

THÈSE

présentée à

L'UNIVERSITÉ PIERRE ET MARIE CURIE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ PARIS VI

Spécialité

Informatique

soutenue par

Gaël THOMAS

Le 20 mai 2005

APPLICATIONS ACTIVES

**Construction dynamique d'environnements d'exécution flexibles
homogènes**

Directeur de thèse : Professeur Bertil FOLLIOT

Jury

Président	Jacques	MALENFANT	Professeur à l'Université Pierre et Marie Curie.
Rapporteurs	Gilles	MULLER	Professeur à l'École des Mines de Nantes.
	Jean-Bernard	STEFANI	Directeur de Recherche à l'INRIA.
Examineurs	Bertil	FOLLIOT	Professeur à l'Université Pierre et Marie Curie.
	Gilles	GRIMAUD	Maître de Conférences à l'Université des Sciences et Technologies de Lille.
	Willy	ZWAENEPOEL	Professeur à l'École Polytechnique Fédérale de Lausanne.

Table des matières

Remerciements	i
Résumé	iii
1 Introduction	1
1 Mise en perspective des applications actives	2
1.1 Domain Specific Languages	2
1.2 Exo-noyaux	2
1.3 Réseaux actifs	3
1.4 Réflexion	3
1.5 Environnements dédiés et surcouches logicielles	3
2 Construction des applications actives	4
2.1 Conception des applications actives	4
2.2 Conception de la μvm	5
2.3 Niveaux d'adaptation	6
3 Plan de la thèse	7
3.1 Adaptation dynamique	7
3.2 Architecture de Machines Virtuelles Virtuelles pour les Applications Actives	8
3.3 Les documents actifs	8
3.4 Construction d'une machine virtuelle Java sous forme de composants adaptables : la JnJVM	9
3.5 Évaluation qualitative de la JnJVM et de la μvm	9
3.6 Conclusions et perspectives	10
4 Le projet Machines Virtuelles Virtuelles	10
2 Adaptabilité dynamique	13
1 Introduction	13
2 Environnements flexibles	13
2.1 Utilisation de la flexibilité dans les environnements	14
2.2 Flexibilité dans les intergiciels	20
3 Outils dédiés à l'adaptation dynamique	23
3.1 La réflexion	23
3.2 Le tissage d'aspects	31
3.3 Les outils d'adaptation : discussion	33
4 Domaines connexes à l'adaptabilité : réseau et langage	35

4.1	Les réseaux actifs	35
4.2	Les Domain Specific Languages	38
5	Les challenges de l'adaptabilité	39
3	Architecture de Machines Virtuelles Virtuelles pour Applications Actives	41
1	Introduction	41
1.1	Les propriétés de la μvm	43
2	La Ynvm	45
2.1	Les Arbres de Syntaxe Abstraite	45
2.2	Compilation et VPU	46
2.3	Réflexivité au niveau langage	47
2.4	Réflexion fonctionnelle	48
3	La μvm	49
3.1	Les ramasse-miettes	50
3.2	Réflexion des composants	56
3.3	Lexeur et Parseur	60
4	Récapitulatif	61
4	Machines Virtuelles Virtuelles pour documents actifs	65
1	Introduction	65
2	Dynamicité des documents	67
2.1	L'approche script	67
2.2	Les approches agents	67
2.3	Script versus Agent	68
3	MVLets pour document multimédia	68
3.1	L'architecture des documents actifs	68
3.2	Réalisation des Documents Actifs	70
4	Deux adaptations des MVLets de documents actifs	72
4.1	Production d'un PostScript Portable	74
4.2	Le composeur de notes	74
5	Qualité de service	75
5.1	Documents actifs et Bossa	75
5.2	La MVLet Bossa	76
5.3	Une vidéo auto-ordonnancée dans un Document Actif	77
6	Conclusion et perspectives	78
5	La JnJVM : construction d'une machine virtuelle Java adaptable	81
1	Introduction	81
2	Conception de la JnJVM	83
3	Réalisation de la JnJVM	85
3.1	Composants JnJVM	85
3.2	Liaison	88
3.3	Compilation	90
3.4	Exception et gestion d'erreurs	91
3.5	Les langages dédiés à l'adaptation	91
3.6	Symboles Java, symboles μvm	92
4	Performances de la JnJVM	93

4.1	Démarrage	94
4.2	Calculs entiers	95
4.3	Calculs flottants	96
4.4	Allocations d'objets	97
4.5	Appels de méthodes	98
4.6	Conclusions sur les mesures	99
5	Conclusion	99
6	Réalisation de quatre applications d'évaluation de la JnJVM	101
1	Introduction	101
2	Adaptation du langage d'entrée	102
2.1	La JVM adaptable à distance	103
2.2	Une entrée XML pour la μvm	106
3	La JVM à analyse d'échappement	107
3.1	L'analyse d'échappement	108
3.2	La MVLet d'analyse d'échappement	109
3.3	Mise en œuvre de l'allocation	112
3.4	Performances de l'analyse d'échappement	114
3.5	Conclusion	119
4	Tissage dynamique d'aspects	119
4.1	La MVLet aspect	120
4.2	Intégration de la JnJVM avec un outil de modélisation	122
4.3	Conclusion	122
5	La JVM à migration de fil d'exécution	123
5.1	Travaux similaires	124
5.2	Migration générique	125
5.3	Migration d'un client Jabber	126
5.4	Conclusion	128
6	Récapitulatif	129
7	Conclusions et perspectives	131
1	Les applications actives, la μvm , les documents actifs et la JnJVM	131
2	Les applications d'évaluation	132
3	Perspectives	133
	Bibliographie	137

Table des figures

1.1	Ensemble des travaux menés dans le projet Machine Virtuelle Virtuelle	11
2.1	Les besoins en adaptation	19
2.2	La flexibilité dans les intergiciels	23
2.3	Utilisation de Proxy en Java	26
2.4	La réflexion dans Dalang	29
2.5	Composeur séquentiel dans Guaraná	30
2.6	Plate-formes dédiées à l'adaptation	34
2.7	Architecture DARPA des réseaux actifs	36
2.8	Déploiement dans ANTS	36
3.1	Architecture des applications actives	42
3.2	Vocabulaire des applications actives	43
3.3	Compilation dans la Ynvm	45
3.4	Utilisation de la VPU	46
3.5	Code compilé en mémoire	46
3.6	Utilisation des syntaxes	47
3.7	Utilisation de la VPU dans une syntaxe	48
3.8	Une mémoire objet	51
3.9	Problème de génération	54
3.10	Structuration du MOP de la μ vm	56
3.11	Structure d'un composant	57
3.12	Architecture d'un méta-composant et de sa méta-interface	58
3.13	Fonctionnement du lexeur et du parseur	61
3.14	Architecture de la μ vm	63
3.15	Introspection dans la μ vm	64
4.1	Architecture globale des Documents Actifs	69
4.2	Architecture des documents actifs multimédias	69
4.3	Exemple de document actif	70
4.4	Une vidéo dans un PostScript	71
4.5	Synchronisation entre MVLets	72
4.6	La MVLet image	73
4.7	Architecture de l'intégration de Bossa aux documents actifs	76
4.8	Ordonnancement en cas de famine	78
4.9	Les MVLets des Documents Actifs	80

5.1	La machine virtuelle virtuelle Java	83
5.2	Les composants de la JnJVM	84
5.3	Réalisation de la JnJVM	87
5.4	Liaison statique de méthode	88
5.5	Appel de méthode virtuelle	89
5.6	Liaison virtuelle de méthode	89
5.7	Exécution de <code>HelloWorld</code>	94
5.8	Utilisation de la mémoire	95
5.9	Arithmétique entière	96
5.10	Arithmétique double	96
5.11	Test de la gestion mémoire	97
5.12	Appels de méthodes	98
6.1	Thématiques abordées dans ce chapitre et niveau adapté dans une MVV . . .	102
6.2	Fonctionnement de la DVVM	104
6.3	La syntaxe XML	106
6.4	Une feuille de style XML	107
6.5	Un exemple d'échappement en Java	109
6.6	Héritage et analyse d'échappement	110
6.7	L'aspect analyse d'échappement	112
6.8	Algorithme de la pile d'allocation	113
6.9	Collection de la pile	114
6.10	Analyse d'échappement statique	115
6.11	Impact de l'analyse d'échappement	116
6.12	Une fonction qui ralentit la JVM à analyse d'échappement	118
6.13	Un exemple de tissage d'aspects	121
6.14	Structure des classes modifiées	123
6.15	Architecture du protocole Jabber	126
6.16	Protocole de migration	127
6.17	Récapitulatif de nos travaux sur les applications actives	130

Remerciements

Je tiens d'abord à remercier Valérie pour sa bonne humeur, son soutien quotidien pendant ces trois années de thèse et pour sa patience lors des nuits passées devant ma machine à rédiger ou à programmer.

Je remercie aussi chaleureusement Bertil pour son encadrement exemplaire, particulièrement pour le soin qu'il a apporté à la relecture de ma thèse et pour le sujet passionnant qu'il m'a proposé, mais aussi pour ses qualités humaines, sa convivialité et son sens de l'écoute.

Toutes les personnes qui ont collaboré ou collaboreront encore au projet Machine Virtuelle Virtuelle et avec qui j'ai eu l'occasion de travailler, Ian Piumarta, Frederic Ogel, Antoine Galland, Assia Hachichi, Cyril Martin, Cyrille Mescam, Charles Clement, Nicolas Geaoffray, Henri-Pierre Charles, Idriss Ben El Kezadri et Marwan Jabbour, ont participé à l'élaboration de ces travaux. Sans eux, et en particulier Ian Piumarta qui a écrit la Ynvm et qui a toujours été disponible pour m'éclairer de ses conseils avisés, je n'aurais jamais pu explorer autant de facettes des applications actives et je leur exprime toute ma gratitude.

Je remercie Gilles Grimaud, Maître de conférence à l'USTL, Jacques Malenfant, Professeur au LIP6 et Willy Zwaenepoel, Professeur à l'EPFL, pour avoir accepté de participer à mon jury de thèse et pour l'attention qu'ils ont portée à mon travail.

Je remercie Gilles Muller, professeur à l'École des Mines de Nantes, à la fois pour avoir accepté d'être rapporteur de ma thèse, mais aussi pour le temps qu'il a consacré à discuter avec moi et à l'intérêt qu'il a porté à mes travaux tout au long de mon doctorat.

Je remercie Jean-Bernard Stefani, directeur de recherche à l'INRIA, d'avoir accepté d'être rapporteur de ma thèse et pour les commentaires et les perspectives de recherche passionnantes qu'il m'a indiquées.

Ces trois années ont aussi été pour moi l'occasion de rencontrer des personnes de grande qualité dans le thème Systèmes Répartis et Coopératifs du Laboratoire d'Informatique de Paris 6, je les remercie pour leur accueil et leurs conseils, en particulier Claude Giraud, Luciana Arantes, Lionel Seinturier, Pierre Sens, Fabrice Kordon, Olivier Marin, Xavier Blanc, Thierry Lanfroy, Denis Poitrenaud et Marie-Pierre Gervais. Je remercie aussi spécialement Marin Bertier, Yann Thierry-Mieg, Fabrice Legond-Aubry et Frédéric Gilliers qui ont débuté leur thèse en même temps que moi et pour qui j'ai une grande amitié.

Je remercie enfin ma famille et mes amis proches pour leur confiance inébranlable, leur présence et leur compréhension et pour le travail de relecture de certains qui a du être spécialement indigeste pour des novices du domaine informatique.

Résumé

L'émergence de nouveaux domaines informatiques entraîne de nouveaux besoins en terme de mécanismes systèmes que les environnements traditionnels ne couvrent pas. Les concepteurs n'ont alors pas d'autres choix que de se tourner vers une surcouche applicative au-dessus d'un environnement existant ou de modifier un environnement existant pour qu'il offre le mécanisme dédié. Aucune de ces deux solutions n'est satisfaisante. D'une part, les surcouches logicielles sont limitées par l'environnement d'exécution et ajoutent une couche qui diminue les performances, d'autre part, les environnements dédiés sont incompatibles entre eux et augmentent l'hétérogénéité logicielle.

Pour réconcilier les performances et la portabilité tout en offrant le mécanisme dédié exact requis par le domaine applicatif, nous proposons de placer le code spécifique dans l'application et d'exécuter l'application dans un environnement d'exécution générique. Une telle *application* devient *active* : elle agit sur son environnement d'exécution générique pour qu'il offre le mécanisme dédié nécessaire à l'exécution de l'application. La construction des applications actives repose sur un environnement d'exécution minimal appelé la micro machine virtuelle (μvm). La μvm est un compilateur dynamique réflexif : la compilation dynamique est utilisée par les applications actives pour injecter le code système et la réflexion est utilisée pour connaître et modifier les propriétés de l'environnement d'exécution.

Dans cette thèse, nous présentons l'architecture des applications actives ainsi que leur environnement d'exécution, la μvm . Cette architecture est validée par deux applications majeures : un environnement d'exécution spécialisable pour documents et une machine virtuelle Java extensible (appelée la JnJVM). Trois spécialisations de la machine virtuelle Java ont été implantées dans ces travaux : une spécialisation de tissage d'aspects permettant à une application active de tisser des aspects applicatifs, une spécialisation de la gestion de la mémoire objet reposant sur de l'analyse statique d'échappement augmentant les performances de l'application jusqu'à 24% et une spécialisation de migration de fil d'exécution permettant de déplacer un fil d'exécution Java d'une machine source vers une machine cible tout en prenant en compte le contexte de l'application. Les applications actives permettent la construction de ces trois spécialisations tout en gardant des applications portables : chaque spécialisation est embarquée dans le code actif de l'application active.

CHAPITRE 1

Introduction

Sommaire

1	Mise en perspective des applications actives	2
2	Construction des applications actives	4
3	Plan de la thèse	7
4	Le projet Machines Virtuelles Virtuelles	10

Les innovations technologiques en informatique de ces dernières années ont entraîné l'apparition de nouveaux domaines applicatifs. Les nouveaux supports (Personal Digital Assistant, cartes à puces, téléphones portables), les nouveaux environnements logiciels (GRID, peer-to-peer) et les nouveaux besoins (qualité de service, télétravail, informatique nomade, ubiquité) ont amené leur lot de nouvelles contraintes que les systèmes traditionnels ne savent pas gérer : ressources matérielles faibles, code mobile, déploiement, migration, tissage d'aspects, réflexion.

Les systèmes actuels étant rigides, on assiste à une prolifération de systèmes ad-hocs et dédiés offrant les mécanismes nécessaires à ces domaines émergents. Ces environnements ad-hocs sont en général des systèmes traditionnels légèrement modifiés pour offrir le mécanisme dédié. Ils sont donc fonctionnellement proches de l'original mais incompatibles avec lui. Les applications développées pour ces environnements dédiés deviennent incompatibles avec l'environnement original et sont peu réutilisables : la plupart du temps, le code dédié est noyé dans le code fonctionnel et il est difficile d'abstraire le code générique du code dédié. De plus, la prolifération d'environnements d'exécution dédiés diminue la factorisation du code : pour intégrer deux mécanismes disjoints de deux environnements dédiés différents, il faut recommencer le travail d'ingénierie pour intégrer les deux mécanismes simultanément.

Notre objectif est donc d'intégrer des mécanismes dédiés sans perdre en portabilité, et pour cela, nous proposons dans cette thèse de placer le code dédié dans l'application. L'application s'exécute alors dans un environnement générique et reste portable. Une telle **application** est dite **active** : elle agit sur un environnement d'exécution générique pour le spécialiser. Les applications actives ont pour principe de s'intégrer avec l'existant. En effet, il ne s'agit pas de créer de toute pièce de nouvelles applications mais d'enrichir les applications existantes avec du code actif qui s'occupe de modifier l'environnement pour qu'il offre les mécanismes dédiés requis. Le terme application active est aussi utilisé dans les

réseaux actifs : les applications actives décrites dans cette thèse sont une généralisation de la notion d'applications actives dans les réseaux actifs (voir section 1.3).

Les applications actives proposent une solution qui unifie l'adaptation statique et l'adaptation dynamique d'un environnement d'exécution pour qu'il offre les mécanismes requis par l'application. L'adaptation statique (au démarrage) permet à l'application de construire l'environnement d'exécution dont elle a besoin alors que l'adaptation dynamique permet de modifier cet environnement d'exécution (ou les applications qu'il exécute) à la volée, sans avoir à redémarrer l'environnement d'exécution. L'adaptation dynamique permet de modifier l'environnement d'exécution et l'application en fonction du contexte extérieur, par exemple le débit sur un réseau ou la charge d'une machine.

1 Mise en perspective des applications actives

Le problème de l'adaptation des environnements d'exécution n'est pas récent. Les applications actives s'inscrivent dans la continuité des nombreux travaux de recherche autour de l'adaptabilité (statique et dynamique) et de la flexibilité. Elles sont à la croisée de plusieurs thématiques : les Domain Specific Languages [vDKV00, MRC⁺00, LMB02], les exo-noyaux [EKO95, FSLM02, SCS02], les réseaux actifs [TW96, vECGS92, Cal99, GFS98], et la réflexion [Smi82, Mae87b, Mae87a].

