récupérer la liste des classes chargées en parcourant celles des types initiés, mais le traitement est alors complexe. Il a donc fallu introduire de nouvelles structures, afin de rendre cette recherche plus efficace. Une fois ces structures mises en place, l'implantation de ces fonctionnalités ne consiste qu'au parcours des différentes listes.

Notez que SableVM ne permet présentement pas le déchargement d'une classe.

4.5.2 Modifications apportées à SableVM

Plusieurs chargeurs peuvent avoir initié le chargement d'une même classe. En plus de la liste des types initiés, nous avons ajouté à chaque chargeur la liste des classes qu'il a effectivement chargées. La maintenance de cette liste est assurée en ajoutant une référence à la classe chargée dans la liste, dès le chargement d'une classe. La récupération s'en trouve facilitée, puisque l'on parcourt simplement chaque liste de classes définies au sein de chaque chargeur.

De plus, nous avons ajouté à SableVM un compteur global, conservant le nombre de classes effectivement chargées. Ce compteur permet de ne pas parcourir les listes deux fois : une première fois pour compter les classes chargées et allouer la mémoire nécessaire, une seconde fois pour effectivement récupérer et renvoyer ces classes au client JVMDI. La récupération de l'ensemble des classes chargées est alors linéaire.

4.5.3 Évaluation de la méthode choisie

D'un point de vue algorithmique, nous avons tenté d'élaborer une stratégie de récupération la moins coûteuse possible, en ne parcourant la liste des classes une seule fois. Aucune recherche précise n'étant effectuée au sein de cette liste, il n'a pas été utile de maintenir la liste triée. Le coût d'une insertion est alors constant.

Cependant, notre solution est toutefois moins performante en matière d'utilisation d'espace mémoire. Si la conservation d'une machine virtuelle peu gourmande en espace avait été le souci premier, nous aurions alors opté pour la conception d'un algorithme effectuant la recherche parmi les listes de types initiés, supprimant les redondances, mais ne conservant pas de liste des types définis.

4.6 Autres fonctions

Dans cette section, nous présentons les fonctions de la JVMDI dont l'implantation n'a pas demandé de modifications majeures aux structures ou algorithmes internes de SableVM. L'aisance d'implantation n'étant pas proportionnelle à l'importance des fonctions, les fonctionnalités décrites dans cette section sont tout aussi utiles que celles détaillées précédemment.

4.6.1 Parcours des piles de contextes d'exécution

Chaque processus possède son propre pile de contextes d'exécution, chacune correspondant à l'appel d'une méthode native, Java ou créée à des fins internes à SableVM (voir partie 2.1.4.2). La représentation interne d'une fenêtre d'exécution dans SableVM conserve l'adresse de début et de fin de cette fenêtre. On peut alors facilement naviguer dans la pile de chaque processus.

Les fonctionnalités offertes par la JVMDI demandent principalement le nombre de fenêtres présentes dans la pile, la référence vers la fenêtre courante, la référence vers la fenêtre précédant une fenêtre donnée, et celle indiquant la position de l'instruction actuellement exécutée. Dans toutes ces fonctionnalités, nous dissimulons au débogueur l'existence de fenêtres créées pour des besoins internes. Ces fenêtres ne correspondent à aucun appel de méthode connue par le débogueur. De plus, leur structure est propre à SableVM et indiquer leur présence à l'utilisateur ne l'aiderait pas dans son débogage (en supposant que le bogue rencontré ne provienne pas de SableVM).

L'affichage de la pile d'exécution sur les débogueurs utilise également cette fonc-

tionnalité. Ainsi, la figure 4.4 illustre la manière dont Eclipse présente les contextes d'exécution existants au sein de la machine virtuelle Java exécutant l'application à déboguer (ici, notre version de SableVM exécutant un programme test).

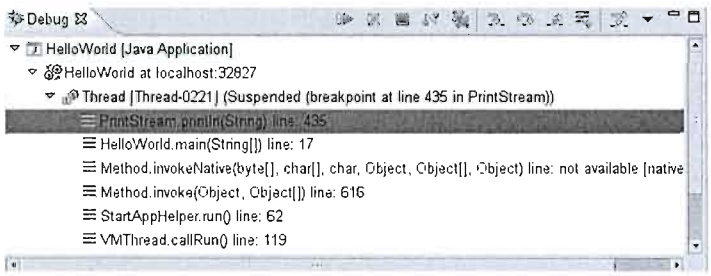


Fig. 4.4 Affichage de la pile d'appels dans Eclipse utilisant SableVM

4.6.2 Accès aux informations des entités Java

D'autres fonctions de la JVMDI répondent aux demandes de renseignements concernant certaines entités Java (classes, méthodes, champs et objets). Toutes les fonctions relatives à ces entités ne consistent qu'à récupérer une information interne, la mettre au format standard et la renvoyer. L'information recherchée étant souvent stockée dans la structure de données correspondant à l'entité observée, sa récupération est alors triviale. Nous ne présenterons donc pas toutes les fonctions de la JVMDI permettant la récupération du nom et de la signature d'une classe, méthode ou champs, du tableau de correspondance entre le code octet d'une méthode et les lignes de code Java, du nom du fichier contenant une classe Java, etc.

Bien que relativement aisées à développer, les fonctions d'accès aux entités Java représentent apportent une information primordiale aux utilisateurs. Elles permettent en effet une visualisation des valeurs des variables, et offrent la possibilité au développeur de confirmer ou non ses hypothèses quant à l'état du programme à un instant donné. L'accès à ces valeurs pouvant se faire également en écriture (en figure 4.5, on modifie la valeur d'un entier sous Eclipse avec notre version de SableVM), cette fonctionnalité permet, par exemple, de tester le comportement d'un programme avec des valeurs inattendues.

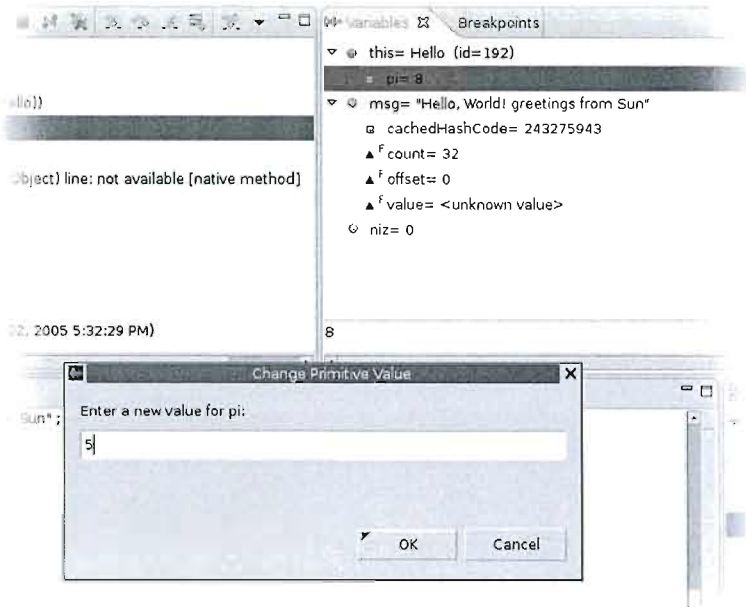


Fig. 4.5 Modification de la valeur d'une variable

4.6.3 Normes utilisées

Les normes publiées par Sun Microsystems sont souvent l'objet de mises à jour. La machine virtuelle est dotée de fonctions permettant de spécifier les versions des normes respectées, afin que le débogueur sache si les deux outils utilisent des normes compatibles. Nous avons implanté dans SableVM les fonctionnalités décrites par la dernière version stable des normes Java publiées par Sun Microsystems, la version 1.4.2. Depuis, une version 1.5 a été publiée.

Il existe, au sein de la JVMDI, certaines fonctionnalités définies comme optionnelles. Certaines d'entre elles ont été implantées au sein de SableVM, d'autres non. La liste de ces fonctionnalités varie donc d'une machine virtuelle à l'autre. Afin que le débogueur puisse être avisé des possibilités d'une machine virtuelle, une fonction de la JVMDI lui permet d'interroger la machine virtuelle sur l'ensemble des fonctions disponibles.

Chapitre V

DÉVELOPPEMENT DE LA BIBLIOTHÈQUE JDWP

La bibliothèque JDWP est l'entité de la JPDA responsable, entre autre, d'effectuer la liaison entre le débogueur et la machine virtuelle. Elle doit, entre autre, transformer les paquets JDWP reçus en provenance du débogueur en appels de fonctions des différentes interfaces de la machine virtuelle et transmettre à l'outil de débogage les événements générés par la machine virtuelle.

Ce chapitre traite des fonctionnalités mises en œuvre au sein de la bibliothèque JDWP. La première partie présente comment cette bibliothèque a été développée afin de pouvoir être accessible par la machine virtuelle. La partie suivante décrit l'organisation des objets manipulés par la bibliothèque JDWP. Ensuite, nous détaillons le mécanisme de traitement des événements reçus. Enfin, la dernière partie précise le traitement des commandes reçues.

5.1 Présentation de la bibliothèque JDWP

Nous revenons, dans ce chapitre, sur les détails du développement de cette entité de JPDA agissant entre débogueur et machine virtuelle Java.

Nous avons nommé cette entité bibliothèque JDWP d'une part car elle implémente le protocole JDWP. D'autre part, car plutôt que d'intégrer les fonctionnalités de débogage au sein de la machine virtuelle, nous avons opté pour un développement sous forme de bibliothèque dynamique C.

Une bibliothèque est un ensemble de symboles (fonctions, variables...) utilisés par le programme principal (la machine virtuelle) regroupés au sein d'un même agrégat. Le compilateur vérifie, durant la phase de mise en place des liens (linking), que tous les objets appelés par le programme principal sont soit liés au programme, soit dans l'une des bibliothèques dynamiques. Les objets présents dans cette bibliothèque ne sont toutefois pas insérés dans l'exécutable : une fois l'exécution débutée, un chargeur dynamique (ou dynamic loader) est responsable de monter les bibliothèques utilisées en mémoire et de les attacher à la copie du programme.

Ainsi, si SableVM n'est pas utilisée pour le débogage, les fonctionnalités ajoutées ne nuisent pas aux performances de la machine virtuelle, puisque la bibliothèque n'est pas montée en mémoire. On conserve également toute la modularité offerte par la JPDA : cette bibliothèque dynamique peut être utilisée par n'importe quelle machine virtuelle respectant les mêmes normes.

5.2 Gestion des objets

L'un des rôles de la bibliothèque JDWP est d'assurer la gérance des objets maniés par le débogueur et la machine virtuelle. Dans cette section, nous présentons tout d'abord de quelle manière la bibliothèque attribue des identifiants aux entités manipulées. La seconde partie détaille les structures de données utilisées pour l'entreposage des objets connus. Enfin, la dernière partie détaille la gestion de chaque type d'objet au sein de la bibliothèque JDWP.

5.2.1 Attribution d'identifiants

Le protocole JDWP ne permet pas aux objets de transiter sur le réseau et utilise des identifiants pour y faire référence. La bibliothèque JDWP assigne à toute entité (classe, objet, méthode...) un identifiant unique, qui restera valide durant toute la durée de vie de cette entité. Cet identifiant peut être global ou local, une fois couplé avec celui de l'entité englobant cette entité. C'est par exemple le cas des identifiants des

champs d'une classe, qui caractérisent un champ au sein d'une classe, et doivent donc être associés à celui de la classe correspondant.

Afin d'assurer l'unicité des identifiants globaux, un compteur a été mis en place au niveau de la bibliothèque. Ce compteur est accessible par toutes les fonctionnalités relatives au débogage, et attribue, de manière séquentielle, les identifiants des nouveaux objets manipulés. Connaissant toujours la valeur de l'identifiant le plus élevé, c'est également un moyen rudimentaire de vérifier la validité des identifiants reçus.

Un système similaire de compteur a également été mis en place afin de gérer les identifiants locaux. Lors des premiers tests, nous nous somme toutefois heurtés à un problème de gestion de ces identifiants. Prenons l'exemple d'une classe $C2$, comprenant trois méthodes, qui hériterait d'une classe $C1$, qui comprendrait deux méthodes. Notre système d'attribution octroierait les identifiants $(C1, 1)$ et $(C1, 2)$ d'une part, et $(C2, 1)$, $(C2, 2)$ et $(C2, 3)$ d'autre part. En héritant de $C1$, $C2$ a également récupéré les méthodes et champs de $C1$. La méthode $(C1, 1)$ peut également être appelée depuis la classe $C2$, mais l'identifiant 1 a déjà été attribué au sein de $C2$. Les identifiants, définis dans les spécifications comme étant uniques au sein d'une classe, doivent en fait l'être au sein d'une arborescence de classes. Le système alors mis en place ne convient pas. La gestion de l'arborescence des types étant trop coûteuse pour la simple attribution d'identifiants, nous avons alors opté pour l'attribution d'identifiants globaux à toutes les entités rencontrées.

Les méthodes et champs n'étant pas des objets Java, il nous est impossible de les entreposer dans la même structure de données que les autres. Des compteurs globaux spécifiques ont alors été mis en place, afin d'attribuer des identifiants aux champs et méthodes manipulés par la bibliothèque JDWP. Ainsi un nouveau compteur est mis place pour chaque type d'entité n'étant pas un objet Java. Les détails de la gestion de chaque type d'objets rencontrés est présenté dans la section 5.2.3.

5.2.2 Structures de données utilisées

Trois structures de données ont été utilisées, pour des besoins divers, durant le développement de la bibliothèque JDWP : les tableaux, les liste chaînées et les *splay trees*. Nous décrivons ici chacune de ces structures, sans aborder l'utilisation qui en est faite.

Un tableau est une liste séquentielle d'éléments. Chaque élément est associé à un indice unique au sein du tableau, permettant un accès direct à cet élément.

Une liste chaînée est une structure de données permettant de conserver un ensemble d'éléments, en associant à chaque élément une référence vers l'élément suivant ou précédent. Dans nos travaux, chaque élément ne devant être accédé qu'une seule fois par parcours de liste, nous n'avons utilisé que des listes simplement chaînées : chaque élément possède une référence vers l'élément suivant.

Un *splay tree* (Sleator et Tarjan, 1985) est un arbre binaire de recherche réarrangeant, permettant un accès rapide aux derniers éléments recherchés. A chaque recherche, l'élément retourné est placé au sommet de l'arbre, ce qui permet aux éléments accédés souvent d'être plus rapidement atteignables que les autres.

Le choix des arbres n'est pas optimal ici. Ce choix a été motivé par le fait que SableVM est dotée d'un mécanisme de génération de code. Ce mécanisme, couplé aux structures de données génériques déjà utilisées, a permis l'automatisation de l'adaptation des arbres déjà présents dans SableVM. L'utilisation de structures génériques impose la rédaction de fonctions permettant de comparer deux éléments de même type, pour chaque type. L'implantation des fonctionnalités de débogage en a toutefois été grandement accélérée, l'utilisation de telles structures ne requérant aucun test de validation supplémentaire.