1.1 Domain Specific Languages

Les Domain Specific Languages [vDKV00] (DSL) sont des langages spécifiques à un domaine particulier. Ils masquent la complexité du domaine en construisant un langage de haut niveau représentant le domaine manipulé. Les travaux sur les DSL (voir la section 4.2 du chapitre 2) montrent qu'un langage bien adapté à un domaine particulier simplifie le développement d'application de ce domaine. Les applications actives présentées dans nos travaux définissent dynamiquement des DSL pour manipuler le domaine dédié construit par l'application active. De cette façon, une application active construit à la fois les mécanismes dédiés du domaine et le langage permettant de le manipuler.

1.2 Exo-noyaux

Les premiers exo-noyaux, en particulier Aegis [EKO95] visent d'abord à construire des systèmes d'exploitation sous la forme de bibliothèques systèmes. Un mécanisme système est implanté sous la forme d'une bibliothèque utilisateur. De cette façon, une application peut choisir quelle politique système elle veut utiliser en choisissant sa librairie système. Les exo-noyaux résolvent donc le problème posé précédemment : l'application reste générique et construit l'environnement dédié dont elle a besoin.

Toutefois, les exo-noyaux n'offrent pas de cadre pour modifier les bibliothèques systèmes : si une application a besoin d'un nouveau mécanisme dédié, le concepteur doit développer une nouvelle bibliothèque système. En particulier, si une fonctionnalité est transversale à plusieurs bibliothèques, ce travail peut s'avérer long. Think [FSLM02, SCS02] résout ce problème en construisant un exo-noyau à base de composants extensibles. Think offre donc de puissants mécanismes pour modifier dynamiquement le cœur du système mais s'intéresse peu aux problèmes posés par les applications : Think n'offre pas de mécanisme pour adapter les applications qui s'exécutent.

Les exo-noyaux offrent des mécanismes pour adapter dynamiquement un noyau d'un système d'exploitation mais ne couvrent pas les aspects langages : il n'existe pas de mécanisme générique pour construire le langage associé au domaine implanté par un composant ou une bibliothèque système. Les applications actives présentées dans cette thèse construisent elles aussi leur environnement d'exécution à l'aide de composants adaptables, mais y ajoutent un langage pour le manipuler.

1.3 Réseaux actifs

Les réseaux actifs [TW96, Cal99] sont des réseaux permettant à des paquets de transporter du code. Ce code est exécuté sur les noeuds du réseau et adapte donc le comportement du réseau en fonction des besoins spécifiques d'une application ou d'un protocole. Les réseaux actifs apportent une réponse au problème d'évolutivité des réseaux : les paquets agissent sur les noeuds et peuvent donc faire évoluer le comportement du réseau. L'approche adoptée dans les réseaux actifs reste spécifique au domaine des réseaux : un paquet actif ne peut que modifier la manière dont un paquet est transporté par le réseau. Les réseaux actifs ne se préoccupent pas d'adapter l'ensemble de l'environnement d'exécution en fonction des besoins du paquet. Les applications actives présentées dans cette thèse généralisent les réseaux actifs à tout l'environnement d'exécution. Les réseaux actifs masquent les couches sous-jacentes (le système et l'environnement d'exécution, voir la section 4.1 du chapitre 2) et limitent donc les possibilités des réseaux actifs. Nos applications actives lèvent cette contrainte puisque l'environnement d'exécution complet est construit par l'application active.

1.4 Réflexion

La réflexion [Smi82, Mae87b, Mae87a] permet à une application de se manipuler elle-même de la même façon qu'elle manipule son principal sujet. La réflexion est largement utilisée dans nos travaux pour qu'une application active puisse modifier son environnement d'exécution. En effet, une application active agit sur l'environnement pour le modifier, or une application active contient une partie de cet environnement. Elle se manipule donc elle-même de la même façon qu'elle manipule l'environnement d'exécution. Dans nos travaux, la réflexion est l'outil permettant aux applications actives d'agir sur l'environnement. La réflexion est plus largement présentée dans la section 3.1 du chapitre 2.

1.5 Environnements dédiés et surcouches logicielles

Une solution parallèle existe déjà pour éviter de construire des environnements dédiés incompatibles avec les environnements d'exécution standards : les surcouches logicielles. Une surcouche logicielle est une bibliothèque émulant le comportement dédié requis par l'application. Elle n'introduit donc pas d'incompatibilité avec l'existant.

Une surcouche logicielle reste toutefois limitée par les possibilités de l'environnement d'exécution sur lequel elle repose : le mécanisme dédié peut très bien ne pas pouvoir être émulé. Par exemple, une machine virtuelle Java standard n'offre pas les abstractions nécessaires pour manipuler la politique de gestion des tâches. Il n'est donc pas possible de créer une surcouche logicielle pour offrir à une application Java une politique de gestion de tâche temps réel. Pour remédier à ce problème, une spécification Java temps réel [BBD⁺00]

incompatible avec la spécification standard de la machine virtuelle Java [GJSB00] a vu le jour.

Les surcouches logicielles sont aussi obligées de contourner les limites de l'environnement sous-jacents ce qui conduit à des implantations peu optimisées. Par exemple, le tissage d'aspects [KLM⁺97, PRS04] (voir la section 3.2 du chapitre 2) géré au niveau applicatif dans une machine virtuelle Java, comme dans Jac [Objb, jaca], introduit une indirection pour accéder au code de l'aspect, ce qui ralentit l'exécution par rapport à une approche dédiée comme Steamloom [BHMO04]. De plus, la machine virtuelle dédiées permet de tisser dynamiquement un aspect en tout point du programme, ce que ne peut pas permettre une surcouche logicielle.

Malgré ces contraintes, les concepteurs préfèrent se tourner vers des surcouches logicielles, lorsque c'est possible, pour construire des applications portables respectant les normes. En effet, il vaut mieux payer un surcoût en temps d'exécution que de prendre le risque d'écrire une application pour un environnement qui ne possède pas de norme ou de spécification suffisantes pour être utilisé à large échelle.

Les applications actives réunissent les qualités des environnements d'exécution dédiés et des surcouches logicielles :

- le mécanisme dédié mis en œuvre par une application active est aussi performant que le même mécanisme dans un environnement dédié car aucune indirection n'est nécessaire pour accéder au mécanisme ;
- une application active n'est pas soumise aux limites de l'environnement sous-jacent car elle plante le mécanisme dédié directement dans l'environnement d'exécution.
- une application active, comme dans les réseaux actifs, est portable car elle n'a pas besoin d'environnement dédié pour s'exécuter ;

2 Construction des applications actives

Pour construire des applications actives, il faut d'abord répondre à deux questions :

- Comment sont conçues les applications actives ?
- Comment est conçu l'environnement d'exécution des applications actives ?

2.1 Conception des applications actives

La construction des applications actives nécessite la présence d'un environnement d'exécution pour applications actives. En effet, une application active construit son environnement en fonction de ses besoins, mais elle a elle-même besoin d'un support d'exécution. Cet environnement d'exécution pour application active est appelé la micro machine virtuelle (μvm). Une application active est chargée dans la μvm et transforme la μvm en l'environnement nécessaire à l'exécution de l'application active.

Une application active est alors constituée de deux parties : la partie dite active, chargée dans la μvm pour construire l'environnement d'exécution et la partie dite passive, exécutée par l'environnement d'exécution. La partie passive peut être vue comme le code fonctionnel de l'application. La partie active correspond alors au code non fonctionnel. La partie passive d'une application active correspond à une application connue et standard (par exemple, une application Java ou un document PostScript) : les applications actives adjoignent aux applications actives du code actif qui enrichie la sémantique de l'application.

La partie active d'une application active doit être constituée de briques de base : de cette façon, les briques de base peuvent être *réutilisées* et *composées* par différentes applications actives, ce qui diminue l'effort de développement à effectuer pour construire une nouvelle application active. Ces briques de base sont appelées des *MVLet*, ce sont des spécifications d'environnement d'exécution (des machines virtuelles). Par exemple, une partie active constituée d'une machine virtuelle Java à tissage d'aspects est *composée* d'une MVLet machine virtuelle Java standard et d'une MVLet de spécialisation aspect. La MVLet machine virtuelle Java peut alors être *réutilisée* par une autre application active pour construire une machine virtuelle Java temps réel.

Les MVLets sont chargées les unes après les autres dans la μvm . Chaque chargement de MVLet construit un environnement d'exécution intermédiaire. Ces environnements sont appelés des *Machines Virtuelles Virtuelles* (MVVs) : ce sont des environnements d'exécution (des machines virtuelles) virtualisés dans le sens où ils peuvent être spécialisés. La μvm est aussi une machine virtuelle virtuelle dans laquelle aucune MVLet n'a été chargée. Une MVLet est chargée dans une MVV pendant qu'elle s'exécute. Elle peut donc être chargée de la même façon au démarrage de l'application active ou pendant son exécution : il n'y a pas de rupture entre configuration et reconfiguration.

2.2 Conception de la μvm

Une des grandes difficultés de ces travaux réside dans la construction de la μvm . En effet, la μvm est un environnement d'exécution pour applications actives. Comme tout environnement d'exécution, la μvm impose des mécanismes particuliers qui peuvent ne pas convenir à la construction de toutes les applications actives. La μvm doit donc être adaptable : chaque mécanisme mis en œuvre dans la μvm doit être accessible au niveau applicatif et modifiable par une application active. De manière à permettre la construction des applications actives, la μvm doit :

- Offrir suffisamment d'abstractions pour étendre et modifier son comportement. L'application doit pouvoir construire l'environnement d'exécution exact dont elle a besoin.
- Offrir des mécanismes de chargement de code pour étendre son comportement et devenir l'environnement d'exécution définie par l'application active.
- Définir un minimum de mécanismes pour ne pas les imposer à une application active. Si la μvm impose des mécanismes de trop haut niveau elle va limiter les possibilités des applications actives comme le font les environnements d'exécution actuels.

A chacun de ses points correspond une des propriétés de la μvm .

- La μvm est environnement totalement réflexif : ainsi, elle offre les mécanismes nécessaires à son adaptation.
- La μvm est un compilateur dynamique et un éditeur de lien avec des bibliothèques systèmes : ainsi, elle permet le chargement de code sous forme de source ou sous forme binaire. Le chargement de source permet d'insérer du code avec un haut niveau sémantique et le chargement de code binaire permet d'insérer du code provenant de bibliothèques existantes. De plus, une application active peut enrichir l'ensemble des mots clés du langage source pour permettre la construction de langages spécifiques (DSL) associés au domaine construit par l'application active.
- La μvm est minimale en fonctionnalité, ainsi elle impose un minimum de contrainte.

L'environnement d'exécution des applications actives, la μvm , est donc un compilateur dynamique et un éditeur de liens réflexif et minimal. La réflexion repose sur une structure

à base de composants réflexifs et sur un ensemble de méthodes permettant de manipuler ces composants (un protocole méta-objet, MOP). La conception de ce protocole méta-objet représente une part importante de notre travail. La μvm est étudiée en profondeur dans le chapitre 3. Elle a fait l'objet d'un prototypage en C++ sous linux, présentée dans le même chapitre, et d'une étude de performance présentée dans le chapitre 5.

2.3 Niveaux d'adaptation

Le but des applications actives est de construire des environnements dédiés sans perdre en portabilité. Elles doivent donc adapter différents niveaux de l'environnement d'exécution : la μvm , le langage, l'environnement d'exécution et l'application passive.

Adaptation de la μvm . Malgré sa minimalité, la μvm impose un certain nombre de mécanismes : une gestion mémoire, une gestion des tâches, une représentation intermédiaire des éléments qu'elle compile, un lexeur (analyseur lexical) et un parseur (analyseur syntaxique). Les deux premiers définissent un modèle d'exécution et les trois suivants un langage. Les choix de conception effectués lors du développement de ses différents éléments ne peuvent pas convenir à toutes les applications : le modèle d'exécution, comme tout mécanisme système, fait des hypothèses sur les applications exécutées et effectue des choix pour l'application (modèle objet, mémoire partagée etc...). Comme les applications actives peuvent spécialiser leur environnement d'exécution en fonction de leurs besoins, elles doivent aussi pouvoir spécialiser ces aspects.

La couche réflexive de la μvm est utilisée par une application active pour modifier les mécanismes systèmes. Une application active peut donc modifier le comportement de la μvm en fonction de ses besoins.

Adaptation du langage. Les enseignements des Domain Specific Language [vDKV00] nous montrent qu'un langage bien adapté à un problème particulier permet de masquer la complexité du domaine, de faciliter sa mise en œuvre et de vérifier plus facilement la cohérence du programme. Une application active doit donc pouvoir modifier le langage de la μvm pour construire son propre langage en fonction de ses besoins et du format de l'application passive. Ainsi, une application active réutilise la chaîne de compilation mise en place dans la μvm pour la spécialiser en fonction de l'application passive manipulée par la partie passive. Enrichir ou modifier la sémantique d'un langage permet de définir de nouveaux mots clés adaptés au domaine, alors que changer la syntaxe d'un langage permet de changer la manière dont sont écrits les programmes.

Pour modifier le langage d'entrée de la μvm , la réflexion des composants internes de la μvm est exploitée pour modifier le comportement du lexeur et du parseur. Un mécanisme annexe permettant de modifier, ajouter ou retirer des mots clés au langage de la μvm est utilisé pour définir de nouvelles macro-instructions. Ces macro-instructions sont assimilables aux mots clés d'un DSL.

Adaptation de l'environnement d'exécution. Le but des applications actives est de construire des environnements d'exécution dédiés tout en restant portable. Pour construire des environnements d'exécution dédiés, il faut pouvoir modifier l'environnement d'exécution. Modifier un environnement d'exécution revient à modifier une MVV : une MVV doit donc

offrir suffisamment de mécanismes de réflexion pour permettre à une MVLet de la modifier en profondeur.

La couche réflexive mise en place dans la μvm est étendue aux MVLets : en réutilisant les composants réflexifs de la μvm pour construire des MVLets, une unique abstraction réflexive est à appréhender pour le concepteur d'applications actives.

Adaptation de l'application passive L'application passive (ce qui est exécuté par une machine virtuelle virtuelle) ne doit pas être figée pour deux raisons principales :

- elle doit pouvoir être adaptée à son contexte d'exécution (au démarrage ou dynamiquement) ;
- plutôt que de redémarrer l'application à chaque modification, il est souhaitable de pouvoir adapter dynamiquement l'application passive, en particulier pour les applications critiques ou commerciales.

L'adaptation de l'application passive n'est pas toujours possible : il faut que l'application passive possède des informations sur sa structure, comme les symboles qu'elle manipule. Lorsque c'est possible, nous montrerons dans ces travaux comment réutiliser la couche réflexive mise en place dans la μvm au niveau de l'application passive.

3 Plan de la thèse

L'objectif de la thèse est de concevoir et de valider la notion d'application active, en particulier :

- de synthétiser les différentes approches visant à construire des environnements adaptables ;
- de construire l'environnement d'exécution des applications actives, la μvm ;
- de construire deux applications actives principales, une permettant de modifier la sémantique d'un document, l'autre permettant de construire des machines virtuelles Java dédiées portables et performantes ;
- de présenter quatre adaptations de ces deux applications actives pour :
 - ajouter de nouveaux langages,
 - modifier le modèle mémoire et le modèle de tâche,
 - adapter dynamiquement l'application à l'aide d'un tisseur d'aspects,
 - migrer des applications ;
- de montrer que les applications actives répondent bien au problème posé, la construction d'environnements dédiés sans introduire d'hétérogénéité.

3.1 Adaptation dynamique

Le chapitre 2 présente un tour d'horizon de la recherche sur l'adaptabilité. Dans la première section, la recherche sur la flexibilité dans les noyaux des systèmes d'exploitation et dans les intergiciels est examinée. Les différents travaux présentés dans cette section montrent principalement quelle est l'utilité de l'adaptabilité dynamique et quels sont les mécanismes utilisés pour la mettre en œuvre.

La suite du chapitre décrits quatre axes de recherche sur l'adaptabilité : la réflexion, le tissage d'aspects, les réseaux actifs et les Domain Specific Language. Les deux premiers sont introduits pour leur utilité dans les applications actives, mais aussi pour montrer la limite

des surcouches logicielles. Les deux études suivantes dégagent les principes des réseaux actifs et des Domain Specific Language et montrent comment ces principes nous ont servis dans nos travaux.

3.2 Architecture de Machines Virtuelles Virtuelles pour les Applications Actives

Le chapitre 3 présente le cœur du travail réalisé pendant cette thèse : la conception des applications actives et la formalisation de la notion de machine virtuelle virtuelle. Ce chapitre décrit aussi en détail la μvm et son ancêtre, la Ynvm .