5.2.3 Stockage et accès aux entités connues

Les entités connues sont toutes celles (classes, méthodes, objets...) ayant déjà été manipulées par la bibliothèque JDWP. Ces entités devant être accessibles par toutes les fonctionnalités relatives au débogage, leur entrepôt est centralisé au sein d'une même structure de données. Nous avons choisi de les conserver dans un tableau. Le tableau choisi est statique, le nombre d'objets contenus est alors plafonné. Afin d'éviter un gaspillage de mémoire, dans le cas d'un tableau trop important, ou un manque de place dans le cas contraire, on utilise le mécanisme de réallocation de la mémoire disponible en C. On peut alors ajuster la taille du tableau : on alloue un tableau initial, et lorsque celui-ci est plein, on double sa capacité.

Les identifiants sont attribués de manière séquentielle depuis la valeur 0. Dès qu'une entité se voit attribuer un identifiant, elle est immédiatement insérée dans le tableau. Ainsi, l'identifiant attribué représente également l'indice de l'élément du tableau contenant cette entité. Cette méthode permet un accès direct (en temps constant) à l'entité, une fois son identifiant connu.

Pour indiquer au débogueur quel objet est manipulé lors de l'appel d'une fonction, on doit être capable de récupérer efficacement son identifiant. On doit alors rechercher cet élément dans le tableau et retourner l'indice correspondant. Cette recherche peut être coûteuse, du fait de la diversité des objets contenus (comment comparer une méthode et une classe ?) et de l'accès séquentiel aux éléments.

Nous avons donc mis en place des index propres à chaque type d'entité présent dans le tableau. Cet index est trié selon des clés qui diffèrent selon la nature des objets, et contient l'identifiant des élément contenus. Les structures de données utilisées varient selon la nature des objets à indexer.

5.2.3.1 Gestion des objets

On doit attribuer aux objets Java des identifiants globaux. On doit donc mettre en place un index général pour les objets et déterminer une fonction de comparaison permettant de parcourir cet index.

La JNI propose une fonction permettant de déterminer si deux objets sont identiques, mais ne permet pas de les ordonner, de déterminer lequel est supérieur à l'autre. Cependant, chaque objet Java possède un haché (ou hashcode), généré par la machine virtuelle et invariant durant toute la durée de vie de l'objet. Ce hashcode est un entier auquel on peut accéder via une fonction de la JNI. La comparaison d'entier étant triviale, nous pouvons utiliser ce haché afin de déterminer si un objet est supérieur à un autre.

L'index utilisé est ici un *splay tree*, dont la clé est le haché de l'objet. Il est toujours possible que deux objets différents possèdent le même hashcode. Chaque nœud de l'index possède alors une liste chaînée d'objets possédant le même haché. On utilise alors la fonction IsSameObject de la JNI pour déterminer si un objet est identique à un autre déjà présent dans la liste, et donc retrouver son identifiant. Dans la pratique, si la fonction de hachage est bien définie, selon le nombre d'objets utilisés, la probabilité de collision est assez faible.

Les objets sont ajoutés aux structures (tableau et index), dès leur première utilisation par la bibliothèque JDWP.

5.2.3.2 Gestion des classes chargées

Au même titre que les objets, chaque classe doit avoir un identifiant global. Afin de pouvoir les indexer, il nous faut, de plus, une moyen de pouvoir comparer deux classes.

La signature d'une classe ne varie pas durant toute la durée de vie de cette classe. On peut donc l'utiliser afin de comparer deux classes. De plus, cette signature est une

chaîne de caractères, il est donc possible d'ordonner les classes selon l'ordre alphabétique de leur signature. Cette méthode permet l'utilisation d'index sous forme de *splay tree*, dont la clé est la signature de la classe.

Deux classes sont identiques si leur nom est le même et qu'elles ont été chargées par le même chargeur (Lindholm et Yellin, 1999). Il est donc possible que plusieurs classes possèdent la même signature. De fait, on doit également conserver, dans chaque nœud, une liste chaînée des classes ayant la même signature (figure 5.1). Pour les classes ayant une même signature, les chargeurs servent alors de discriminants.

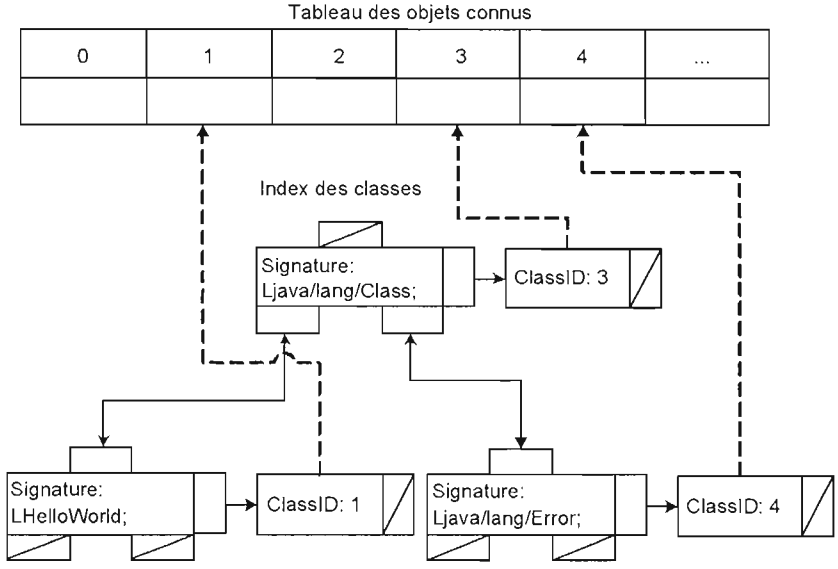


Fig. 5.1 Exemple d'index des classes connues

Le débogueur peut demander la liste de toutes les classes chargées au sein de la machine virtuelle. Cette demande est satisfaite en utilisant la fonction de la JVMDI permettant d'obtenir un telle liste. Cependant, on a vu que le traitement d'une telle demande pouvait être coûteux. Lors de nos expérimentations, nous avons réalisé que ce type de demande était régulièrement effectuée par le débogueur. Ainsi, avant l'initiation de la communication avec l'outil de débogage, nous faisons appel à la fonction JVMDI permettant d'obtenir toutes les classes chargées, et nous remplissons les structures cor-

respondantes (index et tableau). Les structures sont mises à jour à chaque rencontre d'une nouvelle classe. Les requêtes du débogueur ne consistent alors qu'au parcours de l'index.

Le remplissage anticipé de cette structure permet également une réponse plus rapide à d'autres requêtes relatives aux classes. Un débogueur peut demander d'obtenir la liste de toutes les classes ayant une signature donnée. Or, aucune fonction JVMDI ne permet d'obtenir directement une telle liste. On utilise alors l'index qui permet de satisfaire la requête du débogueur à moindre coût.

5.2.3.3 Gestion des champs et méthodes

Les champs et méthodes sont les deux types d'entités devant initialement être identifiées de manière unique au sein d'une classe (donc munies d'identifiants locaux). Cependant pour les raisons détaillées auparavant, nous avons opté pour l'attribution d'identifiants globaux à ces types d'entité. Mais les champs et méthodes ne sont pas des objets Java. Il nous est donc impossible de leur appliquer les fonctions de comparaisons utilisées pour les objets. De plus, en leur attribuant des identifiants obtenus depuis le compteur global d'objets, on condamne certains éléments du tableau à rester vides. Pour éviter ce gaspillage de mémoire, nous avons décidé de mettre en place deux compteurs globaux, l'un pour les méthodes, l'autres pour les champs, permettant de leur attribuer des identifiants. Le tableau ne pouvant contenir que des objets Java, il ne peut pas servir à l'entrepôt des méthodes et champs. On les stocke donc directement dans les nœuds des index utilisés.

Nous utilisons une liste chaînée afin d'indexer les différentes méthodes manipulées. Les fonctions de comparaison de méthodes indiquent uniquement si deux méthodes sont similaires, sans indication sur un ordre quelconque en cas de différence (c'est-à-dire savoir laquelle est supérieure à l'autre). Il nous a donc été impossible d'utiliser une structures arborescente. Le parcours de cette liste, bien que moins efficace qu'un arbre, permet toutefois des recherches plus performantes qu'au sein du tableau dans son intégralité.

Chaque nœud de l'index comporte la méthode elle-même, son identifiant, en plus d'une référence vers le nœud suivant. Il comporte également des informations relatives à la méthode, régulièrement demandées par le débogueur : nom, signature et droits d'accès (private, public...) de la méthode sont conservés afin de simplifier les accès répétés à ces informations. Ces dernières ont été ajoutées aux nœuds suite à l'analyse des requêtes reçues, après une première série de tests avec un débogueur.

Les champs sont gérés de manière analogue, par un index de même type et contenant les mêmes informations.

Un nouveau nœud d'index est créé à chaque fois qu'un champ ou qu'une méthode est manipulée pour la première fois par la bibliothèque JDWP.

5.2.3.4 Gestion des processus

Les processus doivent avoir des identifiants globaux. Les processus étant des objets, on utilise le compteur global afin de leur attribuer un identifiant. De plus, on peut avoir recours à leur hashcode afin de les dissocier.

Le nombre de processus étant relativement faible pour des applications standard, l'index utilisé est une liste chaînée. Il est important de dissocier les processus utilisés pour des besoins de débogage de ceux utilisés par l'application. En effet, l'existence des premiers est cachée au débogueur afin que l'utilisateur ne soit pas en mesure d'en suspendre l'exécution. Deux index sont alors conservés : le premier pour les processus de débogage, le second pour les autres.

La liste des processus utilisés pour le débogage est créée avant le lancement de l'application. Tous les processus lancés après cet instant sont considérés comme utilisés par l'application déboguée.

L'index des processus de débogage n'est mis à jour que pour la suppression de processus, puisque qu'aucun processus de débogage n'est créé par la suite. L'autre index est mis à jour à chaque création ou fin d'exécution de processus, les créations et fins de

processus étant indiquées par la génération d'événements.

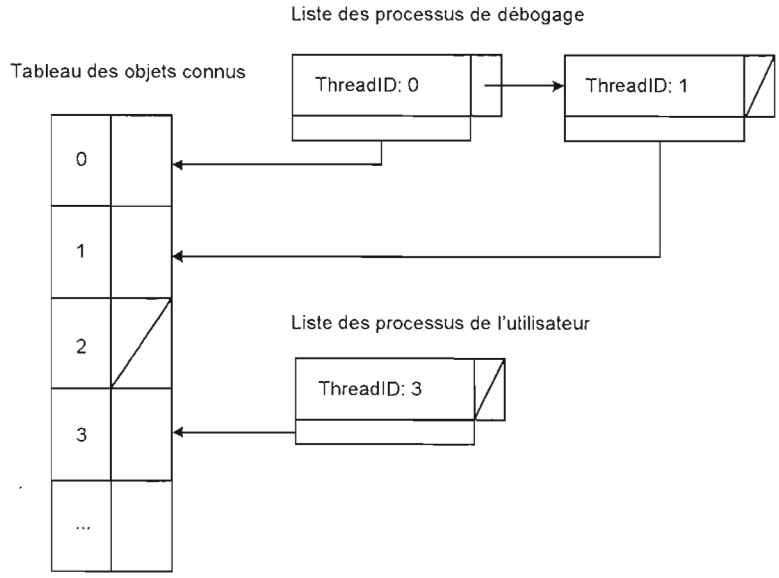


Fig. 5.2 Exemple de liste des processus

5.2.3.5 Gestion des contextes d'exécution

Bien que relatives aux processus les contextes d'exécution doivent avoir des identifiants indépendants, donc globaux. Les contextes d'exécution n'étant pas des objets, nous ne pouvons utiliser le compteur global associé au tableau d'objets connus afin de les identifier. Un compteur spécifique aux contextes d'exécution est donc mis en place.

De la même manière que pour les méthodes et champs, il est impossible, lors de la comparaison, de déterminer quel contexte est inférieur à un autre. On peut simplement déterminer si deux contextes sont identiques. De fait, on utilise un index sous forme de liste chaînée. Chaque nœud comporte le contexte d'exécution elle-même, son identifiant et une référence vers le nœud suivant. De plus, on conserve les informations les plus souvent demandées, afin de faciliter les prochaines requêtes relatives aux contextes : identifiants du processus, de la classe et la méthode correspondants ainsi que l'emplacement de l'appel dans le code Java ayant mené à la création de ce contexte.

Les contextes d'exécution étant valides uniquement durant la durée de vie du processus les contenant, l'index est mis à jour d'une part lors de leur première utilisation (ajout d'un nouveau nœud), mais également lors de la fin d'exécution d'un processus (suppression de tous les nœuds ayant le même identifiant de processus).

5.3 Gestion des événements

Le mécanisme de génération d'événements de la machine virtuelle a été présenté dans la partie 4.2. Les événements transmis arrivent ensuite à la bibliothèque JDWP, alors informée du cours de l'exécution de l'application à déboguer.

Cette partie traite de la gestion des événements reçus par la bibliothèque JDWP. Nous examinons d'abord le mécanisme de filtrage des événements, avant leur envoi au débogueur. Puis nous détaillons la fonction de gestion fournie par le client JVMDI.

5.3.1 Filtrage des événements

Au sein de la machine virtuelle, des séries d'indicateurs, l'une globale et propre à la machine virtuelle, les autres locales et spécifiques à chaque processus, forment un premier filtre. Ce filtre détermine quels événements seront transmis à la bibliothèque JDWP.

Le débogueur peut spécifier quels types d'événement lui faire parvenir, et fournir des tests ou conditions supplémentaires. Les événements doivent alors appartenir aux types indiqués et réussir les tests spécifiés afin d'être transmis au débogueur. La bibliothèque JDWP est responsable de l'établissement des tests et de la transmission des événements.

5.3.1.1 Stockage des tests

Les spécifications de la JPDA décrivent les différents tests pouvant être fournis par le débogueur. Ces tests sont, pour la grande majorité, indépendants de la nature des

événements. On associe cependant à chaque test à un type d'événement, pour affiner la granularité du tri effectué parmi les événements générés. On pourra, par exemple, spécifier les conditions de position (classes, méthodes et lignes) associées aux événements de *breakpoints*, permettant de déterminer quels points d'arrêt doivent effectivement suspendre l'application .

L'ensemble des tests fournis par le débogueur doit être conservé au sein de la bibliothèque JDWP. Pour ce faire, nous avons mis en place un tableau dont chaque indice correspond à un type d'événement. Chaque élément de ce tableau est une liste de requêtes, comprenant chacune un identifiant et une liste de tests. La bibliothèque est responsable de l'attribution d'identifiant à chaque requête. Afin de ne pas avoir à mettre en place un nouveau mécanisme de gestion des identifiants (compteur...), chaque requête se voit attribuer l'identifiant du paquet JDWP l'ayant transportée. Chaque paquet ayant un identifiant unique durant toute la session de débogage, la politique de gestion des identifiants est respectée. Le format des paquets JDWP est décrit dans la section 3.3.2.

Ainsi, le schéma de la figure 5.3 décrit la structure de données accueillant une requête comprenant deux tests.

L'ordre des tests spécifiés est important : les tests doivent être passés dans l'ordre indiqué par le débogueur et sont donc ajoutés à la liste de la requête dans ce même ordre. L'échec à un test implique l'arrêt du parcours de la liste des tests. En effet, certains tests possèdent des conditions changeant à chaque fois que le test est passé (un compte à rebours, par exemple). Ces tests ne doivent alors être passés que si tous les tests précédents ont été réussis, afin de ne pas biaiser le comportement attendu.