La Ynvm est un compilateur dynamique offrant une réflexivité au niveau du langage. Elle nous a servi de base pour construire la μvm , l'environnement d'exécution des applications actives. La μvm enrichie la Ynvm avec une couche réflexive à base de composants : les différents composants constituant la μvm sont accessibles au niveau applicatif, ce qui permet de :

- modifier le comportement de la μvm en fonction des besoins d'une application active ;
- réutiliser la couche réflexive dans les MVlets et ainsi, de pouvoir modifier dynamiquement toute Machine Virtuelle Virtuelle.

La suite de la thèse présente différentes applications actives validant l'approche.

3.3 Les documents actifs

Le chapitre 4 présente une première série d'applications actives basées sur des documents, appelés des documents actifs. Un document actif est document embarquant avec lui du code exécutable (une MVlet) lui permettant d'adapter son contenu, son programme de visualisation et le système d'exploitation sous-jacent en fonction de ses besoins.

Les documents actifs apportent une réponse aux problèmes de la composition de documents multimédia, de la gestion de la qualité de service, des droits d'auteur, de l'hétérogénéité des supports de visualisation et de cohérence. Seuls les deux premiers aspects ont fait l'objet d'un prototype.

La composition de documents multimédias est réalisée en utilisant les programmes de visualisation normaux des documents (*Ghostsript* pour du PostScript, *Mplayer* pour des vidéos etc...) et en composant ses exécutables avec les mécanismes traditionnels des systèmes d'exploitation (tube, signaux, messages à des fenêtres). La composition de documents multimédias montre comment enrichir la sémantique d'un documents sans avoir à toucher ni au contenu du document, ni à son programme de visualisation par défaut, en ajoutant du code actif à un document passif.

La gestion de la qualité de service est étudiée du point de vue de l'ordonnancement du programme de visualisation d'un document dans un système. Ce prototype repose sur Bossa [LMB02], un DSL pour ordonnanceurs développé à l'école des mines de Nantes. Le document actif spécifie ses besoins en terme d'ordonnancement à l'aide des mécanismes mis en œuvre dans Bossa. Nous montrons aussi comment composer les deux premiers travaux pour obtenir des documents multimédia et auto-ordonnés.

Cette première étude montre comment appliquer la notion d'application active au cas particulier des documents et quels sont les avantages apportés par l'approche. En adjoignant du code exécutable aux documents, les limites traditionnelles des documents sont levées.

3.4 Construction d'une machine virtuelle Java sous forme de composants adaptables : la JnJVM

La prolifération d'environnement d'exécution dédiés et incompatibles entre eux touche particulièrement le monde Java. Les machines virtuelles Java sont monolithiques et masquent le système sous-jacent : lorsqu'une application a un besoin spécifique, comme du tissage d'aspects ou une politique mémoire particulière, elle doit soit avoir recours à une machine virtuelle dédiée, soit à une surcouche logicielle peu performante.

Pour implanter des mécanismes dédiés dans les machines virtuelles Java sans perdre ni en performance, ni en portabilité, nous présentons dans le chapitre 5 une MVLet de machine virtuelle Java. Cette MVLet a été développée en réutilisant la couche réflexive de la μvm , ce qui permet à une application active de la spécialiser. Cette MVLet Java est appelée la JnJVM (pour JnJVM is not a Java Virtual Machine¹). Elle offre des performances équivalentes à la première version de la machine virtuelle Java de Sun à compilateur à la volée (la HotSpot 1.0 avec JIT).

Le chapitre 5 présente la problématique des environnements Java dédiés, le prototype de machine virtuelle Java (la JnJVM), et les performances de ce prototype. L'étude de performances de la JnJVM mesure aussi celle de la μvm . Ce chapitre montre qu'il est possible de créer un environnement d'exécution Java flexible et performant sans perdre en portabilité.

3.5 Évaluation qualitative de la JnJVM et de la μvm

Le chapitre 6 présente une étude qualitative de la JnJVM et de la μvm . Cette étude montre le degré de flexibilité de la couche réflexive présentée dans le chapitre 3. Elle est basée sur plusieurs adaptations dynamiques de la μvm et de la JnJVM répondant à des problèmes réels rencontrés dans les environnements d'exécution et plus particulièrement dans les machines virtuelles Java.

Quatre études sont présentées dans ce chapitre.

- La construction de deux nouveaux langages d'entrée dans la μvm : XML et un langage compact utilisé pour sérialiser des programmes sur un réseau. Ces deux langages ont été conçus pour permettre la réutilisation de la chaîne de compilation de la μvm et montre la flexibilité des composants internes de la μvm .
- L'adaptation du modèle mémoire et du modèle de tâches de la JnJVM. Ce travail est basée sur de l'analyse d'échappement [CGS⁺99, Bla03] donnant des renseignements sur la durée de vie des objets Java. La MVLet présentée dans ces travaux montre comment modifier le comportement du ramasse-miettes et la gestion de la synchronisation sur des objets en fonction de leur durée de vie et de leur utilisation multi-tâches. Nos travaux montre des gains en vitesse allant jusqu'à 24% sur la gestion mémoire et jusqu'à 57% sur la synchronisation. Cette étude montre la flexibilité des mécanismes systèmes de la JnJVM.
- Une étude du tissage d'aspects : la définition d'un langage de tissage d'aspects (un DSL aspect) et son utilisation dans la JnJVM. Ces travaux montrent comment adapter une application passive sans avoir à la redémarrer, ce qui nous a permis de coupler la JnJVM avec un outil de modélisation (Objectteering [obja]).
- Une étude de la migration de fil d'exécution. Cette étude vise à déplacer un fil d'exécution Java d'une machine source vers une machine cible tout en prenant en compte

¹La machine virtuelle Java est constituée de la μvm et de la JnJVM.

le contexte de l'application. Ces travaux présentent :

- une gestion de la migration générique au niveau de la machine virtuelle Java sans introduire d'hétérogénéité ;
- une gestion de la migration en fonction du contexte d'une application particulière (un client de messagerie d'un protocole Jabber) tout en gardant un code applicatif portable, écrit indépendamment de la migration.

Ces différentes études mettent en avant les quatre niveaux d'adaptation présentés dans la section 2.3. Elles montrent l'utilité des applications actives, leur facilité de mise en œuvre et le haut degré d'adaptabilité de la μvm et de la JnJVM.

3.6 Conclusions et perspectives

Le chapitre 7 conclut cette thèse et montre qu'il est possible de concevoir des environnements dédiés et performants sans pour autant introduire d'hétérogénéité à l'aide des applications actives. L'ensemble des travaux présentés dans le chapitre 6 et dans le chapitre 4 couvre un large spectre de domaines et montre la reproductibilité des applications actives à de nombreux autres domaines informatiques.

Les applications actives ne diminuent pas pour autant le travail d'ingénierie : la JnJVM construit une machine virtuelle Java complète faisant environ quinze mille lignes de code et le travail effectué pour mettre en œuvre les MVLet de spécialisation Java présentées dans le chapitre 6 est équivalent au travail à effectuer pour implanter ces mécanismes dédiés dans des machines virtuelles Java standards. Les applications actives apportent une réponse au problème de l'évolutivité logicielle et permettent de composer des mécanismes dédiés, elles n'apportent en aucune façon de solution au problème réel posé par le mécanisme dédié lui-même.

4 Le projet Machines Virtuelles Virtuelles

Ces travaux de thèse ont été réalisés au sein du thème Système Répartis et Coopératifs (SRC) du Laboratoire d'Informatique de Paris VI (LIP6) dans le cadre du projet Machine Virtuelle Virtuelle [FPR97, FPR98] (MVV) dirigé par B. Folliot depuis 1997. Le projet MVV vise à étudier de manière systématique la flexibilité dans les environnements d'exécution, soit pour augmenter les performances des applications, soit pour suivre l'évolution logicielle.

La figure 1.1 présente l'ensemble des travaux menés dans le projet Machine Virtuelle Virtuelle. Les principaux travaux que nous avons réalisés dans le cadre de cette thèse sont indiqués dans les cases grisées. La μvm est décrite dans le chapitre 3, les documents actifs et les documents auto-ordonnés dans le chapitre 4, la JnJVM dans le chapitre 5 et les trois Machines Virtuelles Java dans le chapitre 6.

Tous les travaux du projet Machine Virtuelle Virtuelle ont pour point commun une étude systématique de la flexibilité dynamique dans les environnements d'exécution et dans les applications. Ils se séparent en trois grandes familles.

- Réutilisation d'environnements existants et d'applications existantes : les documents actifs et la CVM utilisent des applications existantes (des documents existants, des applications pour intergiciels standards) et des environnements d'exécution existants (ghostscript, mplayer, OpenCCM). Ces travaux montrent comment adapter ou enrichir des environnements à l'aide de code additionnel.

- Construction d’environnements flexibles et réutilisation d’applications existantes : la JnJVM, les trois machines virtuelles Java et les travaux sur le redéploiement de servlet montrent comment profiter d’environnements flexibles pour implanter des mécanismes dédiés, mais en réutilisant des applications existantes.
- Construction d’environnements flexibles et construction de nouvelles applications : FlexORB, C/NN et C/Span montrent les bénéfices d’environnements d’exécutions flexibles et d’applications flexibles pour optimiser l’usage des ressources matérielles. Ces trois travaux ont été présentés notamment dans la thèse de F. Ogel [Oge04].

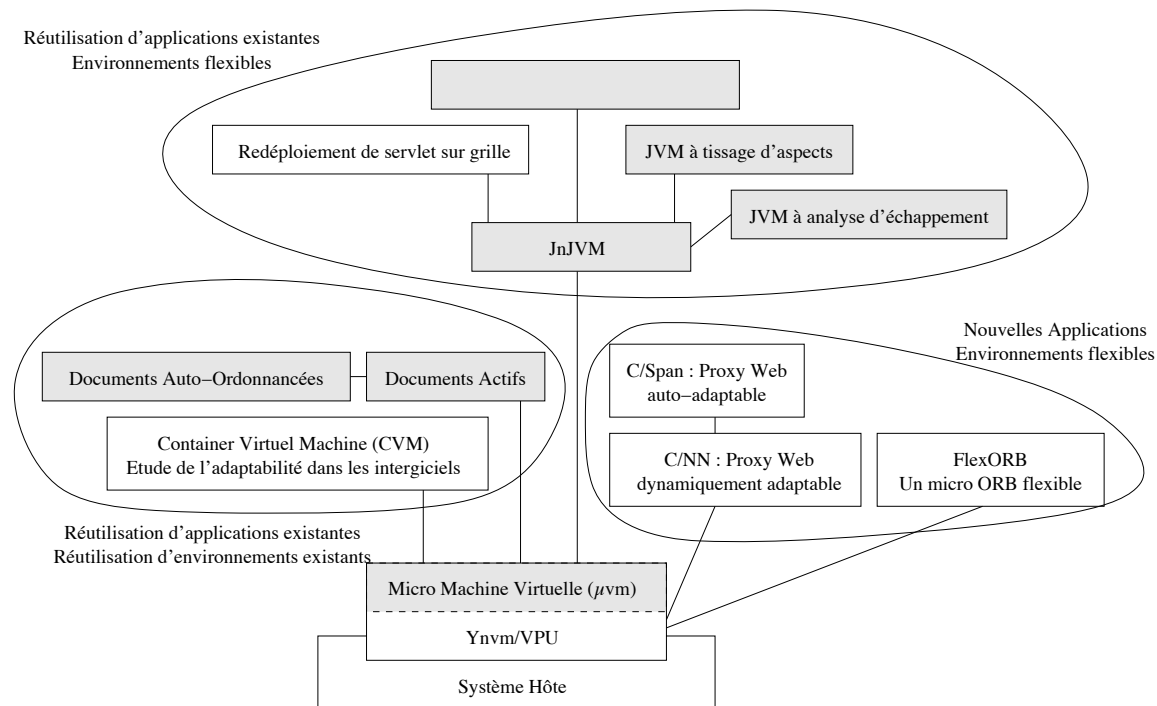


FIG. 1.1 – Ensemble des travaux menés dans le projet Machine Virtuelle Virtuelle

Les travaux de F. Ogel présentent deux instances de la Ynvm, l'ancêtre de la μvm , une pour Think [FSLM02, SCS02], un exo-noyau développé par France Telecom R/D, une autre embarquée dans un noyau linux. Ses travaux s'intéressent surtout à la construction d'environnements spécifiques et montrent les avantages d'une architecture dynamiquement flexible, mais ne s'intéresse pas à l'intégration d'applications existantes dans des environnements flexibles. Les applications actives présentées dans cette thèse généralisent l'approche à des applications préexistantes (des applications Java, des documents standards) et montrent qu'en travaillant au niveau de l'environnement d'exécution, il est possible d'adapter des applications tierces : les applications actives présentées dans cette thèse s'intègrent avec l'existant. La création de la couche réflexive dans la μvm effectuée dans cette thèse unifie aussi les mécanismes flexibles des différents travaux menés dans le projet Machine Virtuelle Virtuelle. En utilisant un mécanisme réflexif unique, les développeurs de MVLets n'ont qu'un seul paradigme à appréhender, ce qui évite des solutions incompatibles entre elles au sein même du projet.

Ces travaux de thèse ont été subventionnés par une bourse MESR entre 2001 et 2004 et complétés par un poste d'Attaché Temporaire d'Enseignement et de Recherche (ATER) entre 2004 et 2005. Ils ont donné lieu à plusieurs publications. La Ynm a été présentée dans [OTGF05, OTP⁺03, OTF05], la JnJVM et la μvm dans [TOG⁺05, TFO03, OTFP03] et les documents actifs dans [TFP02]. Des travaux connexes ont été menés sur les intergiciels, publiés dans [OFT04, HMT⁺04], et sur l'étude du membership, publié dans [FT01].

CHAPITRE 2

Adaptabilité dynamique

Sommaire

1	Introduction	13
2	Environnements flexibles	13
3	Outils dédiés à l'adaptation dynamique	23
4	Domaines connexes à l'adaptabilité : réseau et langage	35
5	Les challenges de l'adaptabilité	39

1 Introduction

Ce chapitre présente une étude générale de l'adaptabilité dynamique, c'est-à-dire de la possibilité pour une application, un environnement d'exécution ou un noyau de système d'exploitation de modifier, paramétrer ou étendre son comportement. Dans la première section, des travaux de recherche utilisant des mécanismes flexibles sont décrits. Cette première étude met en avant les besoins en adaptation dans les systèmes et les intergiciels. La réflexivité et le tissage d'aspects, deux outils permettant la construction d'environnements adaptables dynamiquement, sont présentés dans la section 3. Les prototypes présentés dans cette section montrent qu'il est difficile de construire un environnement adaptable dynamiquement et performant tout en restant homogène avec les environnements d'exécution existants. La section 4 présente deux domaines connexes de recherche qui ont fortement influencé l'élaboration de cette thèse, les réseaux actifs et les Domain Specific Language (DSL). Enfin, la section 5 récapitule et conclut les différents travaux décrits dans ce chapitre.

2 Environnements flexibles

Les environnements d'exécution proposent des abstractions de haut niveau aux applications. Ces abstractions offrent des interfaces clairement définies : elles donnent un cadre de programmation aux développeurs. Par exemple, la norme Posix offre une API de haut niveau pour la gestion des tâches sous la forme des thread Posix, les IPC System V (sous Linux) sont une norme pour faire communiquer les processus d'une machine et la spécification

Java [LY, GJSB00] offre un langage et un format de programme exécutable indépendant de la machine et du système d'exploitation sur lequel est exécuté l'application.

Les choix effectués lors de la conception d'un environnement ne peuvent pas être adaptés à toutes les applications. En effet, certaines applications ont des besoins particuliers qu'un environnement ne peut pas prendre en compte. Pour palier à ce problème, de nombreux travaux de recherche tentent de rendre les environnements d'exécution plus flexibles. Par exemple, les travaux sur les exo-noyaux [EKO95], le chargement de modules dans les noyaux monolithiques ou la notion de POA (voir la section 2.2) dans les intergiciels offrent des mécanismes pour spécialiser ou étendre dynamiquement un environnement d'exécution en fonction des besoins d'une application. Ces systèmes adaptables sont ce que nous appelons les *systèmes ou environnements d'exécution flexibles*.

La sous-section 2.1 décrit une série de travaux de recherche utilisant la flexibilité dans les systèmes d'exploitation et montre que la flexibilité est principalement utilisée pour améliorer l'utilisation des ressources matérielles. La section 2.2 présente les intergiciels et met en avant les mécanismes flexibles utilisés pour adapter dynamiquement ces environnements.

2.1 Utilisation de la flexibilité dans les environnements

Dans cette sous-section, une série de travaux de recherche utilisant la flexibilité est présentée. Ces travaux n'apportent pas de cadre global à l'adaptabilité dynamique, comme les exo-noyaux ou la réflexion, mais montrent quels sont les besoins en flexibilité dans les systèmes et comment la flexibilité est mise en œuvre. Les travaux suivant se séparent en deux grandes familles :

- les cas où la plate-forme d'exécution n'offre pas les mécanismes ou l'interface adéquats à une application ;
- les cas où le contexte d'exécution de l'application ne peut pas être prévu lors de la conception de l'application.

2.1.1 Inadéquation de l'environnement à l'application

Cache d'entrées-sorties applicatif. Cao, Felten et Li ont montré dans leurs travaux sur les caches d'entrées-sorties [CFL94] que la politique utilisée dans un système n'était pas forcément adaptée à toutes les applications. Le cache des blocs d'entrée/sortie¹ a été séparé en deux parties distinctes : une partie gérée par le système de manière standard (un LRU, Least Recently Used) et l'autre déléguée à l'application via un appel à une fonction utilisateur (un *upcall*).

Ce cache a été développé à partir d'un noyau monolithique Ultrix pour des systèmes de fichiers NFS et UFS. Le noyau gère les allocations de blocs et implante la politique LRU. Lorsqu'il n'y a plus de bloc disponible, le bloc le moins récemment utilisé, B, est enlevé de la liste des blocs utilisés par le noyau. Le noyau trouve ensuite quel processus gère le bloc B et lui laisse la possibilité de le remplacer par un autre, C, en faisant un appel en mode utilisateur. Le noyau libère ensuite le bloc C. Cet algorithme offre donc une opportunité au processus de modifier la politique de remplacement de bloc du noyau. Une amélioration de l'algorithme est donnée dans [CFKL96] en permettant à l'application de modifier quel bloc sera chargé en avance lors d'une lecture.

¹Le buffer-cache.

Les résultats de cette étude montrent qu'une politique bien adaptée à une application peut augmenter ses performances de façon considérable. Les performances présentées dans l'article montrent une réduction des erreurs de cache (cache-miss) allant de 10 à 80% ce qui signifie une augmentation de la vitesse d'exécution pouvant aller jusqu'à 45%. Cet exemple illustre parfaitement les bénéfices que peut tirer une application d'un contrôle fin des politiques de l'environnement sous-jacent.

Primitives de gestion mémoire. Le rôle traditionnel de la mémoire virtuelle est d'augmenter l'espace mémoire qu'une application peut adresser, en ne gardant en mémoire physique que les pages les plus souvent utilisées. Dans [AL91], Apel et Li montrent que la mémoire virtuelle peut aussi servir à implanter d'autres algorithmes, comme une mémoire partagée répartie [CF89], un ramasse-miettes temps-réel par recopie [HB78] ou un ramasse-miettes générationnel [LH83].

L'article montre que l'efficacité des algorithmes dépend de la taille des pages : chaque algorithme a une taille de page optimale. Les choix effectués lors de la conception du système vont donc avoir des conséquences néfastes sur les performances de certaines applications. Les trois exemples cités marchent bien avec des pages de petite taille.

Apel et Li isolent six primitives que doit implanter le système pour pouvoir construire ces différents algorithmes. Parmi celles-ci, la primitive `DIRTY` renvoie la liste des pages qui ont été accédées en écriture depuis le dernier appel. Cette primitive permet d'optimiser le ramasse-miettes générationnel [LH83] basé sur l'algorithme de Baker [HB78] : la primitive `DIRTY` peut être utilisée pour connaître les pages où résident des objets ayant été mutés par une autre tâche (le mutateur) pendant que le collecteur effectuait son travail. Le collecteur doit alors parcourir de nouveau ces objets mutés. Bien que les processeurs récents marquent les pages accédées en écriture (voir [Sha96] page 240 pour le Pentium, par exemple), les systèmes Posix et les IPC System V n'implément pas cette primitive : il n'est donc pas possible d'utiliser dans les meilleures conditions l'algorithme de Baker. Cet exemple illustre les difficultés que peut avoir une application lorsque l'environnement dans lequel elle s'exécute n'offre pas une interface suffisante.

Gestion mémoire au niveau applicatif. Un autre exemple de besoin en adaptation est donné par la gestion de la mémoire dans le système. Dans [HC92], Hart et Cheriton présentent une architecture de gestion de pages au niveau utilisateur. Cet exemple est construit dans le noyau V++. Les segments mémoires [Org72] sont associés à un manager applicatif qui s'occupe de gérer la stratégie de mémoire virtuelle à utiliser. Avec cette architecture, le manager peut contrôler la taille de la mémoire physique allouée, la taille des pages, faire de l'observation et gérer localement l'algorithme d'échange ("swap") pour décharger ou charger des pages en fonction des besoins de l'application.

Des travaux proches ont été menés par Krueger et Al. [KLVA93] pour permettre à une application de gérer plus finement sa mémoire. La grande différence entre ces travaux et les précédents est la facilité d'utilisation : un gestionnaire mémoire est un ensemble de classes C++. Pour implanter une nouvelle politique, il suffit d'hériter du gestionnaire par défaut et de surcharger les méthodes qui doivent être spécialisées. En plus de cette gestion mémoire, [KLVA93] présente un outil pour observer comment une application se comporte vis à vis de sa gestion mémoire de manière à en optimiser l'utilisation.

D'autres besoins en flexibilité. De nombreux autres travaux montrent qu'une politique système mal adaptée à l'application peut entraîner de grandes pertes de performances. Plusieurs cas concrets sont cités dans l'article fondateur des exo-noyaux [EKO95], en particulier [Sto81] qui montre qu'un système de fichier inapproprié peut avoir un grand impact sur le fonctionnement d'une base de données, ou [TL94] qui montre qu'une gestion quasi-synchrone des exceptions peut augmenter les performances du système.

Les différents exemples présentés dans cette sous-section montrent le bénéfice que peut tirer une application de la flexibilité du système d'exploitation : avec une gestion des ressources bien adaptée, l'application augmente ses performances.

2.1.2 Inadéquation de l'application à son contexte

Outre l'inadéquation du système à l'application, de nombreux besoins en adaptation proviennent de la non prédictibilité du contexte d'exécution de l'application ou simplement des faibles ressources matérielles dont va disposer l'application.

Ubiquité numérique et contraintes matérielles. L'ubiquité numérique [Wei93] (l'informatique omniprésente) est un domaine informatique où le contexte d'exécution ne peut pas être prévu lors de la conception des environnements ou des applications. La manière de gérer l'énergie, les protocoles réseaux adéquats ou encore les interactions entre les noeuds ne peuvent être connus que lors de l'exécution : les applications et les environnements d'exécution doivent donc offrir des mécanismes pour adapter dynamiquement leur comportement. De plus, les machines à faible ressource matérielle comme les PDA (Personal Digital Assistant) ou les cartes à puce ne peuvent pas embarquer tout le code nécessaire à leur bon fonctionnement. Une collaboration entre des machines ayant plus de puissance et ces objets à faible ressource est alors nécessaire : les applications doivent être capable de migrer dynamiquement des traitements sur des machines à forte capacité.

Le projet Distributed Virtual Machine [SGGB99] (DVM) présente une Machine Virtuelle Java distribuée sur un réseau. Un serveur centralisé (un "firewall" ou un routeur, par exemple) s'occupe de vérifier le bytecode, de le compiler, de le sécuriser ou de l'optimiser. Cette machine envoie ensuite à des machines virtuelles Java clientes du code pré-vérifié ou pré-compilé. De cette façon, plusieurs machines clientes peuvent bénéficier du travail effectué par le serveur et ainsi factoriser ces quatre phases. La DVM peut aussi être utilisée pour éviter à des machines à faibles ressources d'être surchargées en effectuant le travail de compilation et de vérification sur des machines puissantes. Le flexibilité de la DVM est utilisée pour optimiser l'usage des ressources matérielles, en particulier le processeur.

Le micro-noyau Camille [Gri00] est un système pour cartes à puce. Camille utilise une représentation intermédiaire du code, Facade, qui doit être vérifiée avant d'être exécutée. Le code Facade est vérifié en dehors de la carte avec une technique de Proof Carrying Code [Nec97, RR98] (PCC) qui donne une preuve de la validité des types de données manipulées. Cette preuve est vérifiable dans la carte à puce pour un coût en mémoire beaucoup plus léger qu'une vérification complète. Les travaux de recherche visant à optimiser l'usage des ressources matérielles doivent reposer sur des environnements d'exécution flexibles pour intégrer facilement ces optimisations.

Gestion de l'énergie. Une autre utilisation de mécanismes flexibles est donnée par la gestion de l'énergie. Dans [Ell99], Ellis étudie une application simple qui positionne l'utilisateur

sur une carte à l'aide d'un GPS (Global Positioning System) et de cartes chargées à partir d'un serveur. Les différents états de l'application sont séparés en fonction du matériel utilisé et montrent qu'il n'est pas nécessaire d'alimenter simultanément tous les composants (par exemple, pendant la réception d'une carte, l'écran peut être mis en veille). En remontant les états d'énergie du système vers le niveau utilisateur, l'application peut mettre en mode d'économie d'énergie certains appareils lorsqu'ils ne sont pas utilisés et ainsi prolonger la durée de vie de la batterie. Cette étude montre que le comportement du système peut être nettement amélioré par les connaissances de l'application.

J. Flinn et M. Satyanarayanan [aMS99] étudient aussi un système pour prolonger la durée de vie de la batterie. Pour réduire la consommation d'énergie des applications, un outil de gestion de qualité de service est utilisé. Cet outil permet de diminuer l'usage des ressources pour augmenter la durée de vie de la batterie aux prix d'une dégradation de la qualité de service offerte à l'application. En adaptant l'usage des ressources de quatre applications (un lecteur vidéo, un outil de reconnaissance vocale, un afficheur de carte et un navigateur Web), les auteurs montrent qu'on peut espérer augmenter la durée de vie de la batterie d'un tiers en moyenne.

Cache Web. Les caches Web servent à rapprocher les documents des utilisateurs pour :

- diminuer le temps que met un utilisateur à obtenir une page ;
- diminuer la charge des serveurs en déléguant le travail aux caches.

De nombreux systèmes de cache existent (tel que Squid [Wes97]) mais leur configuration est difficile [MNR⁺98, PM98] (passage à l'échelle, adéquation de la politique utilisée). Le trafic sur internet est difficilement prévisible, en particulier lorsqu'un site devient à la mode pendant peu de temps [Sel96] (l'effet "hotspot of the week"). L'un des challenges est donc de pouvoir adapter dynamiquement le comportement des caches. C/Span [OPPF03] présente une architecture de cache Web dynamiquement adaptable. C/Span utilise un outil d'observation et modifie l'algorithme de gestion du cache en fonction des performances observées. Dans les caches Web, l'adaptation est nécessaire à cause de la non prédictibilité du contexte d'utilisation de l'application : les paramètres du systèmes évoluent au cours du temps.

Routage. Les routeurs transmettent les paquets de réseaux en réseaux. Le comportement des utilisateurs sur un réseau est, comme pour les caches Web, imprévisible (taille des files d'attente par exemple). De plus, l'intégration de nouveaux protocoles comme IPv6 ou le multicast IPv4 est long : les routeurs souffrent d'un manque d'adaptabilité. La section 4.1 décrit plus en détail la problématique de l'adaptation dans les réseaux.

2.1.3 Récapitulatif des besoins en flexibilité

Les différents travaux décrits dans cette sous-section mettent en avant des besoins en adaptation. De manière générale, une application a des besoins en adaptation pour deux raisons :

- l'environnement dans lequel elle s'exécute n'est pas approprié à ses besoins, que ce soit pour des problèmes de performances ou des problèmes d'absence de fonctionnalités proposées par l'environnement d'exécution ;
- l'application elle-même ne peut pas prévoir dans quel environnement elle va s'exécuter car elle ne peut pas prévoir le comportement des autres éléments interagissant avec elle.

La tableau 2.1 récapitule les différents besoins en adaptation. Le but poursuivi est souvent d'augmenter les performances d'exécution d'une application en fonction des ses contraintes (1, 2, 3, 4, 7). Les travaux 5 et 6 se concentrent uniquement sur la gestion d'autres ressources (batterie ou réseaux). Chaque adaptation revient à améliorer l'utilisation des ressources de l'environnement : processeur, mémoire, énergie, supports de stockage ou bande passante. Pour construire un environnement dédié à l'adaptation, ces contraintes doivent être prises en compte, en particulier, l'environnement doit offrir de bonnes performances et offrir des abstractions pour que l'application puisse gérer finement les ressources matérielles de sa plate-forme.

	Exemples	But poursuivi	Ressource optimisée	Mise en œuvre
1	Cache d'entrée/sortie au niveau applicatif	Diminuer l'utilisation du disque dur	Disque dur	Crochet dans le noyau + upcall modification de la gestion de blocs
2	Primitives de gestion mémoire	Augmenter la vitesse d'exécution	CPU	Primitives systèmes adaptées
3	Gestion mémoire au niveau applicatif	Diminuer l'utilisation du swap	Mémoire + Disque Dur	Crochet dans le noyau + upcall Modification de la gestion de pages
4	Cache Web	Diminuer le temps de réponse du cache Web	CPU + Mémoire + Disque Dur + Réseau	Observation + auto-adaptation
5	Gestion de l'énergie	Augmenter la durée de vie de la batterie	Batterie	Mise en veille au niveau applicatif
6	Routing	Déployer de nouveaux protocoles	Réseau	Réseaux actifs
7	Contraintes Matérielles	Optimiser la gestion des ressources	CPU + Mémoire	Pré-traitement sur des machines à fortes ressources

FIG. 2.1 – Les besoins en adaptation

2.2 Flexibilité dans les intergiciels

Le rôle des intergiciels est principalement de fournir des abstractions suffisantes pour masquer l'hétérogénéité matérielle de machines effectuant un travail en collaboration. Les intergiciels s'insèrent entre des systèmes (ou des machines virtuelles) et des applications. Ils offrent souvent un certain nombre de services systèmes (comme la persistance, les transactions ou le nommage) ce qui permet au développeur de ne s'occuper que de la partie fonctionnelle de son application. Les intergiciels sont une couche supplémentaire imposant de nouvelles contraintes aux applications : comme pour les systèmes d'exploitation, les intergiciels souffrent d'une certaine rigidité, bien que les besoins en flexibilité aient été pris en compte plus tôt lors de la conception des intergiciels.

2.2.1 Les intergiciels standards

Les intergiciels ont pour vocation de masquer l'hétérogénéité des plate-formes d'exécution. La plupart des intergiciels commerciaux implantent donc des standards ou des normes comme CORBA [Obj04] ou le Corba Component Model [Obj02] (CCM) de l'Object Management Group (OMG), .NET de microsoft, et les EJB ou JavaRMI de Sun Microsystems. Ces différents standards n'ont pas été conçus pour être flexibles. Toutefois, ils offrent des mécanismes d'adaptation assez puissants comme la notion d'intercepteurs ou celle de Portable Object Adaptator (POA) pour CORBA.

Les intercepteurs CORBA [Obj01] servent à insérer des crochets avant ou après le traitement d'une requête par un serveur ou un client. Cette technique permet à une application de modifier la requête avant qu'elle ne soit traitée ou d'allouer un serveur dédié pour cette requête. Les intercepteurs offrent à une application la possibilité de modifier les traitements associés à un serveur particulier. Le passage par un intercepteur a toutefois un coût non négligeable. Marchetti et al. [MVB01] ont étudié l'impact des intercepteurs dans trois ORB différents (JacORB v1_3_21 [jacb], ORBacus v.4 [ORBa] et Orbix 2000 [orbix]) et ont montré que la présence d'un intercepteur augmente le temps de latence de 1,4 à 10% et le temps de traitement d'une requête de 1,5 à 16%. Les meilleurs résultats sont obtenus avec Orbix et montrent que ce coût peut être ramené à des temps raisonnables. La notion d'intercepteurs est proche de la notion d'aspects (voir section 3.2) : ce mécanisme est suffisamment puissant pour permettre une adaptation en profondeur du serveur en modifiant les appels entrants et sortants.

Le POA CORBA ([Dan00], chapitre 7, 8 et 9) est une interface entre les objets et le bus à objets. Le POA peut être paramétré et permet à une application de spécifier des propriétés systèmes pour plusieurs serveurs simultanément. Le POA n'existe que chez le serveur et offre des mécanismes pour configurer la gestion des threads, le nommage interne des objets (Object ID) ou la persistance des références d'objets. Le POA permet aussi d'associer plusieurs identificateurs d'objets à un même serveur : un serveur peut donc implanter plusieurs interfaces simultanément.

Les intergiciels à composants forment la dernière génération d'intergiciels. Les serveurs sont construits sous la forme de composants qui peuvent être connectés entre eux. La grande innovation des intergiciels à composants est la notion de conteneurs. Ils découplent le code métier (fonctionnel) du code système (non fonctionnel) en interceptant les invocations. Les conteneurs offrent une grande flexibilité puisque, pour une même application métier, plusieurs codes systèmes peuvent être associés.

Fractal [BCS02, Sab04] est un modèle à composants hiérarchique : les composants sont soit primitifs, soit composites, soit partagés. L'architecture de Fractal repose sur la notion de contrôleur qui est équivalente à la notion de conteneur. Fractal peut être vu comme une plate-forme à composants réflexifs car elle offre des mécanismes d'introspection (les composants et leurs liaisons sont représentés explicitement) et un contrôle sur le cycle de vie des composants. Fractal offre aussi un système d'interception des invocations entrantes et sortantes des composants et une interposition d'un comportement de contrôle.