5.3.1.2 Mécanisme de filtrage

Dès la réception d'un événement, avant de pouvoir le transmettre au débogueur, on doit s'assurer qu'il répond bien aux critères dictés par le débogueur.

On récupère tout d'abord le type de l'événement, et on parcourt la liste des re-

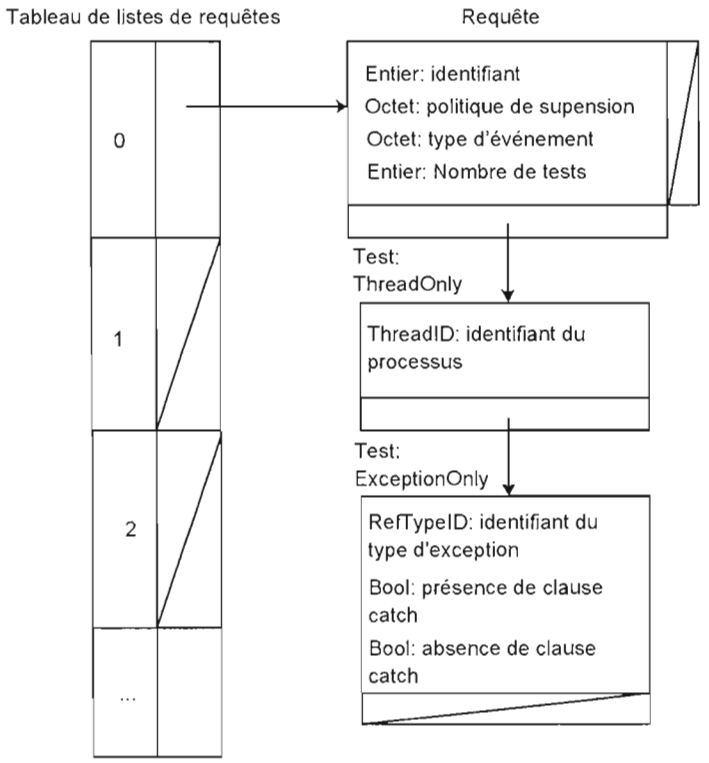


Fig. 5.3 Exemple de conservation d'une requête d'événements

quêtes reçues par le débogueur correspondant à ce type. S'il existe une requête satisfaite, alors on pourra transmettre l'événement. Sinon, le débogueur ne sera pas notifié de cet événement. Si la liste des requêtes associée à ce type d'événement est vide, le comportement à adopter dépend du type : si l'événement est du type *breakpoint* ou exécution pas à pas (*single step*), alors on considère qu'aucune requête n'est satisfaite, et on ne transmet pas l'événement. Si l'événement est d'un autre type, alors on peut l'envoyer au débogueur. Ce comportement n'est pas décrit par la JPDA, mais il découle des deux raisonnements suivants : (1) le débogueur spécifie les points d'arrêts via ce mécanisme de tests. Si aucun test n'a été spécifié, aucun *breakpoint* ne devrait être présent dans le code. Si toutefois on en rencontrait un, on ne doit alors pas le prendre en compte. (2) Par défaut, les indicateurs de la machine virtuelle n'autorisent pas la génération d'événements

relatifs à l'exécution pas à pas. Le seul outil permettant au débogueur d'en modifier la valeur est le mécanisme de requêtes utilisés ici. Si aucune requête n'est dans la liste, alors l'exécution pas à pas ne doit pas être activée. Les événements correspondants ne doivent alors pas être transmis au débogueur.

Si on n'interdit pas l'envoi de l'événement à l'issue du parcourt de cette liste de requêtes, on encode dans un paquet JDWP et on transmet ce dernier au débogueur. Sinon on continue l'exécution de l'application.

5.3.1.3 Filtres

Chaque requête d'événement reçue par la bibliothèque JDWP peut contenir une liste de conditions que les événements du type indiqué doivent satisfaire afin d'être transmis. Cette section décrit l'implantation des différents types de filtre possibles.

Le premier test possible est un compte à rebours : on indique combien d'événements ignorer avant de commencer à les prendre en compte. Ainsi, à chaque fois que ce test est rencontré, on soustrait 1 à la valeur du compteur, si celle-ci est positive. Le test est réussi si la valeur du compteur est à 0.

Il est également possible d'indiquer quel processus ou classe peut générer des événements. On indique alors l'identifiant du processus ou de la classe, le test étant réussi si celui de la classe ou du processus ayant généré cet événement est égal à celui indiqué. La structure de données correspondant à ce filtre ne contient alors qu'en entier dont on modifie la valeur à chaque examen.

Deux autres conditions sont relatives à la classe à l'origine de l'événement : on restreint la génération d'événements aux classes dont la signature correspond, à celle indiquée dans la condition (ClassMatch). On peut aussi indiquer la signature des classes dont on ne veut pas transmettre les événements (ClassExclude). On peut indiquer simplement le début ou la fin des signatures, utilisant le signe * pour la compléter. La comparaison des signatures se fait alors jusqu'à ou à partir de ce caractère *.

Pour les événements offrant des informations précises sur l'emplacement du code ayant généré cet événement, il est possible de filtrer ces événements selon les classes, méthodes et lignes (en indiquant le déplacement dans le code octet) de ceux-ci. Ce type de filtre a été abordé lors de la mise en place de *breakpoints*.

Pour les événements relatifs aux champs, on peut, de la même manière que pour les processus ou les classes, indiquer l'identifiant du champ concerné. Cependant les événements relatifs aux champs n'étant pas mis en place au sein de SableVM, nous n'avons pas pu tester ce type de filtre.

Afin de mettre en place des *catchpoints*, on utilise le mécanisme de filtre pour les événements relatifs aux exceptions : le débogueur précise l'identifiant du type d'exception concerné, ainsi que deux booléens indiquant si le test est réussi (1) lorsqu'il existe une clause catch relative à l'exception soulevée, (2) lorsque l'exception soulevée n'est gérée par aucune clause catch. Si cette clause n'est pas atteignable, les valeurs de ces champs sont à 0. On vérifie alors la valeur de ces champs afin de déterminer si le test est réussi. De plus, on récupère le type de l'exception soulevée grâce à la fonction JNI GetObjectClass. On vérifie ensuite si ce type correspond à celui indiqué grâce à la fonction JNI IsAssignableFrom.

Enfin, ces conditions sont utiles lors de la gestion de l'exécution pas à pas (voir section 2.2.2.2). Dans ce cas, ces tests ne sont ajoutés que par des requêtes d'événements relatifs à l'exécution pas à pas. Ils permettent de spécifier la taille du pas à effectuer lors de l'exécution. Ce type de condition a impliqué une conservation d'informations supplémentaires au sein de la bibliothèque JDWP : lorsque la requête relatif aux événements d'exécution pas à pas est reçue, on conserve l'emplacement actuel (identifiants du processus, de la classe, de la méthode et déplacement dans le code octet), ainsi que la profondeur de la pile de contextes d'exécution du processus ayant effectué cette requête. ci-après, on décrit les types de conditions possibles lors de l'exécution pas à pas.

Si la condition indique que l'on veut passer à l'instruction suivante (*step in*), le test est réussi si la profondeur de la pile de contextes est modifiée (relativement à celle

conservées dans la bibliothèque JDWP), ce qui indique qu'un appel à une méthode a été effectué. Si la profondeur est inchangée, alors les événements passent également le test, puisqu'en cas de boucle sur une seule ligne, l'instruction suivante correspond à la même ligne.

Si la condition indique que l'on veut passer à la ligne suivante en ignorant les appels aux méthodes (*step over*), la profondeur de la pile doit être égale ou inférieure pour que le test soit réussi. Si cette profondeur est égale, on vérifie alors que la ligne correspondante est différente.

Si la condition indique que l'on veut sauter à la fin de la méthode courante (*step out*), tous les événements générés alors que la pile de contexte d'exécution a une profondeur supérieure ou égale à celle conservée au sein de la bibliothèque JDWP sont ignorés.

5.3.2 Fonction de gestion des événements

La bibliothèque JDWP tient le rôle de client JVMDI, du point de vue de la machine virtuelle. Elle est donc tenue de fournir une fonction de gestion des événements, appelée par la machine virtuelle lors de la génération de ceux-ci. Dans notre cas, cette fonction est transmise à la machine virtuelle dès l'acquisition de l'interface JVMDI, avant l'établissement de la communication avec le débogueur.

Certains événements entraînent une mise à jour des structures de données internes à la bibliothèque JDWP. Ces mises à jour sont effectuées au sein de la fonction de traitement des événements. Ce sont ces mises à jour que nous allons décrire ici.

Lorsqu'un événement relatif à la préparation d'une classe survient, ceci indique que cette classe n'a pas encore été manipulée par la bibliothèque JDWP. Elle n'apparaît donc pas encore dans les structures internes de la bibliothèque. L'événement étant généré dès la fin de la préparation de la classe, on peut procéder à la collecte des informations relatives à cette classe. On l'ajoute alors au tableau des objets connus, et on met à jour

l'index des classes avec les informations recueillies.

Les processus utilisés pour le débogage sont lancés avant l'exécution de l'application. Si un événement de création de processus est reçu, on sait alors que ce processus est utilisé dans le cadre de l'exécution de l'application. On lui attribue tout d'abord un identifiant, puis on l'ajoute au tableau des objets connus, avant de mettre à jour l'index relatif aux processus de l'utilisateurs.

Les processus relatifs au débogage ne sont normalement pas arrêtés durant la session de débogage. Si un événement relatif à la fin d'un processus est généré, on doit alors invalider tous les identifiants des contextes d'exécution relative à ce processus. Dans l'index des contextes d'exécution, on supprime alors tous les nœuds comprenant un identifiant de processus identique à celui du processus ayant généré l'événement. De plus, dans le tableau d'objets connus, l'indice correspondant contient alors un objet NULL.

Les autres types d'événements n'engendrent aucune modification des structures de la bibliothèque JDWP. Une fois les modifications apportées, si besoin, les événements doivent passer les tests décrits en partie 5.3.1. Puis, si ces conditions sont satisfaites, on doit créer un paquet JDWP contenant les informations relatives à cet événement, et le transmettre au débogueur. La fonction de gestion d'événements utilise les fonctions d'encodage des valeurs en paquet JDWP, mises en place par la bibliothèque JDWP, afin de créer un tel paquet. Le canal de communication étant partagé par les mécanismes de commandes/réponses et de génération d'événement, l'écriture sur ce canal est protégé par un verrou. Ce verrou est acquis avant l'envoi de chaque paquet, et libéré une fois l'envoi effectué. On évite ainsi les accès concurrents à ce canal et on s'assure de la cohérence et de l'intégralité des paquets envoyés.

5.4 Traitement des paquets

5.4.1 Canal de communication

La spécification de la JPDA ne limite pas les moyens de transport de l'information entre débogueur et machine virtuelle. Dans le cadre de notre développement, nous n'avons implanté que celui requis par les débogueurs que nous avons utilisés lors des phases de test et de validation. Plus précisément, la communication entre débogueur et bibliothèque JDWP s'effectue via un socket. De plus, le débogueur étant responsable du choix du port sur lequel les communications transiteront, aucun mécanisme de balayage des ports afin d'en choisir un inutilisé n'est mis en place au sein de notre bibliothèque.

Le processus de débogage écoute en permanence sur ce canal, afin de réceptionner les paquets en provenance du débogueur et d'exécuter les actions appropriées à chaque type de paquet. Une fois que ce canal n'est plus disponible (i.e. la connexion entre le débogueur et la bibliothèque est coupée), on considère que la session de débogage est terminée, puisqu'aucun mécanisme de reconnexion n'existe.

5.4.2 Réception des commandes

En dehors de la gestion des événements, la grande partie du travail effectué par notre bibliothèque consiste en la transformation de paquets JDWP reçus en appels de fonctions de l'interface JVMDI. Certaines requêtes ne consistent qu'en un appel à une fonction JVMDI et à l'encodage et la transmission de la réponse au débogueur. D'autres ne nécessitent que le parcours d'une structure de données interne à la bibliothèque JDWP. Enfin, un troisième type de commande requiert un traitement plus complexe. Ce sont ces dernières commandes que nous allons décrire dans cette section.

C'est ici que le choix des structures de données revêt toute son importance : des commandes demandant initialement un traitement complexe sont réduites au parcours d'une des structures internes de la bibliothèque JDWP. C'est par exemple le cas des commandes relatives aux classes, ramenée à un simple accès à un arbre, et qui n'appa-

raîtront donc pas dans cette section.

5.4.2.1 Gestion des processus

Les processus visibles par l'utilisateur sont uniquement ceux liés à l'exécution de l'application à déboguer. A chaque commande liée aux processus, on vérifie avant tout que l'utilisateur est autorisé à manipuler ce processus (i.e. que le processus n'est pas un processus de débogage).

Si l'utilisateur demande plusieurs fois la suspension d'un processus, il devra en demander autant de fois la reprise afin que celle-ci soit effective. La bibliothèque JDWP doit alors conserver un compteur de demandes de suspension pour chaque processus. Nous avons donc ajouté ce compteur à chaque nœud de l'index des processus. La valeur du compteur est augmntée à chaque demande de suspension, et réduite à chaque demande de reprise d'exécution. Les fonctions JVMDI de suspension et reprise ne sont appelées que lorsque cette valeur atteint 0.

Les processus doivent être dotés de nom. Cependant, SableVM ne comporte pas encore cette fonctionnalité. Le débogueur utilisé affiche en effet les noms des processus, afin que l'utilisateur puisse les dissocier lors de son débogage. De fait, la bibliothèque JDWP attribue un nom à chaque processus accessible par l'utilisateur. Les noms attribués sont la chaîne de caractère « Thread- » suivi de l'identifiant du processus (par exemple Thread-123456 pour le processus dont l'objet à l'identifiant 123456).

En Java, les processus peuvent être réunis en groupes. SableVM ne permettant pas de manipuler des groupes de processus, là aussi la bibliothèque JDWP doit simuler ce comportement. Ainsi, deux groupes sont créés par défaut : le premier pour les processus liés à l'exécution de l'application, le second pour ceux relatifs au débogage, et invisible à l'utilisateur. Évidemment, l'implantation des commandes relatives aux groupes de processus est biaisée au sein de notre bibliothèque. Les groupes de processus pouvant également être munis de noms, nous avons attribué le nom « main » à celui accessible par l'utilisateur, et « system » à l'autre.

L'utilisateur peut accéder à la pile de contextes d'exécution. En plus demander le contexte courant ou la profondeur de la pile, il peut également demander la récupération de plusieurs contextes au sein de la pile. Il indique alors la position du premier contexte à récupérer, ainsi que le nombre de contextes désirés. La bibliothèque JDWP se charge ensuite d'accéder à cette pile et de retourner ces valeurs, via des fonctions JVMDI. Ces fonctions JVMDI ayant été implantées de manière à dissimuler les contextes internes au client, aucun traitement particulier n'est nécessaire, à l'exception de la gestion des erreurs pouvant survenir (demande d'une nombre de contextes supérieur à ceux effectivement dans la pile...).

5.4.2.2 Accès aux contextes d'exécution

Plusieurs commandes sont relatives aux contextes d'exécution. Cette section en dresse la liste, décrivant à chaque fois le traitement effectué.