Les mécanismes d'adaptation offerts par les intergiciels standards restent rudimentaires : les crochets proposés par le POA sont peu nombreux, il n'existe pas de mécanisme pour modifier le comportement de l'intergiciel lui-même et il n'existe pas d'interface pour adapter les couches sous-jacentes à l'intergiciel. Ce manque de flexibilité des intergiciels standards a entraîné le développement de prototypes, décrits dans la sous-section suivante, offrant des mécanismes d'adaptation plus avancés.

2.2.2 Intergiciels flexibles

La construction d'intergiciels adaptables repose soit sur la modification d'intergiciels existants, soit sur la construction de nouveaux intergiciels.

DynamicTAO [KRL⁺00, KCM⁺00] (basé sur TAO [SC00]) est un environnement CORBA réflexif. DynamicTAO réifie les éléments internes de l'ORB sous la forme de composants appelés composants de configuration. Deux exemples d'adaptation dynamique sont présentés dans [KRL⁺00] : l'ajout d'un service de monitoring et l'ajout d'un service de sécurité à DynamicTAO. DynamicTAO garde une compatibilité avec les applications CORBA tout en offrant un haut degré d'adaptabilité. Une des difficultés que soulève ce projet est le problème de la cohérence lorsqu'une politique est remplacée par une autre : l'exemple donné est le remplacement d'une gestion de thread. Pour passer d'une politique de pool de thread à une nouvelle politique, il faut que l'ancienne politique indique quand son pool ne contient plus aucun thread actif. La politique originale doit donc être conçue de manière à pouvoir être remplacée : la construction d'une politique dans DynamicTAO ne peut pas être effectuée de manière transparente. Ce projet présente une approche visant à rendre le noyau de l'ORB flexible.

AspectIX [HBG⁺98] présente une architecture d'intergiciel basée sur le modèle d'objets à fragments [MGNS94]. Les fragments peuvent masquer la réplication d'un objet distribué, imposer des contraintes temps réelles au canal de communication, cacher des données de l'objet etc... Ces aspects non fonctionnels peuvent être configurés via une interface générique de l'objet. Chaque objet global (un ensemble de fragments) peut être configuré par un profil qui spécifie les aspects que doivent respecter les fragments. Quatre profils sont planifiés, en particulier un profil CORBA qui permet de faire interagir les objets AspectIX avec CORBA. Cette approche permet de séparer l'application de l'intergiciel dans lequel elle est déployée.

OpenORB [BCC⁺99] est une architecture flexible d'intergiciels à composants. L'architecture OpenORB répond au besoin d'adaptation de l'intergiciel lui-même pour, par exemple, offrir du temps réel pour des vidéos. OpenORB est basé sur la réflexion (voir section 3.1). Chaque objet du système est associé à un méta-espace qui en offre une représentation structurelle, mais aussi fonctionnelle. L'ORB est configuré ou re-configuré en utilisant le protocole Meta-Objet. Une implantation de cette architecture a été réalisée avec Oopp [ABE00].

JavaPOD [BR00], un modèle de composant proche des EJB, offre la possibilité d'attacher des propriétés non fonctionnelles aux composants à l'aide de conteneurs ouverts et

extensibles. Contrairement aux EJB, l'ensemble des propriétés fonctionnelles offertes par un conteneur JavaPOD est extensible. L'adaptation est réalisée par le conteneur : il intercepte les invocations et s'occupe de gérer la partie non-fonctionnelle de l'invocation en déléguant les invocations aux composants non-fonctionnels (appelés extensions).

La Container Virtual Machine (CVM) [HMT⁺04] est un outil conçu pour intégrer et adapter dynamiquement les services systèmes (non prévus initialement) dans les intergiciels monolithiques existants. La CVM ne propose pas un nouvel environnement dédié mais utilise les mécanismes de déploiement présents dans les intergiciels standards pour redéployer dynamiquement des services systèmes. Son architecture est généraliste par rapport à l'intergiciel ciblé : les mécanismes mettant en œuvre l'adaptation sont intégrés dynamiquement au moment de l'adaptation. Deux exemples d'adaptation ont été proposés sur la plateforme OpenCCM [ope04]. Le premier repose sur la technologie des intercepteurs portables et ajoute dynamiquement un service de monitoring flexible. Le second repose sur des composants orientés systèmes (COS) et ajoute dynamiquement un mécanisme non-fonctionnel de cryptage des communications.

Une architecture de conteneurs ouverts est proposée dans [VM01]. Cette architecture expose un certain nombre de propriétés du conteneur à l'aide de mécanismes d'interception, de coordination (pour ordonner les appels des fonctions systèmes) et de contrôle. Cette architecture permet d'adapter et d'étendre dynamiquement les fonctions systèmes.

Comet [PBY03] est un intergiciel basé sur des événements. Il peut être adapté en insérant des pré/post crochets aux composants (qui suivent la même sémantique que les intercepteurs) ou des filtres entre les composants. L'originalité de ce travail réside dans deux points :

- Un langage est associé à la reconfiguration dynamique. Les reconfigurations sont donc exprimées avec un haut niveau d'abstraction et peuvent être vérifiées par un compilateur en utilisant le typage et les accès aux éléments (privés/public...).
- Un composant est vu comme un ensemble de composants inter-connectés (comme fractal), ce qui permet d'adapter l'application avec une granularité assez fine.

Un récapitulatif des techniques utilisées pour rendre les intergiciels flexibles est présenté dans la section suivante.

2.2.3 Techniques pour la flexibilité dans les intergiciels

Trois approches se dégagent pour rendre les intergiciels plus flexibles :

- L'approche architecturale. La composition est réalisée de manière flexible en déployant et en reconfigurant dynamiquement l'architecture de l'application. Cette approche ne traite pas la flexibilité des composants individuellement.
- L'approche interception. Elle permet de rendre les composants flexibles individuellement, en interceptant les messages envoyés aux composants.
- L'approche système. Il s'agit de rendre l'intergiciel flexible en définissant des paramètres. Le POA est un outil mettant en œuvre cette approche, il permet, par exemple, de modifier l'ordonnancement des requêtes, la gestion multi-tâches etc... OpenORB, DynamicTAO et AspectIX suivent cette approche, les éléments internes de l'intergiciel peuvent être modifiés par l'application.

La figure 2.2 récapitule les différentes techniques utilisées pour rendre flexibles les plateformes étudiées.

Les approches systèmes et interceptions utilisent les notions de réflexion et d'aspects (voir section suivante) : intercepter des appels ou insérer des crochets revient à déléguer des

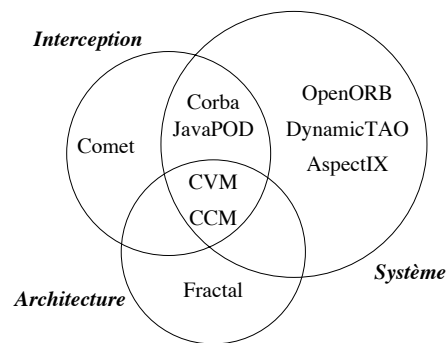


FIG. 2.2 – La flexibilité dans les intergiciels

appels à des méta-objets ou à tisser des aspects. Les intergiciels montrent bien les difficultés rencontrées lors de la conception de systèmes : plus l'hétérogénéité est masquée, plus les interfaces offertes par l'environnement d'exécution sont de haut niveau et plus les contraintes imposées aux applications sont fortes.

Les différents prototypes présentés sont adaptables mais ne permettent pas à une application de modifier les couches sous-jacentes de l'intergiciel comme le système ou la machine virtuelle intermédiaire. Cette limitation impose donc aux applications de correspondre à un intergiciel particulier et ne permet pas à une application d'étendre et de modifier un intergiciel en fonction de ses besoins.

Les intergiciels servent à masquer l'hétérogénéité des plate-formes participant à une application répartie, or les intergiciels eux-mêmes sont incompatibles entre eux. L'hétérogénéité des plate-formes empêche la réutilisation des objets développés pour un intergiciel donné dans un autre.

3 Outils dédiés à l'adaptation dynamique

Deux outils permettant d'adapter dynamiquement des environnements sont décrits dans cette section : la réflexion et le tissage d'aspects. La réflexion est un mécanisme permettant de connaître et de modifier l'état d'une application. Le tissage d'aspects est un paradigme de programmation. Il permet principalement de définir la liaison entre une méthode appelante et une méthode appelée avec un haut niveau d'abstraction. Cette section décrit ces deux outils et présente des environnements d'exécution Java offrant de la réflexion dynamique et du tissage d'aspects dynamique.

3.1 La réflexion

Les premiers travaux sur la réflexion ont été menés par B.C. Smith en 82 dans le cadre de sa thèse : *Procedural Reflexion in Programming languages* [Smi82]. B.C. Smith a défini la réflexion comme *“la capacité d'une entité à s'auto-représenter et plus généralement à se manipuler elle-même, de la même manière qu'elle représente et manipule son principal sujet”*² [Smi90]. La plate-forme développée par B.C. Smith est une machine virtuelle Lisp

²An entity's integral ability to represent, operate on, and otherwise deal with itself in the same way it represents, operates or and deals with its primary subject matter.

écrite et interprétée en Lisp : 3-Lisp.

Les systèmes réflexifs ont suscité de l'intérêt dans les domaines de l'intelligence artificielle, de la programmation logique, de la programmation fonctionnelle et de la programmation objet [DM95]. Il a fallu attendre 1987 pour qu'un consensus se dégage sur la réflexivité avec la thèse de Pattie Maes [Mae87a, Mae87b] qui a introduit la réflexivité dans les langages et systèmes à objets. Un système réflexif permet au développeur d'une application de séparer ce que doit faire l'application (la partie fonctionnelle de l'application) de sa mise en œuvre (la partie non-fonctionnelle).

3.1.1 Généralités

Terminologie. La réflexion peut être de deux natures différentes :

- *L'introspection* : c'est la possibilité, pour un système, de connaître son état. L'introspection permet à un système de répondre à des questions sur son état.
- *L'intercession* : c'est la possibilité, pour un système réflexif, de manipuler et modifier son état. L'introspection est nécessaire à l'intercession.

Pour permettre l'introspection et l'intercession, il faut être capable de donner une représentation de l'état des éléments constituant le système. Cette représentation, lorsqu'elle est différente de la représentation interne du système, est ce qu'on appelle une *réification* de l'état.

La réflexion peut manipuler des éléments de natures différentes. On parle de :

- *réflexion structurelle* lorsque les données internes du programme sont réfléchies ;
- *réflexion comportementale* lorsque les comportements (les algorithmes) du programme sont réfléchis, c'est à dire lorsque le code du programme peut être connu ou modifié par l'application.

La réflexion structurelle permet à une application de connaître et modifier les données de l'application, alors que la réflexion comportementale permet à une application de connaître et modifier le code de l'application. Ces deux types de réflexion sont complémentaires et indépendantes l'une de l'autre.

Réflexion et tour méta. D'après B.C. Smith [Smi84], un système réflexif peut être vu comme un système interprété par lui-même : il est capable de modifier sa propre interprétation en modifiant ses éléments internes. Le système peut donc être vu lui-même comme un programme de ce système. L'interprète est dit le *méta-niveau* de l'application (qui est appelée le niveau de base). La réflexion permet aussi de modifier et de manipuler le méta-niveau de l'application, ce qui introduit un nouveau méta-niveau dans l'application. Cette succession de méta-niveaux est ce qu'on appelle la tour méta. Cette tour méta est très bien illustrée dans MetaXa (voir section 3.1.5). Dans la pratique, seule la relation qui lie le niveau de base au niveau méta est étudiée : par récurrence, tous les niveaux sont étudiés.

Le niveau de base est aussi appelé partie fonctionnelle du système : c'est l'action que doit effectuer une application. Le niveau méta s'occupe donc d'interpréter et de contrôler le niveau de base.

Réflexion Objet.

Méta-Objet. Un méta-objet est le niveau méta d'un objet : il contrôle ou modifie un objet dans une plate-forme orientée objet. La relation qui lie un objet et un méta-objet est appelée "lien méta". De manière générale, un méta-objet est lui-même un objet du système et peut posséder un méta-objet.

MOP. Un MOP (Meta-Object Protocol) est un ensemble de méthodes et de règles permettant de manipuler le niveau de base d'une application. A l'aide de ces méthodes, le développeur modifie le comportement du niveau de base. Le protocole de communication entre le niveau de base et le niveau méta forme le MOP du système.

3.1.2 La réflexion dans Java

Dans la suite de cette étude sur la réflexion, nous nous intéressons principalement au monde Java : le prototype développé pendant ce doctorat est en effet une machine virtuelle Java. On peut distinguer trois familles d'approche pour introduire de la réflexion dans Java :

- les approches langages qui introduisent la réflexion au niveau du langage (avec ou sans pré-processeur) ;
- les approches bytecodes qui travaillent au niveau du binaire Java (les fichiers générés par la compilation de code source Java) ;
- les approches machines virtuelles qui visent à construire des machines virtuelles Java dédiées à la réflexion.

La spécification Java [GJSB00] propose par défaut une interface de programmation (API) pour la réflexion ([CWHT], leçon 58-61, pp. 681). Cette API est constituée de méthodes d'introspection. L'API permet d'accéder à la classe d'un objet, de connaître les méthodes et les champs d'une classe, de consulter/modifier les valeurs de ces champs, d'appeler des méthodes et de créer des instances à partir de ces classes. Cette API est principalement utilisée pour appeler des méthodes ou charger des classes qui ne sont pas connues lors du développement de l'application et qui peuvent être spécifiées à l'aide de fichiers de configuration. L'API Java met aussi à la disposition du développeur des chargeurs de classes qui peuvent modifier la manière dont les classes sont chargées et ainsi modifier le binaire Java au chargement. Les chargeurs de classes introduisent une réflexion comportementale : le code des méthodes Java peut être connu et manipulé. En revanche, l'API de réflexion Java ne permet pas de modifier dynamiquement les méthodes et/ou les champs associés à une classe une fois la classe chargée.

Un mécanisme d'interception dynamique est mis à la disposition du développeur depuis la version 1.3.1 de la spécification du langage Java. Ce mécanisme repose sur deux classes de construction d'interception de l'API de réflexion Java ³. Elles permettent d'encapsuler les appels de méthode d'un objet particulier pour interposer du code avant ou après l'appel. La figure 2.3 [Har01, Blo00] présente un exemple complet d'utilisation de ce mécanisme : toutes les invocations sur l'objet `bar` sont redirigées vers la méthode `invoke` de la classe `TraceProxy`. Celle-ci s'occupe d'afficher quelle méthode est appelée et de l'appeler effectivement.

Sur cet exemple, on constate que plusieurs indirections sont nécessaires pour appeler les méthodes `hello` et `goodbye` : l'appel doit être intercepté par la JVM, ensuite il est délégué

³Les classes `Proxy` et `InvocationHandler`.

<pre>package pub.foo; import pub.bar.*; import pub.proxy.*; public class Foo { public static void main(String[] args) { Bar bar = (Bar) TraceProxy.newInstance(new BarImpl()); bar.hello(2001, "xxx"); bar.goodbye("yyy", 2002); } }</pre>	<pre>package pub.proxy; import java.lang.reflect.*; import pub.bar.*; public class TraceProxy implements java.lang.reflect.InvocationHandler { private Object obj; public static Object newInstance(Object obj) { return java.lang.reflect.Proxy.newProxyInstance(obj.getClass().getClassLoader(), obj.getClass().getInterfaces(), new TraceProxy(obj)); } private TraceProxy(Object obj) { this.obj = obj; } public Object invoke(Object proxy, Method m, Object[] args) throws Throwable { Object result; try { System.out.print("begin method " + m.getName() + "("); for(int i=0; i<args.length; i++) { if(i>0) System.out.print(","); System.out.print(" " + args[i].toString()); } System.out.println(")"); result = m.invoke(obj, args); } catch (InvocationTargetException e) { throw e.getTargetException(); } catch (Exception e) { throw new RuntimeException ("unexpected invocation exception: " + e.getMessage()); } finally { System.out.println("end method " + m.getName()); } return result; } }</pre>
<pre>package pub.bar; import java.io.*; public interface Bar { public void hello(int i, String s); public void goodbye(String s, int i); }</pre>	
<pre>package pub.bar; import java.io.*; public class BarImpl implements Bar { public void hello(int i, String s) { System.out.println (" in pub.bar.Bar.hello"); } public void goodbye(String str, int i) { System.out.println (" in pub.bar.Bar.goodbye"); } }</pre>	
<pre><i>begin method hello(2001, xxx)</i> <i>in pub.bar.Bar.hello</i> <i>end method hello</i> <i>begin method goodbye(yyy, 2002)</i> <i>in pub.bar.Bar.goodbye</i> <i>end method goodbye</i></pre>	

FIG. 2.3 – Utilisation de Proxy en Java

à `invoke`, et enfin la méthode cible est appelée. Cette solution ralentie l'exécution et rend la compréhension du code difficile. En utilisant les mécanismes proposés par les spécifications Java, soit l'appel est dynamique et les performances sont dégradées, soit les performances sont bonnes, mais dans ce cas la méthode appelée ne peut pas être choisie dynamiquement pendant l'exécution de l'application.

3.1.3 Le niveau langage

La réflexion au niveau langage permet d'injecter des mécanismes de réflexion à partir des programmes sources Java, soit en modifiant le fichier source (OpenJava, Proactive), soit en décrivant les points de réflexion dans des fichiers de description externes (Reflective Java).

Reflective Java [WS97] utilise un langage permettant d'exprimer les appels de méthodes que l'on souhaite intercepter et détourner vers un méta-objet. Les fichiers sources n'ont pas besoin d'être modifiés pour utiliser Reflective Java : les scripts de configuration, avec les sources Java, génèrent de nouveaux fichiers sources Java. Les méthodes sont détournées vers des méta-objets qui prennent en charge l'exécution de la méthode. Deux possibilités sont offertes par Reflective Java :

- les liens métas peuvent être aplatis, dans ce cas la vitesse d'exécution est maximale mais plus aucune adaptation dynamique n'est possible ;
- les liens métas peuvent passer par un appel indirect (via l'API de réflexion Java), dans ce cas la vitesse diminue par rapport à l'approche statique, mais les liens métas peuvent être redéfinis pendant l'exécution.

Toutefois, la dynamicité de la seconde approche reste limitée : seules les méthodes qui auront été déclarées avant la compilation peuvent être interceptées et seuls les méta-objets déclarés avant la compilation peuvent être liés à l'objet de base.

OpenJava [Tat99, TCIK00] est une extension du langage Java basée sur l'utilisation d'un pré-processeur pour introduire de la réflexion. De nouveaux mots clés sont ajoutés au langage Java et indiquent au pré-processeur OpenJava quelles classes doivent être liées à un méta-objet. OpenJava permet principalement de capturer les appels de méthodes pour effectuer des pré et post traitements. Après le passage du pré-processeur, le niveau de base et le niveau méta sont aplatis : un nouveau fichier Java standard est généré, il contient les codes de pré et post-traitement directement dans les méthodes. Cette technique présente l'avantage d'éviter les indirections lors des appels aux méthodes capturées mais reste relativement statique puisque toutes les adaptations sont effectuées avant la compilation de l'application.

Proactive (The Java Library for Parallel, Distributed, Concurrent computing with Security and Mobility) [pro04] est une bibliothèque Java orientée calcul parallèle. Proactive est le pendant de MPI [mpi] (Message Passing Interface) pour Java [BBC02]. Proactive repose sur Java RMI pour la communication et sur des mécanismes de réflexion pour capturer les appels de méthodes et les créations d'objets. Proactive est une bibliothèque : il est laissé à la charge du programmeur de l'application d'inclure le code nécessaire à l'utilisation de Proactive dans son programme source. Pour créer une instance d'une classe possédant un méta-objet, il faut déclarer un encapsulateur (Stub) et le méta-objet (Proxy) qui s'occupe de créer l'instance de l'objet. Proactive n'apporte donc pas de nouvelles possibilités à l'API de réflexion Java et est soumise aux mêmes limites.

Ces différentes techniques visant à introduire de la réflexion dans Java au niveau du code source sont soit dynamiques (passage par une indirection dans Reflective Java et Proactive), soit performantes (pas d'indirection d'appel dans Reflective Java ou OpenJava). Ces outils

restent peu dynamiques (les binaires générés sont statiques) et imposent au développeur d'avoir accès au code source de l'application.

3.1.4 La réécriture de bytecode

La réécriture de bytecode permet d'introduire la réflexion directement à partir du binaire Java. Le bytecode Java⁴ est modifié pour insérer les appels aux méta-objets ou pour capturer certains événements. Cette technique évite de modifier la machine virtuelle ou le code source Java. La réécriture de bytecode est largement exploitée dans les plate-formes à aspects (voir section 3.2). La modification de bytecode peut avoir lieu soit lors du chargement de l'application dans la machine virtuelle Java, soit directement en réécrivant le fichier binaire. Les possibilités offertes par la réécriture de bytecode sont les mêmes que celles du niveau langage : les deux approches sont soumises aux mêmes contraintes.

Javassist [Chi98, CN03] est une bibliothèque Java de modification du bytecode d'une classe. Cette modification peut être effectuée soit lors de la compilation du fichier source, soit lors du chargement de la classe dans la machine virtuelle. La bibliothèque BCEL [DvZH] (The ByteCode Engineering Library) est aussi une bibliothèque de manipulation de bytecode Java, facilement utilisable pour tisser des aspects (voir section 3.2) ou introduire de la réflexion. BCEL modifie (ou génère) du bytecode lors du chargement d'une nouvelle classe dans la machine virtuelle Java. Javassist et BCEL peuvent être considérées comme des couches réflexives car modifier le bytecode d'une classe revient à modifier son interprétation. Cette réflexion est donc comportementale. En revanche, aucun MOP n'est défini dans ces bibliothèques : ce sont des outils de construction de MOP ou de tisseur d'aspects.

Dalang [WS99] permet d'introduire de la réflexion en encapsulant les classes Java. Dalang offre une interface de haut niveau pour capturer des appels à des méthodes et pour les déléguer à des méta-objets. Les manipulations de bytecode de Dalang reposent sur BCEL et sont réalisées soit lors du chargement de la classe dans la machine virtuelle Java soit en dehors de la JVM sur le binaire. Dalang utilise un langage de description de liaison (XML) pour définir quelles méthodes et quels champs doivent être interceptés. Dalang se situe à la limite du tisseur d'aspects (voir section 3.2). La figure 2.4 présente le fonctionnement de Dalang [IW99]. Dalang commence par renommer la classe d'origine `Foo.class` en `Foo_orig.class`. Ensuite, une nouvelle classe `Foo.class` est générée automatiquement par Dalang. Cette classe encapsule l'ancien comportement et hérite de la méta-classe `MetaFoo.class` qui implante les comportements réflexifs de la classe.

Kava [IW99, WS01] a été développé par la même équipe que Dalang. C'est une bibliothèque permettant de faire de l'encapsulation de bytecode : un certain nombre de bytecodes d'une méthode peuvent être détournés vers un appel à un méta-objet. On peut, par exemple, n'encapsuler que l'affectation d'un champ d'un objet dans une méthode donnée pour en modifier le comportement. On parle ici de micro-encapsulateur.

La réécriture de bytecode permet d'introduire de la réflexion dans des applications sans avoir à modifier ni les sources des applications, ni la machine virtuelle Java. Toutefois, l'utilisation des bibliothèques type BCEL et Javassist entraîne des erreurs difficilement détectables : les exceptions générées par la machine virtuelle lorsqu'une classe est malformée ne sont pas explicites⁵.

⁴La représentation intermédiaire du code des méthodes Java pour les machines virtuelles Java.

⁵Une exception `ClassFormatErreur` est générée par la VM et n'indique pas d'où vient le problème.

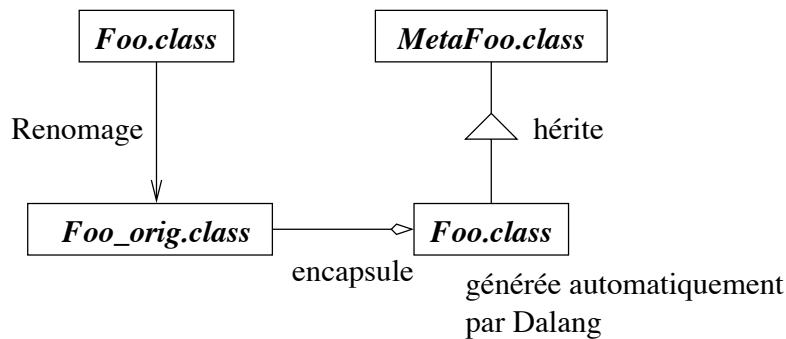


FIG. 2.4 – La réflexion dans Dalang

Les techniques à base de réécriture de bytecode au chargement introduisent du code pour gérer la réflexion et ajoutent des indirections dans le code : la vitesse d'exécution globale de l'application diminue. Les possibilités d'adaptation dynamique sont relativement faibles puisqu'une classe chargée ne peut plus être modifiée. La réécriture de bytecode et la réflexion au niveau langage sont très proches l'une de l'autre : les deux finissent par modifier le binaire de l'application et offrent les mêmes possibilités. Pour les deux approches, plus l'adaptabilité est dynamique (choix du méta-objet à l'exécution), moins les performances sont bonnes : une méthode d'encapsulation est appelée pour délivrer l'appel à la bonne méthode.

3.1.5 Les JVM réflexives

La troisième technique qui introduit de la réflexion dans les applications Java est l'utilisation de machines virtuelles modifiées : c'est la machine virtuelle elle-même qui prend en charge le MOP. Cette approche est la plus dynamique et la plus performante. La machine peut redéfinir des liens métas pendant l'exécution et évite l'utilisation d'indirections dues à l'API de réflexion Java.

Guaraná [OB99, OGB98c, OGB98b, OGB98a] est une machine virtuelle Java réflexive basée sur la machine virtuelle libre *Kaffe OpenVM*. Le MOP de Guaraná repose sur des méta-configurations décrivant les liens métas. Un objet possède au plus un méta-objet, appelé *méta-objet primaire* de l'objet de base. Ce méta-objet peut être un *composeur*, une association de méta-objets.

Lorsqu'une opération doit être effectuée sur un objet de base possédant un méta-objet primaire, l'opération est déléguée à ce méta-objet. Ce dernier renvoie l'un des résultats suivants :

- un résultat qui sera renvoyé à la place de l'opération de base ;
- une opération de remplacement effectuée à la place de l'opération de base ;
- le résultat de l'opération de l'objet de base.

Les composeurs sont des méta-objets particuliers permettant d'organiser une association de plusieurs méta-objets. Un composeur simple a été implémenté : le composeur séquentiel. Ce composeur possède un ensemble de sous méta-objets et appelle successivement l'opération sur ces sous méta-objets. La figure 2.5 présente un exemple de fonctionnement de Dalang. La composition est réalisée avec un composeur séquentiel. L'opération `op()` est appelée sur l'objet de base. Le noyau de Guaraná délègue alors cette opération au méta-objet primaire

(un composeur séquentiel) **Comp**. **Comp** transmet l'opération à **MetaA** qui remplace l'opération **op()** par une opération **op2()** et demande à examiner le résultat à la fin de la séquence. **Comp** transmet ensuite l'opération **op2()** à **MetaB** qui appelle l'opération **op2()** sur l'objet de base et renvoie le résultat **r**. Ce résultat **r** est ensuite envoyé à **MetaA** qui le remplace par un résultat **r'**. C'est ce résultat qui sera renvoyé à l'application.

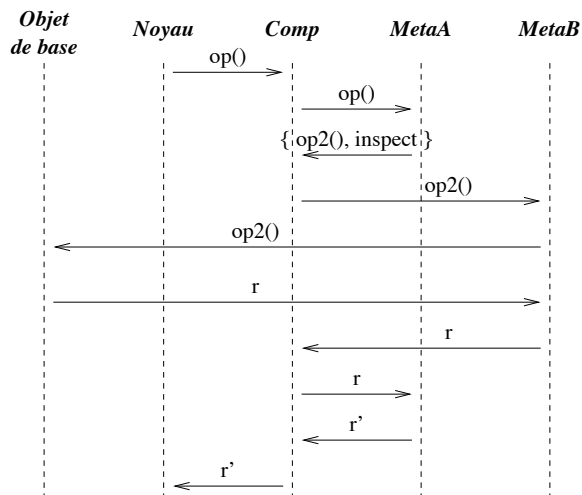


FIG. 2.5 – Compositeur séquentiel dans Guaraná

Différents types de compositeurs peuvent être utilisés, en particulier un compositeur concurrent permettant d'exécuter les différents appels aux méta-objets de façon parallèle. Guaraná possède en outre des méta-configurations permettant de décrire dynamiquement la configuration des liens métas et un système de protection permettant d'assurer la sûreté de la définition des liens métas.

MetaXa⁶ [Gol97, Gol98] est une machine virtuelle Java réflexive basée sur des événements. A chaque objet de base est associé une pile de méta-objets indexés de 1 à n . Lorsqu'une opération $op()$ doit être exécutée, un événement **doExecute(op)** est généré et envoyé au méta-objet de niveau n . Ce méta-objet effectue alors le traitement à son niveau et délègue éventuellement le traitement au niveau du dessous, $n - 1$. Le niveau 1 appelle éventuellement l'opération sur le niveau de base.

Les événements sont générés dans MetaXa lors d'un appel de méthode, d'un accès à un champ ou de la création d'un objet. MetaXa propose une composition de méta-objets proche du compositeur séquentiel de Guaraná mais ne permet pas de changer cette configuration. La liaison entre un objet et une pile de méta-objets est effectuée par le MetaJava System (la machine virtuelle Java MetaXa) qui intercepte les appels.

3.1.6 Récapitulatif

Trois approches introduisent de la réflexion dans les programmes Java : l'approche langage, l'approche à base de réécriture de bytecodes et les machines virtuelles dédiées. Ces

⁶ Anciennement MetaJava.

trois approches sont aussi celles utilisées pour ajouter du tissage d’aspects (voir section 3.2). La section 3.3 discute plus précisément ces différentes méthodes.

3.2 Le tissage d’aspects

La notion d’aspects a été introduite par G. Kiczales [KLM⁺97] en 1997. Un aspect regroupe des fonctionnalités transversales de l’application ([PRS04], chapitres 1 et 2). L’introduction de la programmation orientée aspect (AOP) repose sur deux constats :

- Une méthode est appelée en plusieurs points. Changer la signature ou la sémantique d’une méthode oblige donc le développeur à modifier tous les sites d’appel de cette méthode (*dispersion* du code).
- Certaines caractéristiques d’un programme sont *transversales* au découpage objet (ou fonctionnel). Par exemple, la sécurité dans un protocole client-serveur, est gérée à la fois dans les classes clientes et serveurs.

La notion d’aspects, comme la notion d’objet, n’est pas dépendante d’un environnement particulier. Dans cette section, nous nous intéressons principalement au tissage d’aspects en Java : les plate-formes les plus abouties ont été développées dans ce langage.

3.2.1 Définition

Un aspect est donc une fonctionnalité transversale d’une application. Il est constitué d’un ensemble de point de jonction (“pointcut”) formant une coupe de l’application. L’aspect va s’occuper d’implanter cette coupe. Techniquement, un point de jonction intercepte un appel de méthode, un accès à un champ ou encore la levée d’une exception. Le code de l’aspect, aussi appelé “code advice”, est appelé sur un point de jonction.

Associer une coupe au “code advice” est ce qu’on appelle tisser un aspect. Trois solutions sont utilisées pour tisser des aspects, en intervenant :

- Au niveau langage. L’aspect est tissé lors de la compilation de l’application en étendant le langage. Cette technique produit un fichier exécutable avec l’aspect pré-tissé.
- Au niveau du chargement. L’aspect est tissé lors du chargement de l’application en réécrivant le bytecode de l’application.
- Directement dans la machine virtuelle. La machine virtuelle Java prend en charge le tissage d’aspects dynamiquement pendant l’exécution de l’application.

Ces trois solutions sont à mettre en parallèle avec les techniques permettant de rendre une application réflexive (voir section 3.1). Le but premier de l’AOP est d’améliorer la réutilisabilité du code. Toutefois, le tissage d’aspects s’inscrit dans le cadre de l’adaptabilité dynamique : un aspect peut être tissé (ou re-tissé) pendant l’exécution.

3.2.2 Technique langage

La plate-forme AspectJ [GL03], développée par G. Kiczales, est la première illustration de la notion d’aspects. Elle repose sur des fichiers de description d’aspects : les fichiers sources Java restent compatibles avec des compilateurs traditionnels. Les mots clés des fichiers d’aspects décrivent les points de jonction ainsi que les associations entre les coupes et les “codes advices”. Les premières versions d’AspectJ généraient des fichiers sources. Actuellement des fichiers binaires Java sont générés pour des questions de performances : il est plus facile de

manipuler le bytecode que le source et le parseur/lexeur des premières versions faisait double emploi avec le parseur/lexeur du compilateur Java.

AspectJ évite au développeur de modifier les fichiers source de l'application (le code sans aspects). La description d'un aspect se fait dans des fichiers d'aspects. Le binaire généré par AspectJ reste relativement statique : il n'y a aucun moyen de définir de nouvelles coupes après la phase de compilation.

A chaque point de jonction, le “code advice” encapsulant l'appel original est appelé. Cette méthode effectue un travail avant (mot clé **before**), après (mot clé **after**) ou autour (mot clé **around**) de l'appel original. Le “code advice” est libre d'appeler ou non la méthode originale lors d'un **around**. La notion d'interception dans les intergiciels (voir section 2.2) est donc un cas particulier de tissage d'aspects.

3.2.3 Réécriture de bytecode au chargement

La réécriture de bytecode est utilisée pour injecter les interceptions directement dans le binaire. Jac [Objb, jaca] (Java Aspect Component) utilise cette solution. Les classes qui possèdent des points de jonction sont réécrites par le chargeur de classes de Jac. Chaque point de jonction est redirigé vers un “wrapper” (sémantiquement équivalent à un intercepteur). Le “wrapper” s'occupe d'encapsuler l'appel à la méthode originale et d'effectuer un post ou un pré-traitement.