L'accès aux contextes d'exécution se fait essentiellement pour récupérer ou modifier les variables utilisées par une méthode. Cet accès se fait en indiquant l'identifiant du processus et celui de le contexte à examiner, ainsi qu'en précisant le nombre de variables accédées. De plus, on doit spécifier la position dans le groupe de variables ainsi que le type de la variable à accéder. Il existe une fonction JNI spécifique à chaque type de variable. On appelle alors la fonction adéquate pour chaque variable.

Une commande permet au débogueur d'accéder à l'objet *this* du contexte d'exécution. Cet objet est une référence vers l'instance ayant effectué l'appel à la méthode associée à ce contexte. Si la méthode est statique (au sens Java) ou native, on doit retourner une référence vers la valeur NULL.

La spécification de la machine virtuelle indique que cette référence est passée en paramètre 0, lors de l'appel à une fonction (Lindholm et Yellin, 1999). Elle est donc la première variable locale (à l'indice 0) dans le contexte d'exécution. Cependant, rien ne garantit que cette valeur soit conservée ou qu'elle ne soit pas déplacée au cours de l'exécution de la méthode.

Cette fonctionnalité et la seule pour laquelle nous avons développé deux implantation : l'une spécifique à SableVM, l'autre générique. Dans l'implantation générique, nous retournons simplement la valeur de la variable à la position 0, si l type de le contexte est valide (Java uniquement). Pour l'implantation spécifique à SableVM, nous avons ajouté à la JVMDI une fonction permettant l'accès à cette variable : GetFrameThisObject. SableVM conserve, dans la structure interne des contextes d'exécution, une référence vers l'objet ayant initié l'appel à cette méthode. La nouvelle fonction s'assure tout d'abord de la nature du contexte d'exécution, et renvoie ensuite cette référence.

L'existence de cette fonction spécifique à SableVM est contraire à notre principe initial d'indépendance de la bibliothèque JDWP. Cependant, il nous est apparu important qu'au-delà de la conformité aux normes, nous puissions offrir à nos utilisateurs un outil robuste et fiable. Cette entorse est toutefois compensée par une seconde implantation, conforme à la spécification.

5.4.2.3 Accès aux champs

Les opérations de récupération de la valeur des champs d'une classe ou d'une instance utilisent les fonctions de la JNI. Cependant, ces opérations sont typées, et le type de la valeur retournée doit être connu au préalable : on n'utilisera pas la même fonctions pour accéder à la valeur d'une chaîne de caractères, d'un objet Java ou d'un entier. Pour déterminer la nature d'un champ, la bibliothèque obtient tout d'abord le champ grâce à une fonction JVMDI, puis utilise la fonction JVMDI permettant l'accès à la signature de ce champ. Cette signature indique la nature du champ. Une fois obtenue, il est possible d'appeler la fonction JNI ad hoc pour obtenir la valeur de ce champ. Si cette valeur doit être renvoyée au débogueur, elle doit être précédée d'un octet indiquant son type, lors de l'encodage dans le paquet JDWP.

Lors d'accès aux différents champs ou méthodes d'une classe, on ne respecte pas les droits d'accès (private, public...). Le débogueur peut ainsi demander la valeur d'un champ défini comme private en Java.

5.4.2.4 Manipulation des tableaux

En Java, un tableau peut contenir tout type d'entité, dans la mesure où toutes les éléments sont de même nature. On peut donc avoir un tableau contenant des objets Java, des objets de type primitif (entiers, booléens...), des tableaux, des chaînes de caractères, etc.

De la même manière que pour l'accès aux champs, l'accès aux éléments d'un tableau est effectué via une fonction JNI propre à chaque type. Il nous faut donc déterminer ce type avant de pouvoir accéder au contenu du tableau. On récupère alors la signature du tableau, nous indiquant le type des objets contenus. Puis on peut faire appel aux fonctions JNI adéquates, en portant une attention particulière à la gestion d'erreurs (indice invalide...).

5.4.2.5 Requêtes d'événements

Lorsque le débogueur soumet une requête concernant la réception d'événements, la première étape consiste à constituer les objets représentant la requête et éventuellement les filtres. On les ajoute ensuite à la structure de données de la bibliothèque JDWP (tableau de listes de requêtes).

La bibliothèque JDWP conserve également une liste d'indicateurs stipulant quels types d'événements doivent être transmis au débogueur. Si la liste était vide avant l'ajout de la dernière requête, on s'assure que la bibliothèque autorise bien l'envoi de tels événements, en modifiant les indicateurs appropriés. De plus, si besoin, on effectue la même modification au sein de la machine virtuelle, par le biais des fonctions JVMDI associées. Ainsi, on sépare clairement les autorisations de notifications envoyées de la machine virtuelle à la bibliothèque, de celles envoyées de la bibliothèque vers le débogueur. Cette séparation peut permettre, par exemple, à la bibliothèque de toujours recevoir les événements relatifs à la création et à la fin d'exécution des threads, afin de mettre à jours ses structures de données internes, mais de ne pas envoyer ces événements

au débogueur, si ce dernier n'en a aucune utilité.

La commande de suppression d'une requête ne contient que l'identifiant de celle-ci. On parcourt alors toutes les listes de requêtes afin de retrouver puis supprimer celle désignée. Si, après la suppression de la requête, la liste la contenant devient vide, on attribue aux indicateurs de la machine virtuelle leur valeur par défaut (la valeur initiale indiquée par les spécifications de la JPDA).

Si la commande d'ajout ou suppression d'une requête relative aux points d'arrêts est reçue, on procède à l'appel aux fonctions JVMDI permettant l'ajout ou la suppression de *breakpoints*, en plus des traitements décrits ci-dessus.

La JPDA ne définissant pas d'événement pouvant être transmis par le débogueur à la machine virtuelle ou à la bibliothèque JDWP, la gestion des paquets reçus ne traite pas le cas des paquets contenant des événements.

Chapitre VI

TESTS ET INTÉGRATION

L'interface de débogage a été implantée au sein SableVM. Cette machine virtuelle utilise la bibliothèque JDWP afin de communiquer avec le reste des éléments de l'architecture de débogage Java proposée par Sun Microsystems. Tout au long du développement de ces deux entités, nous avons tenu à effectuer des tests sur chaque élément développé.

Ce chapitre présente les tests de chaque élément, ainsi que les résultats obtenus. La première partie traite des expériences effectuées sur SableVM, plus précisément celles relatives à l'interface de débogage implantée. Le seconde partie présente le test de la bibliothèque JDWP, ainsi que l'intégration de l'ensemble au sein d'un environnement de développement Java. Enfin, la dernière partie relate des résultats obtenus, suivis de nos recommandations quant aux travaux futurs.

6.1 Test de l'interface de débogage

6.1.1 Tests unitaires

Tout au long de l'implantation de la JVMDI au sein de SableVM, nous avons tenu à valider les fonctions au fur et à mesure de leur rédaction. La JVMDI est à la base de la communication avec la machine virtuelle, et nous avons planifié d'utiliser SableVM afin d'effectuer les différents tests de la bibliothèque JDWP. Il nous fallait donc une certaine assurance quant à la machine virtuelle utilisée.

Chaque fonction a été testée dès sa mise en place au sein de SableVM. Aucun jeu de tests standard n'étant disponible, il a donc fallu rédiger un tel jeu, ainsi que définir les résultats attendus pour chacune des fonctions testées.

Afin de pouvoir effectuer ces tests, nous avons du développer deux entités : (1) une classe Java exécutée par SableVM, et manipulant des objets relatifs aux fonctions testées, (2) un client JVMDI léger, récupérant la référence vers la JVMDI de SableVM et effectuant les appels aux fonctions expérimentées. Par exemple, afin de tester la fonction de récupération des processus existants, GetAllThreads, nous avons rédigé une classe Java créant un nombre défini de processus légers. Le client JVMDI effectue ensuite l'appel à cette fonction et affiche le résultat obtenu. Il nous est donc possible de comparer aisément le résultat obtenu au résultat attendu.

La même classe a été utilisée pour les tests de toutes les fonctions de la JVMDI, les portions de codes inutilisées étant en commentaires. Cette classe et le client JVMDI créé ont été ajoutés au code source distribué. De cette manière, les utilisateurs sont libres d'effectuer eux-mêmes ces tests, ou d'avoir une base afin de confectionner de nouveaux tests.

L'implantation de certaines fonctions a demandé certains changements aux mécanismes internes de SableVM. En plus de tester les fonctions de la JVMDI, nous avons également tenu à tester les modifications apportées à SableVM. La validation de tels changements est plus délicate que le test d'une simple fonction. Souvent, le comportement de l'application reste identique dans les cas ordinaires, après modifications. Seuls les cas spéciaux laissent transparaître des différences ou erreurs. C'est donc en exécutant des applications plus lourdes, supportées par SableVM avant notre intervention, que la validation de ces changements a été faite.

Conscients que les tests effectués ne sont pas exhaustifs, SableVM met à disposition des utilisateurs un système de soumission de bogue(s) [1].

---

[1] http ://sablevm.org/bugs.html

6.1.2 Tests bout en bout

Une fois les fonctions testées individuellement, nous avons élaboré des scénarii de tests plus élaborés, en tenant compte des fonctions disponibles au moment de l'élaboration.

En reprenant l'exemple présenté dans la section précédente, nous avons testé les fonctionnalités relatives aux processus en suivant le scénario suivant : après avoir récupéré tous les processus existants, dont certains ont été suspendus durant l'exécution de l'application Java, nous avons suspendu ceux en activité, suspendu ceux déjà suspendus (afin de tester le comportement en cas d'erreur), demandé le statut de chacun des processus, puis relancé leur exécution. Ce cas de test permet l'utilisation d'un éventail plus large de fonctions dans une même session de débogage. nous avons donc pu évaluer les interactions possibles entre les fonctions de la JVMDI.

Cette étape nous a donné une ébauche de ce qu'allait être le développement de la bibliothèque JDWP, et nous a permis d'anticiper certains aspects qui ne nous étaient pas apparus lors de l'étude préliminaire. Par exemple, c'est à ce moment que nous avons constaté la nécessité de conserver les objets manipulés par la bibliothèque.

Le but de notre étude étant d'avoir un ensemble machine virtuelle/bibliothèque JDWP fonctionnel et robuste le plus rapidement possible, nous avons concentré nos efforts sur les fonctions essentielles de la JVMDI. Nous n'avons donc pas implanté l'ensemble des fonctions existantes avant de débuter le développement de la bibliothèque JDWP. Aujourd'hui, la JVMDI est implantée à 90% dans notre machine virtuelle Java : à quelques exceptions près, seules les fonctionnalités relatives à la manipulation des *monitors* Java et des *watchpoints* n'ont pas été développées.

6.2 Test de la bibliothèque JDWP

Le développement et les tests de la bibliothèque JDWP n'ont pas été réalisés de la même manière que ceux de l'implantation de la JVMDI. Afin de décider quel(s) mé-

canisme(s) de transport choisir et quelles commandes implanter en premier, nous avons opté pour l'intégration de nos travaux au sein d'un environnement de développement. Ainsi, le mécanisme de transport choisi est celui utilisé par cet environnement, et les commandes implantées sont celles appelées par le débogueur.

Cette manière de procéder présente l'avantage de requérir moins de tests supplémentaires à la fin du développement, puisque l'outil est déjà fonctionnel avec le débogueur choisi.

Dans cette partie, nous présentons tout d'abord comment nous avons choisi l'environnement de développement auquel intégrer nos travaux. Ensuite, la seconde section détaille comment nous avons intégré nos travaux à ce dernier. Enfin, nous présentons les tests complémentaires effectués à la fin du développement de la bibliothèque JDWP.

6.2.1 Choix de l'environnement de développement

La validation de nos travaux ne peut être effectuée que par l'utilisation effective de SableVM et de la bibliothèque JDWP au sein d'un environnement de développement standard. Cette utilisation permet de s'assurer du bon fonctionnement des éléments développés. De plus, l'intégration dans un environnement standard garantit un respect des normes en vigueur.

Pour ce faire, nous devons choisir à quel environnement de développement intégrer nos travaux. Nous recensons tout d'abord les critères importants à nos yeux, avant de présenter l'outil de notre choix, respectant ces critères.

6.2.1.1 Critères de choix

Lors du choix du débogueur à utiliser plusieurs critères ont été pris en compte. Nous détaillons ici quelles ont été les contraintes que nous avons imposées dans le choix de l'outil à utiliser.

Le respect des normes est un critère fondamental. En effet, il nous serait impos-

sible de valider nos travaux avec un outil utilisant des protocoles propriétaires afin de communiquer avec les autres entités de l'architecture de débogage. On recherche donc un outil se conformant aux spécifications de la JPDA. Bien que de nombreux éditeurs clament leur compatibilité avec les standards existants, aucune certification n'existe quant à la conformité aux normes dictées par la JPDA. L'assurance qu'un débogueur soit conforme aux standards est en partie donnée par la diversité des applications l'ayant utilisé. Dans notre cas, si un débogueur a été employé avec succès dans les phases de développement d'applications Java reposant sur des machines virtuelles différentes et créées par des éditeurs indépendants, le risque de non conformité est faible.

De nombreux types de débogueurs existent. Certains spécifiques à un langage de programmation, d'autres plus génériques. L'outil que nous recherchons doit permettre de déboguer du code Java, mais doit également être doté d'une interface graphique. Une fois la mise en place des mécanismes de communication, alors que les fonctionnalités offertes ne sont pas toutes implantées, nous devons être capable de visualiser rapidement, voire en temps réel, le comportement de notre outil. Cette visualisation est effectuée par le biais de divers moyens d'impression. L'utilisation d'un outil comportant une interface graphique évoluée facilite cette impression, évitant d'avoir recours à l'utilisation des routines d'accès aux fichiers.

SableVM est un outil libre, sous licence LGPL (Lesser General Public License, publiée par GNU). Cette licence est très permissive et autorise la redistribution du code source de SableVM. Ainsi, notre machine virtuelle peut être utilisée à des fins d'enseignement ou de recherche, sans contraintes contractuelles lourdes. Il a donc été important à nos yeux que l'outil utilisé offre une liberté similaire aux utilisateurs, indépendamment de la licence choisie. L'accès au code source du débogueur nous permet également d'anticiper les attentes du débogueur et donc, parfois, clarifier des points de la spécification, quelques fois confuse. D'autre part, cette mise à disposition du code permet de s'assurer de la validité des algorithmes utilisés ou de vérifier que les mécanismes en place sont bien conformes aux normes.

Enfin, la visibilité a été un critère déterminant dans notre décision. L'outil de notre choix doit être largement répandu au sein de la communauté d'utilisateurs, la compatibilité avec un outil populaire offrant plus de crédibilité à nos travaux. Bien entendu, nos réalisations étant conformes aux normes, l'utilisateur est libre de recourir à n'importe quel autre débogueur, également conforme aux normes, afin de développer ses applications Java.

6.2.1.2 Outil choisi

Au regard de toutes les conditions citées précédemment, nous avons opté pour l'utilisation d'Eclipse[2].

Eclipse est un environnement de développement ouvert, développé en Java. Bien qu'il ne soit pas spécifique à Java, Eclipse offre toutes les fonctionnalités relatives au débogage et à l'exécution du code Java. L'outil est doté d'une interface graphique intuitive : la décomposition de cette interface en fenêtres, regroupées en perspectives, permet à l'utilisateur de personnaliser son environnement de travail.