Jac permet de re-tisser un aspect dynamiquement : la liaison entre les “wrappers” et les “codes advices” peuvent être redéfinis à la volée. Cependant, le nombre de points de jonction n'est pas extensible dynamiquement : une fois la classe chargée dans la machine virtuelle, il n'existe aucun moyen pour ajouter un nouveau point de jonction.

Les performances en terme de vitesse d'un point de jonction sont assez faibles. À chaque point de jonction, le “wrapper” est appelé avec la méthode interceptée en argument. Deux indirections aux moins sont alors nécessaires pour appeler la méthode cible :

- appel du “wrapper” ;
- appel d'une fonction **proceed** qui appelle la méthode interceptée.

Comme le nombre de points de jonction n'est pas extensible dynamiquement, un “wrapper” est défini à chaque point de jonction appartenant à une coupe éventuelle, même si celle-ci n'est pas utilisée.

JBoss AOP [jbo] est assez proche de Jac, tant en terme de fonctionnalité que de fonctionnement. Les aspects sont décrits dans des fichiers XML et tissés à l'exécution (après définition des points de jonction au chargement). AspectWerkz [Bon04] suit aussi le même principe. Les aspects sont définis dans des fichiers XML ou dans les commentaires du fichier Java (type JavaDoc).

3.2.4 Technique Machine Virtuelle Modifiée

Steamloom [BHMO04] est une machine virtuelle Java basée sur la JikesRVM d'IBM[jik]. Steamloom permet de tisser les aspects dynamiquement pendant l'exécution d'une application : les aspects sont décrits sous la forme de programmes Java associant des points de jonction avec des “codes advices”. La machine virtuelle prend en charge le tissage : elle indique au compilateur Just In Time (JIT) de la JikesRVM que les méthodes sur lesquelles sont tissées des aspects doivent être recompilées.

L'approche adoptée par Steamloom permet à la fois de se dégager des problèmes liés à la vitesse d'exécution des points de jonction en insérant directement dans le code les interceptions, mais aussi des problèmes liés à l'extensibilité dynamique en ajoutant, après le chargement d'une classe, les points de jonction dynamiquement.

3.3 Les outils d'adaptation : discussion

Le tissage d'aspects dynamique et la réflexion offrent des mécanismes de haut niveau pour adapter des applications pendant qu'elles s'exécutent. Ces mécanismes reviennent à intercepter des événements (affectation d'un champ, appel d'une méthode, levée d'une exception). Pour introduire du tissage d'aspects ou de la réflexion dans des programmes Java, trois approches sont utilisées : l'approche langage, la réécriture de bytecode et les machines virtuelles dédiées. Le tableau 2.6 récapitule les différentes plate-formes présentées et met en avant trois points importants dans ces environnements : adaptabilité dynamique, portabilité et performance.

Les approches langage et réécriture de bytecode sont équivalentes en terme de possibilités. En effet, tout ce qui peut être réalisé pendant la compilation peut l'être au chargement en modifiant le bytecode. L'avantage de la réécriture de bytecode par rapport à l'approche langage est l'instant de l'adaptation : en injectant la réflexion ou les aspects pendant la compilation, c'est le concepteur de l'application qui choisit quels aspects et quels liens métas seront mis en œuvre. En injectant la réflexion ou les aspects directement dans le bytecode (au chargement ou dans le binaire), c'est l'utilisateur (ou l'administrateur) qui choisit. La réécriture de bytecode offre donc une utilisation plus souple sans désavantage par rapport à l'approche langage.

Que ce soit avec l'approche langage ou avec la réécriture de bytecode, les performances sont en contradiction avec la dynamique. Pour avoir de bonnes performances, il faut aplatir les indirections (vers les "codes advices" ou vers les méta-objets). En évitant les indirections, les performances augmentent mais plus aucun choix n'est possible pendant l'exécution. Pour modifier dynamiquement quel "code advice" ou quel méta-objet sera appelé pendant l'exécution, il faut laisser une indirection supplémentaire. Elle ralentit alors l'exécution globale de l'application. De plus, même si aucun lien méta ou aucun "code advice" n'est présent en un point à un instant donné, l'indirection doit être présente pour rendre adaptable l'application. Pour rendre une application entièrement adaptable dynamiquement, il faudrait placer ces indirections sur tous les appels de méthodes, tous les accès aux champs, toutes les levées d'exception, voir tous les bytecodes... Ce qui n'est pas réaliste pour des raisons de performances.

Les machines virtuelles Java modifiées résolvent ce problème : les liens vers les méta-objets ou vers les "codes advices" sont aplatés, tout en offrant un haut degré d'adaptation dynamique. En effet, la machine virtuelle peut utiliser ses structures internes pour poser les points d'adaptation dynamiquement. Steamloom illustre bien les bénéfices de l'approche : les points de jonction sont insérés dynamiquement en recompilant les méthodes présentes dans une coupe. Cette nouvelle compilation permet d'aplatir l'accès au "code advice".

Toutefois, les machines virtuelles modifiées sont incompatibles entre elles et diminuent la portabilité des applications. Cette constatation pousse donc les concepteurs d'application à respecter la norme et à se tourner vers des solutions portables. Ces machines virtuelles Java modifiées restent donc malheureusement peu utilisées et le travail effectué pour obtenir de bonnes performances et un haut degré d'adaptabilité n'est que peu exploité. Ces remarques

	Langage		Réécriture		JVM modifiée		Ad Dyn	PB	Perf
	Refl	Asp	Refl	Asp	Refl	Asp			
Reflective Java	X						+	+	-
							-	+	+
OpenJava	X						-	+	+
Proactive	X						-	+	+
AspectJ		X					-	+	+
BCel			X				+	+	-
							-	+	+
Javassist			X				+	+	-
							-	+	+
Dalang			X				-	+	+
Kava			X				-	+	+
Jac				X			+	+	-
JBossAOP				X			+	+	-
AspectWerkz				X			+	+	-
Guaraná					X		++	-	+
MetaXa					X		++	-	+
Steamloom						X	++	-	+

Ad Dyn	Adaptabilité Dynamique	
	-	Aucune
	+	Adaptation dynamique des liens métas ou des points de jonction existants
	++	Ajout dynamique de liens métas ou de points de jonction
PB	Portabilité Binaire	
	-	Non
	+	Oui
Perf	Performance	
	-	Diminuée
	+	Normale

FIG. 2.6 – Plate-formes dédiées à l'adaptation

sont plus générales : plus un environnement est adaptable dynamiquement, plus il perd soit en performance, soit en portabilité. Le but de cette thèse est de réconcilier ces trois points.

4 Domaines connexes à l'adaptabilité : réseau et langage

4.1 Les réseaux actifs

Les réseaux actifs sont un cadre général pour le déploiement dynamique de protocoles de communication et de services dans les environnements distribués. Les réseaux actifs sont apparus avec les besoins de traitements spécifiques de certains paquets dans un routeur. Dans [TW96], Tannenhouse et Wetherall présentent les premières technologies pour réseaux actifs et montrent les bénéfices de cette approche :

- L'échange de code est la base pour des protocoles adaptables et permet une plus grande interaction que l'échange de données brutes.
- Un paquet actif (un paquet et un code associé) permet à une application de déployer un protocole dédié à des besoins spécifiques. Les réseaux actifs donnent un cadre générique pour le déploiement de nouveaux protocoles sans interruption de service, par exemple, un "firewall", de caches Web ou de routeur multicast.
- Les réseaux actifs permettent de déployer plus vite de nouveaux protocoles (les premières versions du standard IPv6 datent de 94, le standard a été adopté par l'IETF en 98 [DH98] et reste peu utilisé en 2004...).

Les réseaux actifs forment un cadre hautement adaptable puisque du code est exécuté dans les routeurs par les paquets actifs. Les problématiques soulevées par cette approche sont nombreuses. Les principales sont la sécurité, la compilation à la volée et les performances. La notion de réseaux actifs a beaucoup contribué à l'élaboration de cette thèse : les applications actives présentées dans nos travaux sont directement inspirées de la notion de paquets actifs dans les réseaux actifs.

Un premier travail de recherche, précurseur des technologies actives, est donné dans "active message" [vECGS92] qui s'intéresse à la communication dans une machine multi-processeurs. Dans Active Message, chaque message contient une unique instruction qui correspond à un gestionnaire utilisateur chez le destinataire. Cette technique permet au message de spécifier à quel gestionnaire il doit être délivré, ce qui accélère la phase de distribution (dispatch).

4.1.1 L'architecture DARPA des réseaux actifs

L'architecture des réseaux actifs est clairement définie par le DARPA [Cal99, GFS98] (voir figure 2.7). Un réseau actif est constitué de noeuds actifs inter-connectés. Chaque noeud actif fait tourner un système d'exploitation pour noeud (NodeOS) et un ou plusieurs Environnement d'Exécution (EE). Les NodeOS sont responsables de l'allocation et de la gestion des ressources du noeud. Chaque environnement d'exécution implante une machine virtuelle qui interprète (ou exécute) les paquets actifs. Une application pour réseaux actifs est une application exécutée sur un ensemble d'environnement d'exécution : c'est l'application pour réseaux actifs qui fournit un service particulier pour l'utilisateur final.

Bien que dans la terminologie donnée par la DARPA, les applications s'exécutant sur plusieurs noeuds soient appelées applications actives, nous n'utilisons pas ce terme pour

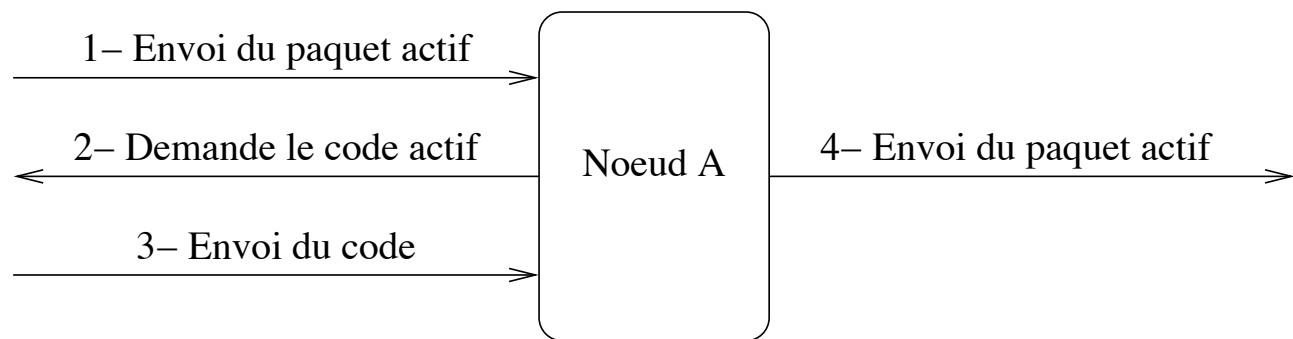


FIG. 2.8 – Déploiement dans ANTS

qui empêche l'application pour réseaux actifs d'accéder aux fonctions fournies par le NodeOS. Cette contrainte entraîne des performances extrêmement basses : de l'ordre de 160 paquets traités par seconde sur un Pentium Pro à 200MHz [WGT98]. Le projet [THL02] vise à construire un environnement ANTS pour machine nue à partir d'une machine virtuelle Java dédiée (JanosVM) et d'un environnement ANTS (ANTSR) modifié. Cette approche lève les limites imposées par l'utilisation d'une machine virtuelle Java standard.

Packet Language for Active Networks (Plan) [HKM⁺98, AHK⁺98] est un langage dédié au réseaux actifs. Plan suit une approche intégrée : chaque paquet contient le code que le paquet doit exécuter (sous forme de source). Si un code particulier est utilisé fréquemment sur un noeud, une grande bande passante est alors gaspillée. Une étude comparative de ANTS et Plan est présentée dans [AFD⁺00]. Cette évaluation montre que la taille d'un paquet encapsulant un protocole de ping est de 11134 octets dans PLAN contre 91 pour ANTS. La taille et le temps de traitement des paquets dans PLAN entraîne des performances plus faibles que ANTS, de l'ordre de 50 secondes pour traiter 15 ping (aller-retour) contre quelques centaines de milli-secondes dans ANTS.

4.1.3 D'autres approches de réseaux actifs

YAAP [RBH00] (Yet Another Active Protocol) implante les fonctionnalités communes aux réseaux actifs sans compromettre la sécurité ou les performances. Cette plate-forme de recherche est une synthèse des autres travaux dans le domaine. Elle repose sur une machine virtuelle Java qui interopère avec le noyau (Linux) via des fonctions de la Java Native Interface (JNI) [Lia99] qui sont appelées directement à partir du noyau (upcalls). Les applications pour réseaux actifs se présentent sous la forme de programme Java, ce qui assure la sécurité.

CANES [SKB⁺01] est un environnement d'exécution basé sur Bowman [MBZC00], un NodeOS générique, qui propose d'étendre l'environnement d'exécution à l'aide de fonctions spécialisables en des points clairement définis (slot). Cette approche reste relativement peu dynamique (le nombre de points d'adaptation est fixe) mais donne des performances proches des réseaux non-actifs. Le protocole multicast implanté au dessus de CANES montre que l'environnement d'exécution doit proposer un certain nombre de primitives optimisées, en particulier une gestion fine des temporisateurs.

ASP-EE [BLBF02] (Active Signalisation Protocol) est un environnement d'exécution

pour réseaux actifs écrit et programmable en Java. Le typage fort et l'isolation des paquets actifs assure la sécurité. La flexibilité est assurée par le chargement dynamique de classes qui spécialisent le comportement de l'application pour réseaux actifs. L'environnement d'exécution ASP offre des fonctionnalités de haut niveau aux applications pour réseaux actifs comme la persistante, une gestion de tâches, des horloges ainsi qu'un système de communication inter applications pour réseaux actifs.

ANN [DPP⁺99] est un réseau actif implantant le protocole ANTS et se proposant de supporter un trafic au giga-octets par seconde. Ces travaux mettent en avant les problèmes de performances dans les réseaux actifs : sur un Pentium à 300MHz avec un débit de 10 Gb/s et une moyenne de 512 octets par paquet, un routeur doit pouvoir traiter chaque paquet en 114 cycles d'horloge. Partant de ce constat, l'article se penche sur la couche matérielle des NodeOS et montre qu'avec une répartition de charge entre différents processeurs et l'utilisation de FPGAs (Field Programmable Gate Arrays), de telles performances devraient être atteintes. Cependant, ANN n'est qu'une proposition et aucun prototype n'a été implanté pour vérifier les arguments avancés.

P4 [HSM98] est un exemple d'adaptation d'un routeur utilisant aussi des FPGAs. P4 s'intéresse au contrôle de congestion dans TCP. Pour TCP, des paquets erronés proviennent forcément d'une congestion or, des paquets erronés peuvent aussi provenir d'un bruit sur le lien. L'article propose donc d'implanter dans des FPGAs un contrôle d'erreurs (Forward Control Error) permettant de retrouver la paquet envoyée avec de la redondance d'informations. Cette redondance d'informations diminue les performances du routeur lorsque le bruit est faible mais devient meilleure que le contrôle de congestion de TCP lorsque le bruit augmente. P4 ne rentre pas directement dans le cadre des réseaux actifs (aucune plate-forme de déploiement de code n'est proposée), mais, en conclusion de l'article, les auteurs proposent une généralisation de l'approche pour que les différents noeuds puissent changer le protocole de correction d'erreurs en fonction du bruit sur le réseau, ce qui revient à construire un réseau actif reprogrammant le matériel sous-jacent des NodeOS.

4.1.4 Conclusion

Les réseaux actifs forment un paradigme novateur dans le domaine des réseaux, mais aussi dans le domaine des systèmes (les besoins en adaptation de l'environnement sont importants) et de la compilation (les performances requises dans un routeur nécessitent que le code des paquets actifs soit compilé à la volée sur chaque noeud). Les différents prototypes de réseaux actifs présentés proposent un haut degré d'adaptabilité du protocole mais ne permettent pas à une application pour réseaux actifs de reconfigurer en profondeur l'environnement (l'environnement d'exécution, le NodeOS ou même le matériel). Ces limitations imposent donc à l'application de correspondre à un environnement d'exécution spécifique. On retombe donc sur les difficultés présentées dans les systèmes en section 2.1 : les choix effectués par l'EE ne sont pas forcément adaptés à tous les besoins d'une application.

Avec la multiplication des protocoles de réseaux actifs se pose aussi des problèmes d'hétérogénéité et de compatibilité entre réseaux actifs. Il n'existe pas encore de passerelle pour faire communiquer deux types de réseaux ensemble.