L'association au projet Eclipse de grands noms de l'édition de logiciels tels qu'IBM, HP, Borland ou Rational Software offre une certaine garantie quant à la pérennité de l'outil et une crédibilité incontestable. De plus, cette association permet à Eclipse de jouir d'une visibilité remarquable au sein de la communauté de développeurs de logiciels.

Eclipse est distribué sous une licence particulière : l'Eclipse Public License (EPL). Elle permet la redistribution gratuite du code source à travers le monde, pour les organismes sans but lucratif. Cette licence est donc conforme à la politique de liberté adoptée par SableVM.

La réalisation de notre étude ayant été échelonnée sur plus d'une année, plusieurs versions d'Eclipse ont été publiées durant ce laps de temps. Ainsi, nous avons eu l'opportunité d'intégrer nos travaux aux environnements Eclipse 2.1, 3.0 et 3.1. Les évolutions

---

[2]http ://www.eclipse.org

apportées à cet environnement de développement durant les derniers mois n'ont jamais affecté les mécanismes de communications entre débogueur et machine virtuelle Java, gage que les standards ont été respectés de part et d'autre.

Avant de présenter plus en détails dans la section suivante l'intégration de nos travaux à l'environnement Eclipse, la figure 6.1 vous propose une vue d'ensemble des éléments de l'architecture de débogage ainsi formée.

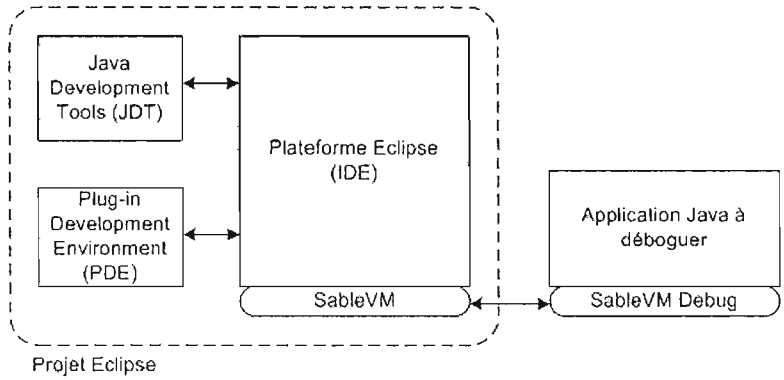
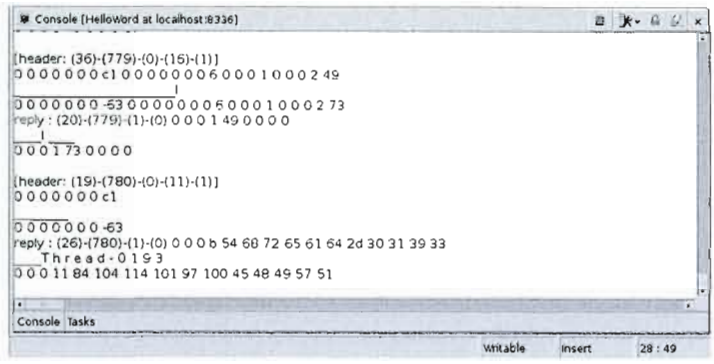


Fig. 6.1 Architecture de débogage Eclipse/SableVM

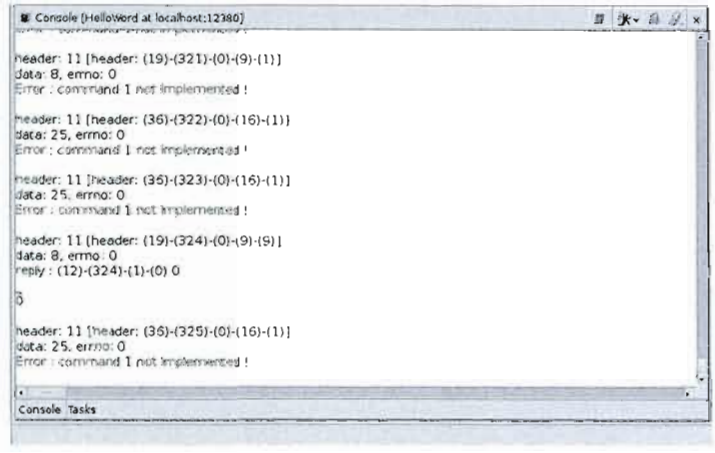
6.2.2 Intégration à Eclipse

Le choix du mécanisme de transport est indiqué par le débogueur dans la ligne de commande de lancement de la machine virtuelle (voir l'exemple présenté en figure 3.7). La mise en place de ce mécanisme est propre au langage de programmation utilisé. En langage C, de nombreuses routines permettent l'établissement de connexion distante de type socket. Le test associé ne consiste qu'à vérifier que cette connexion est établie.

Une fois les outils connectés, nous avons rédigé un ensemble de fonctions permettant l'affichage des paquets JDWP en transit sur le réseau (figure 6.2). De cette manière, en profitant de l'interface graphique du débogueur, il nous a été simple de déterminer quelles commandes implanter, ainsi que les données envoyées comme paramètres de celles-ci.



(a) Affichage de commandes implantées



(b) Affichage de commandes non implantées

Fig. 6.2 Affichage de paquets JDWP reçus par la bibliothèque JDWP

En plus de faciliter le développement, la création de ces fonctions d'affichage de paquets nous a donné les bases de l'encodage et du décodage des données dans les paquets JDWP.

Une fois les commandes implantées, les fonctionnalités relatives ont été testées durant quelques sessions de débogage (figure 6.3). Nous avons donc créé quelques classes ou utilisé des applications Java développées dans le cadre de travaux précédents afin d'utiliser l'éventail le plus large possible de fonctionnalités de débogage.

Conscients que les applications de notre cru ne comportent peut-être pas de dif-

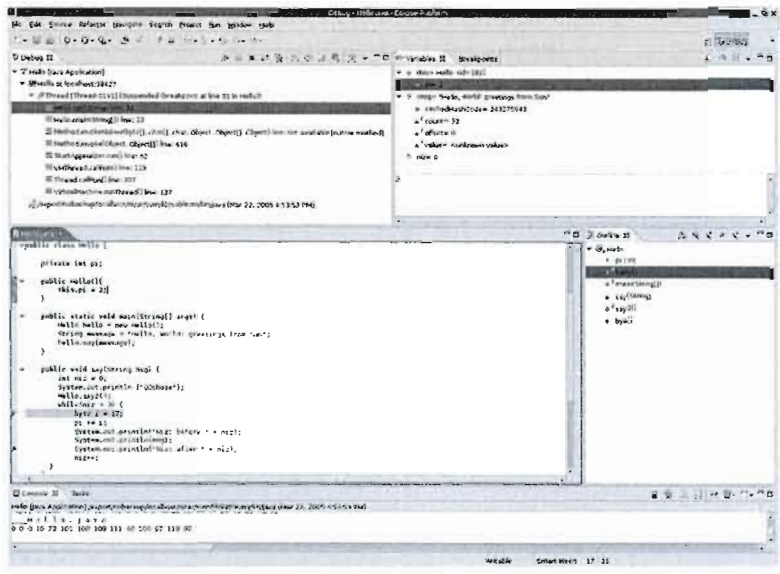


Fig. 6.3 Perspective de débogage avec SableVM sous Eclipse

ficultés suffisantes pour tester les fonctionnalités de débogage et les structures internes de SableVM, nous avons tenté le débogage d'une application plus conséquente : Ant. Ant, édité par Apache[3], est un outil Java de construction d'application, au même titre que les outils *make* ou *jam*. Ainsi, l'exécution de la version 1.6.3 de Ant nous a permis de mettre en exergue des erreurs quant à la gestion des exceptions dans notre code de débogage.

De la même manière que pour l'implantation de la JVMDI au sein de SableVM, la bibliothèque JDWP ne traite pas toutes les commandes recensées dans la spécification. Seules celles étant effectivement appelées par le débogueur utilisé ont été développées et testées. Ainsi, lors de la publication finale de nos travaux, 95% des commandes sont traitées. Seules celles relatives à la manipulation de monitors Java et celles faisant appel à de la reflection (du package java.lang.reflect) manquent à notre bibliothèque.

---

[3]http ://ant.apache.org

6.2.3 Tests complémentaires

Une fois convaincus de la robustesse de nos travaux, nous avons décidé de les soumettre à des tests complémentaires. Ces tests ont pour but la validation de certaines propriétés des éléments que nous avons développés.

6.2.3.1 Modularité

L'un des points forts de la JPDA et la modularité offerte. Cette modularité permet, en théorie, d'utiliser, aux différents niveaux de l'architecture, n'importe quel outil conforme aux spécifications. Afin de nous assurer qu'une telle propriété est conservée, nous avons tenté de lancer une session de débogage en introduisant un élément autre que ceux que nous avons développés.

Nous avons été capables de compiler et débuter une session de débogage en ayant recours à Eclipse comme débogueur, notre bibliothèque JDWP comme interface et la machine virtuelle de Sun Microsystems[4] pour l'exécution de l'application Java à déboguer. Plus précisément, nous avons utilisé le J2SE SDK (Java 2 Platform Standard Edition Software Development Kit) 1.4.2.08, incluant la machine virtuelle HotSpot. Cette machine virtuelle vient prendre la place de SableVM dans l'architecture de débogage.

Bien que les tests préliminaires nous aient confirmé notre conformité aux spécifications de la JPDA, certains problèmes inhérents à la gestion des processus nous ont empêché de pouvoir déboguer entièrement une application avec la machine virtuelle HotSpot. Étant relativement difficile d'obtenir des informations quant à la gestion ou aux structures utilisées par les machines virtuelles commerciales, et considérant le manque de temps relatif, nous n'avons pas poursuivi notre investigation sur les raisons de ce dysfonctionnement.

---

[4]http .//java.sun.com

6.2.3.2 Portabilité

Tous nos travaux ont été réalisé en langage C, respectant la norme ANSI. Le respect de cette norme garantit une portabilité du code écrit, indépendamment de la plateforme sur laquelle il est compilé et exécuté.

Régulièrement durant le développement de SableVM et de la bibliothèque JDWP, nous avons compilé et exécute notre code sur des architectures diverses. Ces opérations nous ont permis d'attester la portabilité du code produit. Jusqu'à aujourd'hui, nous avons effectué des tests sur Linux/PowerPC, Linux/Intel et Linux/Sparc sans que le comportement de notre application ne varie.

6.3 Résultats et travaux futurs

6.3.1 Résultats

Le résultat de nos travaux a placé SableVM au premier plan de la scène du développement de logiciel libre, faisant de notre machine virtuelle Java la première offrant des fonctionnalités de débogage conformes aux normes. SableVM est, de fait, la première machine virtuelle Java libre capable d'intégrer des environnements de développements standards. Cette nouvelle étape permet aux utilisateurs de s'affranchir des contraintes légales liées à l'utilisation de Java, et offre aux adeptes de SableVM un éventail d'outils compatibles aussi vaste que ceux offerts par les machines virtuelles Java commerciales.

De plus, nous espérons que cette étude servira de support aux développeurs d'autres machines virtuelles Java. Nous pensons avoir mis en évidence les différents points à prendre en compte lors de l'intégration d'attributs de débogage au sein d'une machine virtuelle Java, indépendamment des spécificités de SableVM.

Avant l'achèvement du développement, nous avons procédé à la publication d'une version expérimentale de notre interface de débogage. Ainsi, dès le mois d'avril 2005, la publication de SableVM et de la bibliothèque JDWP a été annoncée sur la liste de

diffusion des utilisateurs de SableVM, et une section relative à l'interface de débogage a été ajoutée au site http ://sablevm.org. La première version publiée ne possède pas toutes les fonctionnalités disponibles aujourd'hui, mais est assez robuste pour le débogage d'une application Java simple. Cette publication met à disposition SableVM à un grand nombre d'utilisateurs, multipliant, de fait, le nombre de tests effectués, et nous aidant dans la validation de nos travaux.

Le script permettant l'accès à cette version et à la mise en place d'un environnement libre et complet de débogage Java est présenté en annexe A.

Depuis cette première publication, aucune autre version, plus aboutie, n'a été présentée à la communauté d'utilisateurs. En effet, après l'ajout de nouvelles fonctionnalités et la corrections de certains bogues, nos travaux seront intégrés au tronc principal de développement de SableVM, les résultats des tests étant jugés satisfaisants. Une version complète de la machine virtuelle sera publiée dès septembre 2005.

La publication de nos travaux suscite aujourd'hui un grand intérêt auprès des utilisateurs. En juillet 2005, nous avons reçu, par exemple, des propositions émanant d'étudiants suisses de l'Interstaatliche Hochshule Für Technik Buchs, désireux de participer au développement de l'interface de débogage.

6.3.2 Travaux futurs

Bien qu'étant relativement robustes et fonctionnels, on pourrait toutefois apporter des améliorations aux éléments développés. Nous présentons ici nos propositions de maintenance corrective et évolutive possibles.

6.3.2.1 Optimisation

Lors du développement de la bibliothèque JDWP, nous avons, par commodité, opté pour la réutilisation de structures de données déjà présentes dans SableVM. Cette réutilisation offrait l'avantage d'un mécanisme de génération de code et une fiabilité de

l'implantation déjà vérifiée.

Bien que les performances offertes et l'aisance d'utilisation sont grandes, il est envisageable que les structures de données choisies ne conviennent pas à un utilisateur. Nous avons privilégié des structures au temps amorti d'accès faible ($\log(n)$ par opération pour les splay trees), mais on aurait pu opter pour la minimisation du pire des cas, par exemple.

6.3.2.2 Achèvement de l'implantation

Comme décrit dans les sections de tests, nos travaux sont fonctionnels, mais ne comportent pas toutes les fonctionnalités existantes. Ainsi, aussi bien dans l'implantation de la JVMDI que dans le traitement des commandes reçus par la bibliothèque JDWP, nous recommandons le développement des fonctions manquantes, afin d'offrir aux utilisateurs des possibilités comparables à tous les outils commerciaux existants.

De plus, la génération de certains événements, non utilisés, n'est pas mise en place au sein de SableVM. Ainsi, les événements relatifs à la suppression des fenêtres d'exécution de la pile d'un processus ne sont pas notifiés au client JVMDI. De plus, le client n'est pas averti de ceux associés à la modification de la valeur d'un champ. Ces derniers étant utilisés uniquement pour la mise en place de *watchpoints*, nous ne les avons pas insérés dans les mécanismes de notre machine virtuelle.

Une liste des fonctionnalités manquantes est disponible sur le site de SableVM. de plus, les modifications concernant chaque nouvelle version sont publiées d'une part sur les listes de diffusion des utilisateurs de SableVM, mais également sur le site Internet.

6.3.2.3 Conformité aux nouvelles normes

Depuis quelques mois, alors que nos travaux étaient déjà avancés, Sun Microsystems a publié une nouvelle version des normes relatives à Java. La version 1.5.0 du J2SE présente des modifications majeures dans l'approche du débogage, entre autres.

Alors que les fonctionnalités relatives au débogage et au profilage du code Java étaient jusqu'ici séparées en deux interfaces, JVMDI et JVMPI (Java Virtual Machine Profiler Interface), elles sont désormais réunies en une seule : la JVMTI (Java Virtual Machine Tool Interface).

En plus de cette réunification, la nouvelle norme propose des formats de fonctions, ainsi qu'un mécanisme de gestion des événements différents. Ajoutez à cela que certaines fonctions sont désormais définies comme optionnelles au sein de la JVMDI. La plupart des fonctionnalités de débogage restent toutefois identiques. Les modifications à apporter afin de se conformer aux nouvelles normes restent minimes, ne consistant qu'en la modification des paramètres d'appel aux fonctions de la JVMDI, pour la plus grande part.

6.3.2.4 Suggestions d'amélioration

L'une des fonctionnalités remarquables des machines virtuelles Java est son ramasse-miettes. Cependant, dans la JPDA, aucun événement ne fait référence au recyclage de la mémoire.

La bibliothèque JDWP procède, durant toute la durée de la session de débogage à la collecte des objets manipulés. De cette collecte découle un amoncellement d'objets, et une utilisation croissante de l'espace mémoire. Seuls les objets représentant les processus et les fenêtres d'exécution associées sont libérés dès la réception d'un événement de fin de processus. Aucune autre libération de la mémoire n'est effectuée par la bibliothèque JDWP avant la fin de la session. On peut décemment douter que tous les objets manipulés durant une session de débogage perdureront durant toute cette session.

Nous proposons de mettre en place un événement généré lors du recyclage d'un objet. Le client JVMDI recevant un tel événement sait alors qu'il peut procéder à la libération des structures de données contenant l'objet identifié.

Cette fonctionnalité n'étant pas directement liée au débogage au même titre que

la mise en place des *breakpoints*, il serait possible de la spécifier comme optionnelle, si elle venait à être ajoutée à la JPDA.

Chapitre VII

TRAVAUX RELIÉS

Le débogage tient une place prépondérante dans le cycle de développement d'un logiciel. Les enjeux financiers importants liés au besoin croissant de logiciels fiables et performants stimulent la recherche dans ce domaine. Notre étude porte sur l'ajout de fonctionnalités relatives au débogage classique au sein d'une machine virtuelle Java. Cependant, d'autres paradigmes de débogage existent.

Bien que nous ne nous soyons pas inspirés d'autres paradigmes lors de notre réalisation, il nous paraît essentiel de ne pas réduire le débogage à son paradigme le plus répandu. Nous proposons donc ici la présentation de certaines des avancées effectuées dans le débogage, espérant que cette étude puisse inspirer les auteurs de travaux similaires relatifs à d'autres paradigmes. Nous exposons dans ce chapitre des concepts permettant d'optimiser le temps de recherche d'erreur au sein d'une application. Étant données l'étendue et la diversité des recherches, cette présentation n'est pas exhaustive mais donne un aperçu de certaines avancées dans le domaine.

A l'achèvement de nos travaux, aucune architecture de débogage Java libre n'était disponible. Les seules machines virtuelles Java possédant des attributs relatifs au débogage étant à but commercial, l'accès à leur code source est restreint par l'acceptation des termes des licences associées. De telles licences n'autorisent pas la publication de travaux connexes, et n'auraient donc pas permis la parution de ce mémoire. Il nous est donc impossible de comparer ici les travaux réalisés dans notre étude et leurs équivalents

commerciaux.

Ce chapitre présente principalement trois paradigmes de débogage. Dans chacun des cas, nous décrivons d'abord la méthode de débogage avant d'en présenter une application concrète. La première partie de ce chapitre traite du delta débogage. Puis, nous développerons, dans la seconde partie, les avantages de la vue en coupe d'un programme. Enfin, la troisième partie aborde les différents aspects du débogage bidirectionnel.

7.1 Delta débogage

Lorsqu'un rapport de bogue est soumis, la première étape de traitement est de tenter de reproduire ce bogue. Si la reproduction aboutit, la seconde étape consiste à isoler le facteur de défaillance : on minimise le cas de test.

Lorsqu'en juillet 1999, la compagnie Netscape décida de publier librement le code source de son navigateur Netscape Communicator, le projet Mozilla a vu le jour. Mozilla avait pour but d'inciter un plus grand nombre d'utilisateurs à s'impliquer dans le développement du navigateur. Les bases de données de l'organisation contenaient à l'époque approximativement 370 rapports de bogues ouverts[1]. Les développeurs étaient alors confrontés à un besoin crucial d'automatisation dans leur tâche de minimalisation des cas de test. Les premières recherches dans cette direction furent entreprises.

Cette section présente le delta débogage, créé afin de répondre au besoin d'automatisation lors de la recherche de cas de test minimaux. La première partie de cette section présente l'algorithme de delta débogage. Cette présentation sera suivie d'exemples d'outils existants, permettant la localisation d'erreurs grâce à cet algorithme.

---

[1] le statut d'un bogue est ouvert s'il a été soumis, mais qu'aucun traitement ne lui a été appliqué. Généralement, on utilise les statuts *ouvert*, *assigné*, *réouvert*, *résolu*, *vérifié* et *fermé*.

7.1.1 Présentation de l'algorithme

Aussi connu sous le nom d'*automated debugging*, le delta débogage est une approche de débogage dérivée de l'*automated testing*. Principalement basé sur les travaux d'Andreas Zeller (Zeller et Hildebrandt, 2002), ce concept consiste à générer et effectuer automatiquement des tests sur un programme, séparant les tests reproduisant un bogue donné des autres. Cette technique repose sur l'hypothèse d'existence de cas minimaux de test et permet d'isoler les cas minimaux causant la défaillance.

Lorsque le comportement anormal est observé, on récupère les paramètres d'entrée du programme. On définit un cas de test comme étant un ensemble de modifications apportées à l'entrée d'un programme. Aucune contrainte n'est imposée quant à la nature des modifications, celles-ci pouvant être un changement apporté au code source, une différence dans les paramètres d'entrée d'un programme...

On recherche les cas de test minimaux selon un procédé s'apparentant à une dichotomie : si c, un cas de test produisant la défaillance, ne contient qu'une seule modification, alors c est minimal par définition. Sinon, on le divise en deux sous-ensembles c_1 et c_2 équivalents. Pour ce faire, on suppose l'utilisation d'une procédure de tests capable de générer un cas de test en apportant les modifications à une entrée du programme, soumettre ce cas au programme et renvoyer le résultat obtenu. Ce résultat peut être 1 si le programme s'est exécuté normalement, 0 si le bogue a été reproduit ou ? si le résultat est indéterminé (le programme a terminé son exécution anormalement, par exemple). On teste alors chacun des deux sous-ensembles. Trois possibilités s'offrent alors :

- $test(c_1) = 0$, dans quel cas on peut réduire l'ensemble à c_1,
- $test(c_2) = 0$, dans quel cas on peut réduire l'ensemble à c_2,
- les deux tests ont réussi ou restent indéterminés, dans quel cas ni c_1, ni c_2 ne peuvent être choisis comme réduction de l'ensemble. Dans ce cas, on doit trouver une autre méthode de division de c.

Plus formellement, l'algorithme du delta débogage est présenté en figure 7.1.

L'algorithme de delta débogage minimisant $ddmin(c)$ est :
$ddmin(c) = ddmin_2(c, 2)$ où

$$ddmin_2(c, n) = \begin{cases} ddmin_2(c_i, 2) & \text{si } \exists i : test(c_i) = 0 \\ ddmin_2(\bar{c_i}, max(n-1, 2)) & \text{si } \exists i : test(\bar{c_i}) = 0 \\ ddmin_2(c, min(|c|, 2n)) & \text{si } n < |c| \\ c & \text{sinon} \end{cases}$$

avec $c_1, \ldots, c_n \subseteq c$ tels que $\cup c_i = c$, les c_i deux à deux disjoints, $\forall c_i(|c_i| \approx |c|/n)$, et $\bar{c_i} = c - c_i$.

L'invariant de boucle de $ddmin_2$ est $test(c) = 0 \wedge n \leq |c|$.

Fig. 7.1 Algorithme de delta débogage minimisant (Zeller et Hildebrandt, 2002).

Dans le meilleur cas, l'algorithme effectue moins de $2\log_2(|c|)$ tests pour déterminer les cas de test minimaux. Le meilleur cas se produit lorsque qu'une seule modification Δ_i provoque le bogue, et que tous les cas des test contenant cette modification provoquent une défaillance. Dans le pire cas, les cas de test minimaux sont produits après $3|c| + |c|^2$ tests (Cleve et Zeller, 2005).

Cette première version de l'algorithme a, par la suite, été étendue afin de pouvoir répondre à de nouvelles contraintes. Ainsi, l'utilisation d'outils comme DejaVu d'IBM (Deterministic Java Replay Utility[2]) permet d'appliquer ce processus aux programmes multi-processus en conservant le même ordonnancement des processus (Choi et Zeller, 2002) lors de chaque exécution. D'autres études ont permis d'apporter des informations complémentaires aux cas de test minimaux, comme la localisation temporelle et spatiale des bogues (Zeller, 2002).

7.1.2 Recherche d'erreur grâce au delta débogage

Plusieurs implémentations de cet algorithme sont disponibles aujourd'hui. Les outils développés sont de deux formes : applications autonomes, ou extensions d'outils

[2] http ://www.research.ibm.com/dejavu/

existants. AskIgor (http ://www.askigor.org), par exemple, est l'interface web d'un delta débogueur auquel on peut soumettre des programmes écrits en C/C++ sur plateforme x86. Un « diagnostic » du programme est alors proposé en ligne, indiquant les résultats obtenus (cas de test minimaux) suite à l'application du delta débogage sur le programme.

Des extensions à l'environnement de développement Eclipse sont également proposées (Bouillon, Burger et Zeller, 2003), permettant aux utilisateurs d'intégrer le delta débogage aux possibilités offertes par leur outil de développement habituel.

7.2 Vue en coupe d'un programme

Au début des années 1980, Weiser introduisit un nouveau concept afin de faciliter la recherche d'erreurs dans un programme séquentiel : la coupe de programme ou *program slicing* (Weiser, 1981). Son idée première était de ne montrer au programmeur que les informations pertinentes à sa recherche, et donc de faire disparaître le reste du code. On parle alors de la coupe d'un programme. Cette coupe possède deux propriétés importantes : (1) elle est obtenue en supprimant de l'information, (2) son comportement est identique à celui du programme dont elle est issue.

Weiser a ouvert la voie à de nombreuses initiatives reprenant et adaptant le même principe. Depuis la publication originale de cet algorithme, des centaines d'articles ont été publiés sur le sujet, et des dizaines de techniques dérivées ont vu le jour (Xu et al., 2005). On ne présente pas ici une liste exhaustive des recherches effectuées, simplement les trois principales tendances. Nous commençons donc par décrire la coupe statique introduite par Weiser. La partie suivante présente l'extension apportée à cette méthode afin de répondre aux contraintes de la programmation orientée objet. Puis nous traitons de la coupe dynamique, utilisant des données recueillies lors de l'exécution du programme. La dernière partie présente les outils existants permettant la construction de ces coupes, ainsi que quelques utilisations qui en sont faites.

7.2.1 Coupe statique

La coupe statique, ou *static slicing*, est une technique permettant de faire apparaître les instructions pouvant avoir une influence sur une collection de variables données. Elle est dite statique car elle utilise uniquement les informations disponibles dans le code source.

L'élaboration d'une coupe commence par la construction d'un graphe orienté représentant les dépendances entre les diverses instructions et les variables présentes dans le corps d'une procédure. Ce graphe est ensuite élagué pour ne conserver que les nœuds ayant une influence direct sur un ensemble d'instructions et de variables donné. On spécifie cet ensemble grâce à un critère de coupe. Formellement, ce critère de coupe est un couple $C = (l, V)$, avec l l'emplacement de l'instruction dans le code source et V l'ensemble des variables à observer. On peut alors définir l'ensemble des variables et instructions qui ont eu une incidence sur les variables de V (coupe arrière) ou celles influencées par les variables de V (coupe avant).

Afin de construire ces coupes, on détermine les influences entre les différentes instructions d'un programme : on construit un graphe de dépendance du programme (PDG) pour chaque fonction à étudier. Ce graphe possède un nœud pour chaque instruction simple du programme (assignation, lecture, écriture...), et un nœud pour chaque prédicat de contrôle (expression à l'intérieur d'une condition *if-then-else*, *while*...). Chaque arc représente une dépendance de contrôle ou de donnée entre deux nœuds. Plus précisément, une dépendance de données est l'utilisation par un nœud j d'une variable définie à un nœud i alors qu'il existe un chemin d'exécution allant de i à j. Une dépendance de contrôle indique qu'un nœud j peut ou non être atteint selon le booléen résultant de l'exécution du nœud i précédent. En n'utilisant que l'une ou l'autre des dépendances, on peut également construire les graphes des flots de contrôle (*controlflow graph*, CFG) et flots de données (*dataflow graph*, DFG) du programme. En supposant qu'à partir du PDG d'un programme, le CFG résultant comporte n nœuds et e arêtes, l'élaboration des coupes a un coût, dans le pire cas, de l'ordre de $O(v * (n + e))$, où v représente le

nombre de variables à observer.

En étendant la notion de PDG aux dépendance d'un système, Horwitz, Reps et Binkeley ont permis la création de coupes statiques inter-procédurales (Horwitz, Reps et Binkeley, 1988) en définissant le graphe de dépendance du système (*system dependence graph*, SDG). La principale innovation de ce type de structure est l'incorporation des appels de fonctions, ajoutant un nœud pour chaque paramètre d'une fonction appelée, ainsi que pour chaque valeur retournée en sortie. Des variations de ce graphe ont également été utilisées dans l'optimisation de compilateurs.

7.2.2 Coupe orientée objet

Dans le cas de la programmation orientée objet, les méthodes sont contenues dans des objets. On ne peut donc considérer un programme comme une suite séquentielle d'instructions. Si, pour chaque méthode, on peut produire le SDG correspondant, toutes les particularités de la programmation orientée objet ne sont pas représentées. Les classes et leurs attributs, les objets, la notion d'héritage et de polymorphisme, entre autres, sont des concepts qui n'ont pas été pris en compte lors de l'élaboration de l'algorithme original de coupe.

Larsen et Harrold ont proposé une méthode de coupe permettant de répondre aux spécificités de la programmation orientée objet (Larsen et Harrold, 1996). Leur principale innovation repose dans la réutilisation des composantes lors de la construction d'un graphe : la représentation d'une classe, un graphe, peut être incorporée dans une classe ou une application qui utiliserait cette classe. Ce graphe de dépendance de classe (*class dependence graph*, ClDG) comporte les informations sur les dépendances de données et de contrôle d'une classe sans connaissance de son environnement. L'ajout de nœuds formels pour les entrées, les références modifiées, ainsi que pour les variables globales références dans le corps d'une méthode, facilite l'insertion du graphe dans la representation d'une classe dérivée, par exemple. On ajouterait alors des connexions à ces nœuds ainsi que de nouveaux PDG pour ses propres méthodes.

Le polymorphisme, propriété permettant au type d'un objet d'etre connu lors de l'exécution uniquement, est supporté grâce à l'insertion de nœuds de choix dans le SDG. Ces nœuds permettent d'effectuer des connexions avec tous les types éventuels lorsque plusieurs types sont possibles pour un objet.

Finalement, la construction du SDG d'un programme complet n'est que la création de connexions appropriées entre les différents ClDG. Cette représentation implique une légère modification du critère de coupe $C = (p, x)$, ou p représente une instruction et x représente soit une variable, soit un appel à une méthode. Dans les deux cas, x doit être présent dans l'instruction p. La construction des coupes est alors identique au cas statique présenté précédemment, bien que les graphes, dans le cas orienté objet, puissent comporter un nombre de nœuds beaucoup plus important.

Par la suite, cette technique a été étendue afin de pouvoir correspondre aux spécificités du langage Java (Chen et Xu, 2001), ou à celles des applications distribuées (Zhao, 1999).

7.2.3 Coupe dynamique

Les coupes dynamiques, contrairement aux coupes statiques, requièrent au moins une exécution du programme. Durant cette exécution, on procède à la récolte d'informations. La coupe dynamique tire ensuite partie de ces informations pour affiner l'élagage des données présentées à l'utilisateur. Ces coupes sont dites dynamiques du fait de l'utilisation de données recueillies lors de l'exécution.

Si dans le cas de la coupe statique, le programmeur peut, pour un emplacement donné dans le code, choisir entre les deux coupes possibles (avant et arrière), dans le cas de la coupe dynamique, seule la coupe avant est disponible. Celle-ci utilise également un critère de coupe C, définit comme un triplet $C = (E, I^i, V)$, où E définit les valeurs d'entrée du programme, I^i une valeur de l'historique d'exécution et V l'ensemble des variables à observer. L'historique d'exécution est un ensemble $EH = \{I^1, \ldots, I^i, \ldots, I^n\}$, où chaque I^i représente un emplacement I dans le code avant le i^{eme} pas d'exécution.

Dans une boucle, par exemple, cette notation permet de spécifier à partir de quelle itération commencer la construction de la coupe. De fait, il devient plus aisé d'examiner le comportement des branchements conditionnels d'un programme.

Agrawal et Horgan ont proposé quatre approches dérivées (Agrawal et Horgan, 1990) permettant la construction de ces coupes. Ils ont notamment introduit la notion de graphe de dépendance dynamique (*Dynamic Dependence Graph*, DDG), comportant un nœud pour chaque occurrence d'une instruction, ainsi qu'une technique de réduction de ce graphe. Le calcul des coupes est alors trivial une fois le graphe construit. Pour un critère donné, on obtient une version élaguée de la coupe statique, réduisant le montant d'information présenté à l'utilisateur lors du débogage. Ces techniques restent néanmoins gourmandes en espace mémoire. Des études ont alors suggère des méthodes pour réduire le coût des coupes dynamiques (Zhang et Gupta, 2004), voire un compromis entre la taille des graphes générés et la précision offerte (Zhang, Gupta et Zhang, 2003).

7.2.4 Recherche d'erreur grâce à la vue en coupe

Généralement, la visualisation des coupes se fait directement dans le code source d'un programme, évitant à l'utilisateur de faire lui-même l'analogie entre les nœuds du graphe présenté et le code qu'il écrit. Ainsi, les lignes de code contenant des variables ou instructions appartenant à un nœud de la coupe sont surlignées directement dans le code source.

De nombreux *slicers* sont disponibles. Nous n'en citerons ici que quelques uns, utilisés pour la coupe de programmes Java. La quasi totalité des slicers orientés objet existants sont basés sur les travaux de Larsen et Harrold (Larsen et Harrold, 1996), bien que Liang (Liang et Harrold, 1998) ait soulevé le fait que le passage de champs Java comme paramètres de méthodes ne soit pas représenté. La première implémentation de ces travaux est celle de Kovacs (Kovacs, Magyar et Gyimothy, 1996), adaptant simplement la méthode aux spécificités du code Java.

Walkinshaw (Walkinshaw, Roper et Wood, 2003) a développé un générateur de

SDG, utilisant la plateforme SOOT (Vallée-Rai et al., 1999). Son approche permet la représentation de l'héritage, mais les exceptions ne sont toujours pas présentes dans les graphes construits. Il a proposé, par la suite, une méthode similaire, mais dynamique (Walkinshaw, Roper et Wood, 2005).

Le slicer Bandera (Dwyer et Hatcliff, 1999) est présenté comme un vérificateur de modèle à partir du code source Java, plutôt qu'un outil d'aide a l'analyse de programme. Cette approche démontre que la coupe d'un programme n'est pas cantonnée au débogage. On retrouve l'utilisation de coupes dans des applications aussi variées que le test de programme, la maintenance, la parallélisation automatique de l'exécution de programme, la vérification, la gestion et l'intégration de versions...

7.3 Débogage bidirectionnel

Le processus de débogage commence lorsque l'utilisateur observe un comportement inattendu du programme. Généralement, il essaie alors de retrouver la cause de ce comportement, afin d'y apporter les corrections nécessaires. La localisation de cette cause débute avec l'émission d'une hypothèse quant à l'emplacement de cette erreur. L'utilisateur relance alors l'exécution du programme et la suspend avant d'avoir atteint l'emplacement supposé. Cette suspension peut être effectuée par divers moyens, selon le débogueur, et permet l'observation de l'environnement avant l'erreur. Durant cette observation, il vérifie ses hypothèses quant à l'état du programme jusqu'à en trouver au moins une fausse. Puis il réitère ce procédé afin de trouver la cause de la différence entre sa supposition et les valeurs observées.

Naturellement, le procédé de recherche d'erreur remonte le cours de l'exécution du programme. Or la majorité des débogueurs standards ne proposent qu'un mécanisme d'exécution pas à pas dans « le sens de la marche ». Tenant compte de cette inadéquation entre la démarche naturelle et celle contrainte par les capacités des outils, des études ont été menées quant à l'ajout d'une fonctionnalité de pas à pas arrière. On parle alors de débogage bidirectionnel.

Deux approches existent actuellement pour la réalisation d'une telle fonctionnalité. La première est basée sur l'historique d'exécution, et demande de conserver les données nécessaires à la réexécution. La seconde utilise des points de reprise : lors d'une exécution arrière, elle rejoue l'exécution depuis le dernier point de reprise jusqu'à l'instruction précédant l'instruction courante. Les sections suivantes traitent ces deux approches, dans l'ordre indiqué, avant de conclure sur l'utilisation d'outils existants.

7.3.1 Solution basée sur la conservation de l'historique d'exécution

Les méthodes exposées dans cette partie sont celles basées sur la conservation (généralement dans un fichier, sur disque) d'information relatives à l'exécution d'une application. Deux principales tendances se dégagent : (1) la conception d'outils permettant une visualisation conviviale des informations recueillies (on parle alors d'*offline debugging*), (2) l'utilisation des informations stockées pour faciliter les sauts durant l'exécution.

La collecte d'informations relatives à l'exécution nécessite une instrumentation du code de l'application, afin de pouvoir récupérer les informations pertinentes. Dans de nombreux cas, cette instrumentation passe par la modification du compilateur. Cette modification permet d'insérer dans le code compilé des instructions de récupération aux instructions désirées (par exemple l'incrémentation d'un compteur lors de chaque appel de fonction). Cette méthode nécessite des connaissances suffisantes des théories régissant la compilation du langage de l'application. Puis, une fois l'historique généré, un autre outil permet de visualiser les données collectées.

Récemment, les travaux de Lewis et Ducassé (Lewis et Ducassé, 2003) ont produit un débogueur bidirectionnel pour le langage Java, basé sur la conservation de l'historique d'exécution : ODB (Omniscient Debugger). Leur outil conserve un trace de tous les changements d'états du programme, tirant partie du système d'événements mis en place par la JPDA. Les événements relatifs à une affectation ou à l'appel de fonctions sont stockés sur disque durant l'exécution de l'application. Une fois l'exécution termi-

née, l'utilisateur peut alors utiliser ODB afin de naviguer parmi les différents états du programme, et donc rechercher la faute sans contrainte lors des déplacements. Dans ce cas, aucune connaissance concernant la compilation du langage Java n'est nécessaire afin d'obtenir les informations désirées.

Afin de permettre des mouvements vers les états précédents du programme, de nombreuses études ont proposé la génération du code inverse de l'application. Cette génération n'est pas toujours évidente, et le cas des instructions non structurées en langage C en est un exemple. On appelle instruction non structurée les instructions conduisant à un saut dans l'exécution vers un point arbitraire du programme. C'est le cas des instructions *goto, break, return* ou *continue.* Une fois la cible du saut atteinte, il est difficile de déterminer l'origine du saut (cette origine pouvant ne pas être unique). Biswas et Mall (Biswas et Mall, 1999) se sont appuyés sur une partie de l'historique afin de faciliter la génération des instructions inverses. La figure 7.2 présente un exemple de programme, du fichier contenant l'historique et du programme inverse correspondants.

Dans toutes les solutions basées sur la conservation de l'historique d'exécution, le stockage d'une quantité considérable de données reste une contrainte importante, et réduit l'utilisation de telles techniques aux systèmes disposant de ressources suffisantes.

7.3.2 Solution basée sur l'utilisation de points de reprise

On ne peut vraisemblablement pas considérer une approche de débogage bidirectionnel sans conservation d'un historique quelconque. Cependant, l'espace mémoire utilisé pour de tels historiques peut devenir prohibitif. Ainsi, des études ont été menées afin de permettre une utilisation minimale de ces historiques, réduisant ainsi le volume d'information à conserver.

Boothe propose une approche basée sur l'utilisation de points de reprise, ne stockant que l'historique des entrées/sorties du programme (Boothe, 2000). Lors de la compilation en mode débogage du programme, du code est inséré à chaque début d'instruction. Ce code incrémente des compteurs globaux de deux types : (1) compteur de

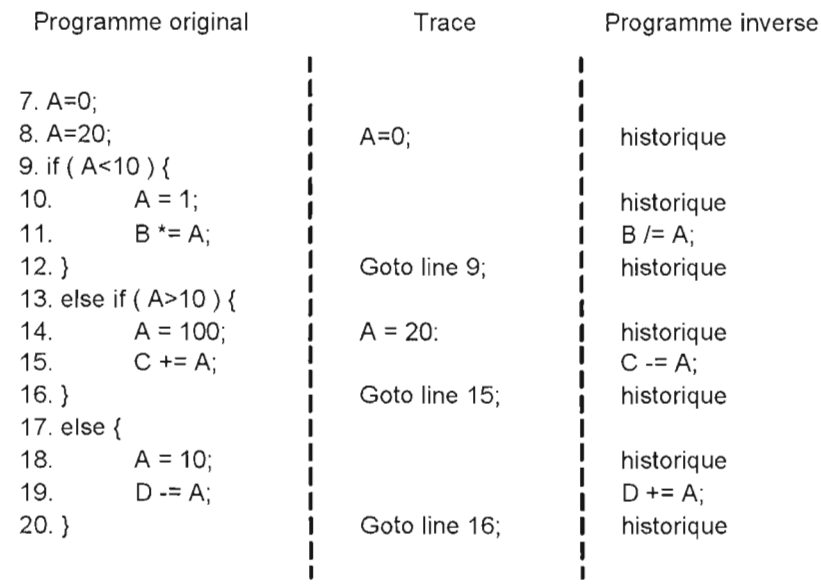


Fig. 7.2 Exemple de programme et de son inverse (Biswas et Mall, 1999)

pas, unité de mesure du système, (2) compteur de profondeur, incrémenté à chaque appel de fonction, permettant de connaître la hauteur de la pile des fenêtres d'exécution. D'autres compteurs spécialisés, sont insérés dynamiquement lors de l'appel aux routines de déplacement (avant ou arrière). Lors de l'appel à une de ces routines, la destination du saut est exprimée en fonction des valeurs des différents compteurs mis en place. On réexécute alors le programme et le suspend lorsque les compteurs atteignent les valeurs de destination. Pour la plupart des mouvements en arrière, au moins deux réexécutions sont nécessaires : la première met en place les compteurs spécifiques au mouvement désiré, la seconde utilise ces compteurs pour suspendre son cours à l'emplacement approprié.

Afin de réduire le coût des réexécutions, on enregistre, à intervalles réguliers, l'état du programme. Ces enregistrements sont appelés points de reprise. Restent à définir *quand* et *quoi* enregistrer.

Pour décider *quand* placer ces points, on doit considérer que plus la durée des intervalles entre chaque point est importante, moins le coût induit par leur insertion sera important : le volume d'information à enregistrer à chaque point est moins élevé. Mais les calculs effectuées lors de la réexécution sont proportionnels à la distance entre l'emplacement de l'exécution courante et le point de reprise le plus proche. Partant de l'hypothèse que plus une exécution est longue, plus elle a de chance de se prolonger, on peut insérer et supprimer les points de reprise dynamiquement : on ne conserve que les n plus récents, et l'intervalle séparant deux points augmente de manière exponentielle. Initialement, la durée d'un intervalle peut être configurée à une valeur entre 0.1 et 1 seconde, le point de reprise étant posé au pas suivant (défini par le compteur de pas).

En insérant ces points, on crée un ensemble de fenêtres d'exécution, pouvant chacune être rejouée séparément. Selon l'importance que l'on accorde à cette propriété d'indépendance des fenêtres, plusieurs critères sont à prendre en compte afin de décider *quoi* enregistrer (Netzer et Weaver, 1994). Une première stratégie consisterait à stocker l'ensemble de valeurs visibles dans la fenêtre précédente, ce qui permettrait une autonomie des fenêtres mais augmente le volume d'information à conserver avec chaque point de reprise. Il serait également possible de ne conserver que les valeurs ayant été accédées dans la fenêtre d'exécution précédente. Dans ce cas, l'indépendance n'est plus assurée. Mais en réduisant le volume d'information enregistrée, on oblige l'outil à se munir d'un mécanisme de récupération des valeurs dans les points de reprise précédents. Une dernière méthode consisterait à n'enregistrer que les écritures et assignations. Par rapport à la méthode précédente, on diminue encore l'information à conserver, mais le mécanisme de recherche des valeurs devrait alors remonter plus loin dans l'historique afin de récupérer une valeur.

La validité de cette méthode repose sur le caractère déterministe de la réexécution. La conservation de l'historique des entrées/sorties y joue un rôle capital. c'est pourquoi on enregistre également les valeurs retournées par les appels à des routines du système d'exploitation (lecture d'un fichier, date...). Dans certains cas, toutefois, les routines systèmes sont simplement réexécutées. C'est le cas pour la primitive permettant de

modifier le sommet de la pile d'un processus.

7.3.3 Recherche d'erreur grâce au débogage bidirectionnel

Lors du débogage d'une application, on ne prend généralement pas en consideration les dégradation de performance dues à l'utilisation d'un débogueur. La collecte d'information lors de l'exécution demande une charge supplémentaire au système, notamment dans le cas du débogage bidirectionnel. C'est pourquoi les applications en temps réel sont souvent considérées comme le parent pauvre du débogage (Glass, 1980).

L'utilisation de méthodes basées sur l'historique d'exécution se heurte à la croissance rapide de cet historique. Si celui-ci est conservé sur disque, chaque accès au fichier dégrade les performances de l'application. Ainsi, l'utilisation de l'historique multiplie le temps d'exécution par un facteur de 4 à 7, selon le type d'application. De plus, cette méthode requiert l'utilisation d'un interpréteur, ralentissant l'exécution d'un facteur de 20 à 40 fois pour la machine virtuelle Java Kaffe (Cook, 2002), par exemple.

Dans le cas de la méthode utilisant des points de reprise, les performances varient grandement selon les types d'applications. Cette différence réside principalement dans la conservation de l'historique des entrées/sorties : les applications faisant un usage intense d'interfaces graphiques seront moins affectées que les applications de calcul. De même, pour les applications distribuées, cet historique permet de supprimer le temps de transit des données sur le réseau, améliorant ainsi les performances. Finalement, on observe un temps de réexécution lors du débogage supérieur de 95 à 124% du temps d'exécution initial.

L'outil développé par Boothe (Boothe, 2000) dans sons étude contient toutes les commandes d'un débogueur traditionnel, étendu des mouvements inverses (*next*, *bnext*, *step*, *bstep*...).

7.4 Conclusion

Cette section a présenté trois paradigmes de débogage. Chacun de ces concepts a été développé dans le but de réduire le temps consacré à la recherche de faute au sein d'un programme. Le delta débogage permet l'automatisation de la recherche de cas de test minimaux reproduisant un bogue donné. La vue en coupe ne présente à l'utilisateur que les informations pertinentes à sa recherche. Le débogage bidirectionnel offre une latitude de mouvement plus importante au programmeur lors de la recherche d'erreur.

Dans tous les cas, la mise en pratique de ces concepts passe par le développement d'un outil de débogage offrant les fonctionnalités présentées, et requiert donc, dans le cas de Java, une collaboration de l'environnement d'exécution (machine virtuelle).

Chapitre VIII

CONCLUSION

Ce mémoire a présenté la réalisation de la première interface de débogage Java totalement libre.

Ayant opté pour l'architecture proposée par Sun Microsystems, la Java Platform Debugger Architecture, nous avons tout d'abord mis en place, les mécanismes de support au débogage au sein de SableVM, une machine virtuelle Java libre et conforme aux normes. Cette implantation a comporté un remaniement de certains algorithmes internes de la machine virtuelle, ainsi que l'ajout de nouvelles fonctionnalités relatives au débogage. Ainsi, à l'issue de nos travaux, SableVM a été la première machine virtuelle Java libre dotée de la Java Virtual Machine Debug Interface.

Cette étude a également conduit à la conception et à la réalisation d'un module indépendant, en charge de connecter machine virtuelle Java et débogueur. Ce module permet la transcription d'information sous forme de paquets respectant le protocole JDWP (Java Debug Wire Protocol) et leur envoi sur une machine distante via un socket. D'autre part, il est responsable de la gestion (attribution d'identifiants, stockage et indexation) des objets manipulés durant toute la session de débogage : . Il permet également la gestion des événements générés par la machine virtuelle. L'indépendance de ce module lui permet d'être utilisé entre toute machine virtuelle munie d'interface de débogage et tout débogueur Java conforme aux normes.

De plus, nous avons assemblé les éléments conçus ou modifiés à d'autres éléments

existants et disponibles librement, afin de mettre en place la première architecture de débogage Java totalement libre.

Les résultats obtenus lors des tests ont démontré la pertinence des différents choix effectués lors du développement. L'utilisation de débogueurs totalement indépendants de la machine virtuelle utilisée, tel Eclipse, et la bonne tenue des sessions de débogage effectuées ont permis la validation de la conformité de nos travaux aux normes en vigueur.

Tout au long de ce mémoire, nous nous sommes efforcés de donner un aperçu des normes régissant le débogage Java, proposant ponctuellement certaines améliorations. Nous espérons ainsi clarifier certains points et mettre en évidence les concepts sous-jacents à certaines fonctionnalités présentes dans les normes étudiées.

Finalement, cette étude peut également servir de guide, jalonnant le développement d'une interface de débogage, aussi bien au cœur de la machine virtuelle qu'au sein du module indépendant. Dans cette optique, nous avons tenté de soulever certains problèmes de manière générique, sans rester spécifique aux outils utilisés (SableVM, Eclipse...).

Annexe A

SCRIPT D'INSTALLATION RAPIDE

SableVM possède désormais les interfaces nécessaires au débogage du code Java. Cette annexe fournit les instructions nécessaires à la mise en place et le paramétrage de tous les éléments de l'architecture de débogage.

Notez que deux machines virtuelles sont nécessaires : l'une pour exécuter votre application Java, l'autre pour l'exécution d'Eclipse. La première doit impérativement être celle disponible à l'issue de nos travaux, la seconde une version récente de SableVM.

Il est important de ne pas exécuter Eclipse avec la version de SableVM contenant les fonctionnalités de débogage.

Ces instructions, ainsi qu'une version plus détaillée, sont également disponibles sur le Site officiel de SableVM : http ://sablevm.org

A.1 Récupérer et installer les sources

Les instructions suivantes supposent l'installation préalable de toutes les dépendances nécessaires. Pour les utilisateurs Debian, exécutez :

apt-get build-dep sablevm sablevm-classlib

Puis, exécutez les commandes suivantes :

```
su cd /tmp

wget -c
ftp://ftp.cse.buffalo.edu/pub/Eclipse/eclipse/downloads/drops/
S-3.1M4-200412162000/eclipse-SDK-3.1M4-linux-gtk.zip

wget -c
http://sablevm.org/people/nizar/sablevm-nizar-debug+3895.tar.gz

wget -c
http://sablevm.org/people/nizar/sablevm-classpath-nizar-debug+3895.tar.gz

wget -c http://sablevm.org/people/nizar/eclipse-sablevm

wget -c
http://sablevm.org/download/release/1.11.3/sablevm-1.11..tar.gz

wget -c
http://sablevm.org/download/release/1.11.3/sablevm-classpath-1.11.3.tar.gz

for archive in sablevm-1.11.3.tar.gz
sablevm-classpath-1.11.3.tar.gz sablevm-nizar-debug+3895.tar.gz
sablevm-classpath-nizar-debug+3895.tar.gz; do tar -zxf \$ archive;
done

cd /tmp/sablevm-1.11.3 && ./configure && make install

cd /tmp/sablevm-classpath-1.11.3 && ./configure && make install

cd /tmp/sablevm-nizar-debug+3895 && ./configure && make install

cd /tmp/sablevm-classpath-nizar-debug+3895 && ./configure && make
install

cd /usr/local && unzip /tmp/eclipse-SDK-3.1M4-linux-gtk.zip

chmod +x /tmp/eclipse-sablevm && cp -f /tmp/eclipse-sablevm
/usr/local/bin

exit
```

A.2 Paramétrer Eclipse

Une fois l'installation achevée, vous devriez être capables de lancer Eclipse en tapant, dans un terminal, la commande : eclipse-sablevm

Les instructions suivantes vous permettent de paramétrer votre environnement de développement Eclipse afin d'utiliser la version débogage de SableVM :

* Si vous lancez Eclipse pour la première fois, choisissez l'emplacement de votre espace de travail (Workspace) et choisissez 'Workbench' à l'écran de lancement,
* Dans le menu, choisissez Window puis Preferences,
* Déroulez le menu 'Java' et choisissez 'Installed JREs',
* Cliquez sur 'Search...', naviguez jusqu'à /usr/local, sélectionnez sablevm-debug et cliquez 'OK'.
* Cochez la case pour sélectionner sablevm comme JRE par défaut et cliquez 'OK'.

Vous êtes désormais capables de tirer tous les avantages d'un environnement de développement ergonomique afin de créer vos applications Java.

Bibliographie

(Agrawal et Horgan, 1990) Agrawal, H. et Horgan, J. R. 1990. « Dynamic program slicing ». In *PLDI '90 : Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, p. 246–256, White Plains, NY, USA. ACM Press.

(Biswas et Mall, 1999) Biswas, B. et Mall, R. 1999. « Reverse execution of programs ». In *ACM SIGPLAN Notices*, p. 61–69. ACM Press.

(Boothe, 2000) Boothe, B. 2000. « Efficient algorithms for bidirectional debugging ». In *PLDI '00 : Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, p. 299–310, Vancouver, BC, Canada. ACM Press.

(Bouillon, Burger et Zeller, 2003) Bouillon, P., Burger, M., et Zeller, A. 2003. « Automated debugging in eclipse : (at the touch of not even a button) ». In *eclipse '03 : Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, p. 1–5, Anaheim, CA, USA. ACM Press.

(Chen et Xu, 2001) Chen, Z. et Xu, B. 2001. « Slicing object-oriented java programs ». In *ACM SIGPLAN Notices*, p. 33–40. ACM Press.

(Choi et Zeller, 2002) Choi, J.-D. et Zeller, A. 2002. « Isolating failure-inducing thread schedules ». In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, Rome, Italie. Egalement publie en tant qu'IBM Research Report RC22440 (W0205-083).

(Cleve et Zeller, 2005) Cleve, H. et Zeller, A. 2005. « Locating causes of program failures ». In *Proceedings of the 27th International Conference on Software Engineering*, p. 342–351, St Louis, MO, USA. ACM Press.

(Cook, 2002) Cook, J. J. 2002. « Reverse execution of java byte-code ». In *Computer Journal*, p. 608–619. Oxford University Press.

(Dwyer et Hatcliff, 1999) Dwyer, M. B. et Hatcliff, J. 1999. « Slicing software for model construction ». In *Proceedings of the 1999 ACM Workshop on Partial Evaluation and Semantic-Based Program Manipulation*, p. 105–118. ACM Press.

(Gagnon et Hendren, 2003) Gagnon, E. et Hendren, L. 2003. « Effective inline-threaded interpretation of java bytecode using preparation sequences ». In *Proceedings of the 12th International Conference on Compiler Construction : CC 2003*, p. 170–184, Varsovie, Pologne. Springer.

(Gagnon, 2002) Gagnon, E. M. 2002. « A portable research framework for the execution of java bytecode ». Thèse de Doctorat, McGill University, Canada.

(Glass, 1980) Glass, R. L. 1980. « Real-time : the lost world of software debugging and testing ». In *Communications of the ACM*, p. 264–271. ACM Press.

(Hailpern et Santhanam, 2002) Hailpern, B. et Santhanam, P. 2002. « Software debugging, testing, and verification », *IBM Systems Journal*, vol. 41, no. 1. http ://www.research.ibm.com/journal/sj/.

(Heisenberg, 1930) Heisenberg, W. 1930. *The Physical Principles of the Quantum Theory*. University of Chicago Press.

(Horwitz, Reps et Binkeley, 1988) Horwitz, S., Reps, T., et Binkeley, D. 1988. « Interprocedural slicing using dependence graphs ». In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and implementation*, p. 35–46, Atlanta, GA, USA. ACM Press.

(Intel, 1995) Intel 1995. Intel386 sx microprocessor. Rapport, Intel Corporation. http ://www.intel.com/design/intarch/datashts/240187.htm.

(Kovacs, Magyar et Gyimothy, 1996) Kovacs, G., Magyar, F., et Gyimothy, T. 1996. Static slicing of java programs. Rapport, Jozsef Attila University, Hongrie.

(Larsen et Harrold, 1996) Larsen, L. et Harrold, M. J. 1996. « Slicing object-oriented software ». In *ICSE '96 : Proceedings of the 18th international conference on Software engineering*, p. 495–505, Berlin, Allemagne. IEEE Press.

(Lewis et Ducassé, 2003) Lewis, B. et Ducassé, M. 2003. « Using events to debug java programs backwards in time ». In *OOPSLA '03 : Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, p. 96–97, Anaheim, CA, USA. ACM Press.

(Liang et Harrold, 1998) Liang, D. et Harrold, M. J. 1998. « Slicing objects using system dependence graphs ». In *ICSM '98 : Proceedings of the International Conference on Software Maintenance*, p. 358–367. IEEE Computer Society.

(Liang, 1999) Liang, S. 1999. *Java Native Interface : Programmer's Guide and Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

(Lindholm et Yellin, 1999) Lindholm, T. et Yellin, F. 1999. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

(Netzer et Weaver, 1994) Netzer, R. H. B. et Weaver, M. H. 1994. « Optimal tracing and incremental reexecution for debugging long-running programs ». In *PLDI '94 : Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, p. 313–325, Orlando, FL, USA. ACM Press.

(Piumarta et Riccardi, 1998) Piumarta, I. et Riccardi, F. 1998. « Optimizing direct-threaded code by selective inlining ». In *SIGPLAN 98 : Conference on Programming Language Design and Implementation*, p. 291–300. ACM Press.

(Rosenblum, 2004) Rosenblum, M. 2004. « The reincarnation of virtual machines ». In *ACM Queue*, p. 34–40. Edward Grossman.

(Sleator et Tarjan, 1985) Sleator, D. D. et Tarjan, R. E. 1985. « Self-adjusting binary search trees ». In *Journal of the ACM*, p. 652–686. ACM Press.

(Smith et Nair, 2005) Smith, J. E. et Nair, R. 2005. *Virtual Machine Architectures, Implementations and Applications.* Elsevier Science.

(Stallman, Pesch et Shebsr, 2002) Stallman, R., Pesch, R., et Shebsr, S. 2002. *Debugging with GDB : The GNU Source-Level Debugger.* GNU Press.

(Sun, 2002) Sun 2002. *Java Platform Debugger Architecture.* http ://java.sun.com/j2se/1.4.2/docs/guide/jpda/.

(Vallée-Rai et al., 1999) Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., et Co, P. 1999. « Soot - a java optimization framework ». In *Proceedings of CASCON 1999*, p. 125–135. ACM Press. http ://www.sable.mcgill.ca/soot.

(Walkinshaw, Roper et Wood, 2003) Walkinshaw, N., Roper, M., et Wood, M. 2003. « The java system dependence graph ». In *Proceedings of Source Code Analysis and Manipulation (SCAM'03)*, p. 55–64, Amsterdam, Pays Bas. IEEE Press.

(Walkinshaw, Roper et Wood, 2005) —— 2005. « Understanding object-oriented source code from the behavioural perspective ». In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*, p. 215–224, St

Louis, MO, USA. IEEE Press.

(Weiser, 1981) Weiser, M. 1981. « Program slicing ». In *ICSE '81 : Proceedings of the 5th international conference on Software engineering*, p. 439–449, San Diego, CA, USA. IEEE Press.

(Xu et al., 2005) Xu, B., Qian, J., Zhang, X., Wu, Z., et Chen, L. 2005. « A brief survey of program slicing ». In *SIGSOFT Software Engineering Notes*, p. 1–36, Washington DC, USA. ACM Press.

(Zeller, 2002) Zeller, A. 2002. « Isolating cause-effect chains from computer programs ». In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE-10)*, p. 1–10, Charleston, SC, USA. ACM Press.

(Zeller et Hildebrandt, 2002) Zeller, A. et Hildebrandt, R. 2002. « Simplifying and isolating failure-inducing input ». In *IEEE Transaction on Software Engineering*, p. 183–200, Piscataway, NJ, USA. IEEE Press.

(Zhang et Gupta, 2004) Zhang, X. et Gupta, R. 2004. « Cost effective dynamic program slicing ». In *PLDI '04 : Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, p. 94–106, Washington DC, USA. ACM Press.

(Zhang, Gupta et Zhang, 2003) Zhang, X., Gupta, R., et Zhang, Y. 2003. « Precise dynamic slicing algorithms ». In *ICSE '03 : Proceedings of the 25th International Conference on Software Engineering*, p. 319–329, Washington DC, USA. IEEE Computer Society.

(Zhao, 1999) Zhao, J. 1999. « Slicing concurrent java programs ». In *IWPC '99 : Proceedings of the 7th International Workshop on Program Comprehension*, p. 126–133, Fukuoka, Japon. IEEE Computer Society.