4.2 Les Domain Specific Languages

Les Domain Specific Languages (DSL) sont des langages dédiés à des domaines particuliers. Le but des DSL est principalement d'offrir une interface de programmation de haut niveau qui capture le savoir faire des spécialistes d'un domaine particulier. L'utilisation de DSL permet [vDKV00] :

- d'éviter les erreurs de développement dues à l'utilisation d'un langage généraliste : le DSL étant dédié au domaine particulier, il offre des macro-instructions qui masquent la complexité technique du domaine ;
- de réutiliser facilement du code : si l'API système ou le matériel change, il suffit de modifier le compilateur (ou l'interpréteur) du DSL pour porter tous les programmes écrits dans ce DSL ;
- de vérifier le code avant de le compiler : le langage étant dédié, le compilateur peut faire des assumptions sur la cohérence du programme ;
- de rendre le code plus clair et donc plus facile à valider par un expert du domaine.

Toutefois, capturer la spécificité d'un domaine et le retranscrire dans un langage dédié est un travail long et difficile. Les DSL sont des langages concis, mais ils n'empêchent pas le développeur d'avoir à se familiariser avec une nouvelle syntaxe et de nouveaux concepts. Les DSL sont compilés ou interprétés : il arrive assez fréquemment que le programme généré par un DSL soit moins optimisé qu'un code produit à la main. Ce dernier problème n'en est pas vraiment un : de la même façon, un programme écrit en assembleur est bien souvent plus performant qu'un programme écrit dans un langage de haut niveau. Les langages de haut niveau sont utilisés car ils offrent des outils de vérification et permettent d'écrire un code plus concis et plus clair. Cette remarque s'applique aussi aux DSL.

De nombreux DSL ont été développés, citons par exemple Bossa [LMB02] dédié à la construction d'ordonnanceurs, HiPEC [LCC97] dédié à la construction de gestionnaires de mémoire virtuelle au niveau applicatif, PLAN-P [TCM98] dédié à la construction d'application pour réseaux actifs vérifiable dans des réseaux actifs ou encore le langage GAL [TMC97] dédié à des pilotes de cartes graphiques,

Devil [RMC⁺00, MRC⁺00] est dédié à la construction de pilotes de périphériques. 85% des bugs dans windows XP proviennent des pilotes de périphériques [SBL03], ces erreurs sont principalement dues à la manipulation bits à bits d'entiers qui peuvent représenter plus de 30% du code d'un pilote. Devil illustre parfaitement les DSL : les accès bas niveaux (`in` et `out` sur les ports d'un i386) sont masqués par Devil avec des expressions de plus haut niveau. Ces expressions sont facilement lisibles et vérifiables. Devil a été utilisé pour construire un pilote de carte réseau ethernet, de souris, de carte son et de carte graphique. Les performances du code produit par Devil sont légèrement moins bonnes que celles du même pilote développé à la main (sous Linux), mais les programmes écrits en Devil assurent une plus grande réutilisabilité du code ainsi qu'une meilleure sûreté.

5 Les challenges de l'adaptabilité

Les environnements d'exécution offrant des mécanismes d'adaptation sont principalement utilisés pour améliorer l'utilisation des ressources matérielles. Cette optimisation des ressources est utilisée pour :

- augmenter la vitesse d'exécution d'une application ;

- exécuter des applications sur des machines n’ayant pas les ressources suffisantes pour exécuter l’application.

Lors de la conception de systèmes adaptables (ou flexibles), il ne faut pas perdre de vue le problème à résoudre : un environnement d’exécution adaptable ne doit pas dégrader les performances des applications. Deux raisons principales poussent les concepteurs d’environnements à créer des environnements flexibles :

- les choix effectués lors de la conception de l’environnement ne peuvent pas convenir à toutes les applications ;
- les choix effectués lors de la conception de l’application peuvent s’avérer inadéquate lorsque l’utilisation (le contexte) de l’application change.

La flexibilité dans les intergiciels pose les mêmes problèmes que dans les systèmes d’exploitation : en masquant les couches sous-jacentes à l’application, de nouvelles contraintes sont imposées. Les mécanismes flexibles offerts par les intergiciels ne sont pas encore suffisants et sont soumis aux mêmes problèmes que les systèmes réflexifs ou les systèmes à tissage d’aspects : soit les performances de l’intergiciel sont dégradées, soit les solutions proposées deviennent incompatibles entre elles.

L’étude de la réflexion et du tissage d’aspects montre qu’il est difficile de réconcilier trois propriétés : l’adaptabilité dynamique, la portabilité et les performances. En introduisant des outils d’adaptation dans des plate-formes standards, l’adaptabilité reste peu dynamique et il y a une contradiction entre l’adaptabilité dynamique et les performances. En introduisant des outils d’adaptation dans de nouveaux environnements (ou des environnements existants modifiés), l’adaptabilité dynamique et les performances sont maximales, mais les applications développées pour ces environnements deviennent inutilisables dans les environnements préexistants.

Les réseaux actifs sont une manière originale d’introduire de la flexibilité : les paquets transportent avec eux du code exécutable (sous forme de paquets actifs ou de capsules). Les paquets peuvent donc agir sur leur environnement pour en modifier le fonctionnement et l’adapter aux besoins d’un protocole particulier. Les plate-formes décrites dans ce chapitre restent toutefois peu adaptables et peu performantes. L’idée phare des réseaux actifs, à savoir le paquet modifie son environnement pour l’adapter à ses besoins, est à la base de nos travaux.

Les DSL ne s’intéressent pas à l’adaptabilité mais nous enseignent que des langages bien adaptés à des domaines particuliers peuvent résoudre de manière élégantes les contraintes du domaine. Les différents langages étudiés dédiés au tissage d’aspects ou à la réflexion sont des langages spécifiques. Chacun de ces langages impose un mécanisme d’adaptation particulier. Nous proposons dans cette thèse de construire simultanément le mécanisme d’adaptation adéquate pour une application particulière et le langage associé.

CHAPITRE 3

Architecture de Machines Virtuelles Virtuelles pour Applications Actives

Sommaire

1	Introduction	41
2	La Ynvm	45
3	La μvm	49
4	Récapitulatif	61

1 Introduction

Pour faire face à l'évolution des besoins logiciels, nous posons l'hypothèse suivante : *c'est à l'application de prendre en charge la construction de son environnement*. Une telle application devient *active*, elle transforme un environnement d'exécution pour qu'il offre les abstractions et les mécanismes logiciels nécessaires à son bon fonctionnement. Les applications actives réunissent les qualités des surcouches logicielles et des environnements dédiés :

- Elles construisent exactement l'environnement qui leur correspond ainsi que les mécanismes les mettant en œuvre.
- Elles construisent des environnements performants. Le cœur de l'environnement d'exécution prend en charge de façon optimisée les mécanismes nécessaires au bon fonctionnement de l'application.
- Elles sont portables, une application active embarque avec elle ses spécificités. Les environnements ad-hoc sont ainsi évités.

Les applications actives permettent d'intégrer rapidement de nouveaux mécanismes dédiés dus à l'émergence de nouveaux domaines applicatifs, mais permettent aussi d'intégrer des mécanismes dédiés lorsque la politique par défaut de l'environnement est inadéquate.

La construction des applications actives nécessite l'existence d'une plate-forme générique que l'application va spécialiser en fonction de ses besoins. Pour adopter cette architecture, il faut être capable de spécifier ce que signifie générique. La solution proposée consiste à construire un environnement le plus générique possible : un environnement qui ne possède

aucune sémantique propre, mais qui peut être spécialisé par ajouts successifs pour, finalement, correspondre à un environnement dédié à une application. Cet environnement neutre est ce que nous appelons la *micro machine virtuelle* [OTGF05, OTP⁺03, OTF05] (μvm).

Les environnements construits par ajouts successifs de spécialisations sont appelés des *Machines Virtuelles Virtuelles* [FPR97, FPR98] (MVV) : ce sont des environnements d'exécution virtualisés. Pour passer d'une machine virtuelle virtuelle à une autre, une spécification d'environnement appelée MVLet est chargée dans une machine virtuelle virtuelle.

Une application active est alors constituée d'un ensemble de MVLets qui agissent sur une MVV et d'une application dite passive, exécutée par l'environnement construit en chargeant les MVLets dans la MVV. L'application passive est une application écrite pour un environnement connu et standard, par exemple un binaire Java ou un fichier PostScript : les applications actives réutilisent au maximum l'existant.

Les autres apports majeurs des applications actives sont :

- La présence d'un mécanisme unique pour configurer ou reconfigurer un environnement d'exécution. Les MVLets peuvent être chargées au démarrage ou pendant l'exécution d'une application.
- La factorisation du développement des MVLets. Une fois une spécialisation d'une MVV développée, elle peut être réutilisée par n'importe quelle application active.
- La compatibilité avec le code applicatif existant. Les applications actives ne font qu'ajouter aux applications standards des mécanismes dédiés et les outils les mettant en œuvre sous la forme de MVLets.
- Une diminution du temps de déploiement de mécanismes dédiés dans des environnements d'exécution existants, c'est directement le développeur d'application qui prend en charge ce déploiement.

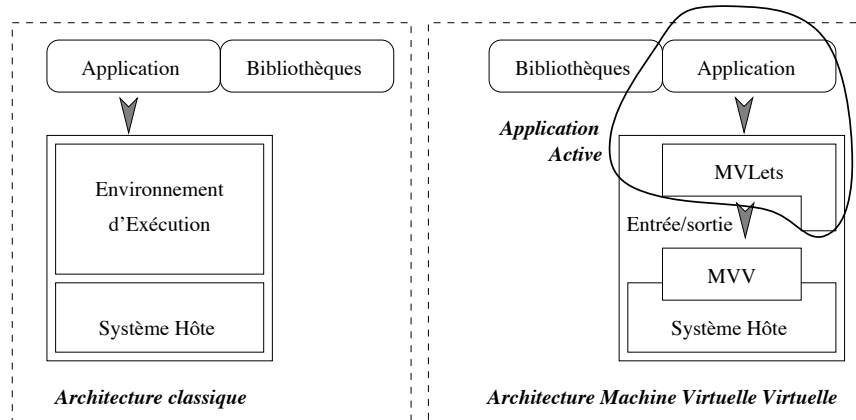


FIG. 3.1 – Architecture des applications actives

La figure 3.1 présente l'architecture classique à gauche et l'architecture Machine Virtuelle Virtuelle à droite. L'environnement d'exécution est scindé en deux parties, une Machine Virtuelle Virtuelle et un ensemble de MVLets. Un MVV possède un canal de communication (noté "entrée/sortie" sur la figure) utilisé pour charger des MVLets. Le chargement des MVLets est dynamique : il n'y a pas de rupture entre configuration et reconfiguration. Contrairement à l'architecture classique, l'architecture des applications actives permet à la

fois à l'application de construire l'environnement d'exécution qui lui convient, mais aussi de le modifier dynamiquement pendant l'exécution du code applicatif.

Les MVLets sont soit chargées dans une MVV, soit contenues dans une application active : aucune hypothèse n'est faite sur l'état de la MVV lors du chargement d'une application active. Une application active peut très bien embarquer avec elle une machine virtuelle Java réflexive complète, ou seulement la spécialisation réflexive. Une fois les MVLets et l'application passive chargées dans une MVV, une nouvelle MVV est construite, contenant l'ancienne MVV, les nouvelles MVLets et l'application passive. Il faut aussi noter qu'une MVLet n'augmente pas forcément une MVV : une MVLet peut très bien supprimer des fonctionnalités d'une MVV.

Cette architecture s'inspire des réseaux actifs [TW96] : l'environnement d'exécution d'une application active est une machine virtuelle virtuelle. La différence entre les deux architectures vient du degré d'adaptation possible. Dans un réseau actif, seuls des points clairement définis dans l'environnement d'exécution peuvent être adaptés (voir la section 4.1 du chapitre 2). Avec nos applications actives, tout l'environnement peut être adapté.

Une application active peut être vue comme un document (l'application) embarquant avec lui son plugin de visualisation (une MVLet). Des travaux sur des applications actives à base de documents sont présentés dans cette thèse dans le chapitre 4. Avec une vision plus exo-noyau, une application active est la réunion de l'application, de ses bibliothèques systèmes et d'un mécanisme d'adaptation dynamique de ces bibliothèques.

Les problèmes de sécurité n'ont pas été abordés dans ces travaux faute de temps. Les applications actives semblent en contradiction avec la notion de sécurité. La sécurité est considérée ici comme un problème applicatif. Sécuriser une Machine Virtuelle Virtuelle revient à supprimer ou vérifier les droits des fonctions dangereuses ou simplement à fermer le canal de communication de la μvm . Comme une MVLet peut diminuer les possibilités d'une MVV, une MVLet de sécurité spécifique à une Machine Virtuelle Virtuelle peut très bien s'occuper d'effectuer ce travail.

La figure 3.2 récapitule le vocabulaire des applications actives.

MVV	un environnement d'exécution pour applications actives
MVLet	un fichier de spécialisation d'une MVV
μvm	une implantation d'une MVV (petite en fonctionnalités et en taille)
Application Active	une application munie de MVLets

FIG. 3.2 – Vocabulaire des applications actives

1.1 Les propriétés de la μvm

Pour répondre aux besoins de flexibilité, de performance et d'extensibilité, la μvm doit offrir un haut niveau de réflexivité pour pouvoir être adaptée, elle doit être capable de charger du code pour être extensible et elle doit être minimale en fonctionnalité pour ne pas imposer de mécanismes particuliers à l'application.

1.1.1 Réflexivité

La μvm doit être adaptable dynamiquement : une MVLet doit pouvoir spécialiser des parties de l'environnement. La réflexivité [Smi82, Mae87a] est le mécanisme le plus approprié pour permettre de connaître et modifier l'état d'un environnement : la μvm doit offrir de puissants mécanismes de réflexion. Cette réflexion doit être de différentes natures pour offrir un large spectre d'outils d'adaptation : introspection et intercession, réflexion structurelle et comportementale. Pour que la μvm soit performante, les mécanismes de réflexion doivent être le moins intrusif possible : les performances de la μvm ne doivent pas être dégradées par ces mécanismes. De plus, toutes les parties d'une MVV doivent être adaptables.

- La μvm : si les mécanismes proposés par la μvm ne sont pas adéquates, une application active doit pouvoir les modifier. La réflexion des structures et du code de la μvm permet à une application active de réutiliser ou étendre les données et les comportements de la μvm .
- Les MVLets et l'application passive : une application active doit aussi pouvoir spécialiser, augmenter ou diminuer les possibilités de l'application ou de l'environnement d'exécution statiquement ou dynamiquement.

1.1.2 Chargement de code

Lors de la construction de la μvm , il faut prendre en compte les problèmes de performance : si on choisit d'enrichir des environnements avec des mécanismes dédiés, c'est pour éviter les pertes de performance dues aux surcouches applicatives. La μvm doit non seulement être performante, mais doit aussi permettre la construction de MVV performantes.

La μvm doit être extensible : il faut être capable d'injecter du code dans la μvm pour la spécialiser. L'interprétation de ce code est hors de question à cause des problèmes de performances. La μvm doit donc être capable de compiler du code dynamiquement ou de charger du binaire dynamiquement. Les deux possibilités existent dans la μvm : la μvm est un compilateur et un éditeur de lien avec des bibliothèques partagées. La première solution offre une utilisation plus souple : il est difficile de modifier ce qui se trouve déjà dans la mémoire d'une MVV uniquement avec des fichiers binaires. La seconde permet aux MVV de s'intégrer avec des bibliothèques existantes. <

1.1.3 Minimalité

La μvm doit être le plus neutre possible par rapport aux MVLets. En effet, seule sa neutralité permet de ne pas imposer de contraintes aux applications actives. En suivant les enseignements des exo-noyaux [vDKV00], on s'aperçoit que le meilleur moyen d'obtenir un environnement neutre est de construire un environnement minimal. La minimalité de la μvm lui évite de ne pas imposer de mécanismes particuliers aux MVLets.

1.1.4 Fonctionnalité la μvm

La μvm est donc un compilateur/éditeur de lien dynamique et réflexif. Un prototype de MVV minimale a été développé par I. Piumarta, la Ynvm¹. Ce prototype présentait plusieurs lacunes pour ces travaux qui nous ont conduit à le modifier et à l'enrichir pour obtenir le

¹Pour Ynvm is Not a Virtual Machine.

prototype de μvm La section suivante décrit la Ynvm et la section 3 décrit la μvm et les extensions réflexives apportées à la Ynvm pour permettre la construction des applications actives. Enfin, la section 4 conclut ce chapitre et récapitule les propriétés de la μvm .

2 La Ynvm

La Ynvm est un compilateur dynamique offrant deux mécanismes de réflexion : un au niveau du langage, un autre au niveau des fonctions internes. La réflexion langage permet aux applications chargées dans la Ynvm de modifier la représentation intermédiaire des fonctions qui vont être compilées. La réflexion des fonctions internes permet à une application d'appeler n'importe quelle fonction interne de la Ynvm. La Ynvm a été développée par I. Piumarta dans le cadre du projet Machine Virtuelle Virtuelle [FPR97, FPR98]. Elle prend en entrée des programmes écrits dans un langage syntaxiquement proche de Scheme². Le fonctionnement global est présenté sur la figure 3.3. Les programmes sont transformés en arbres de syntaxe abstraite (AST, une représentation intermédiaire du code) par un lexeur et un parseur.

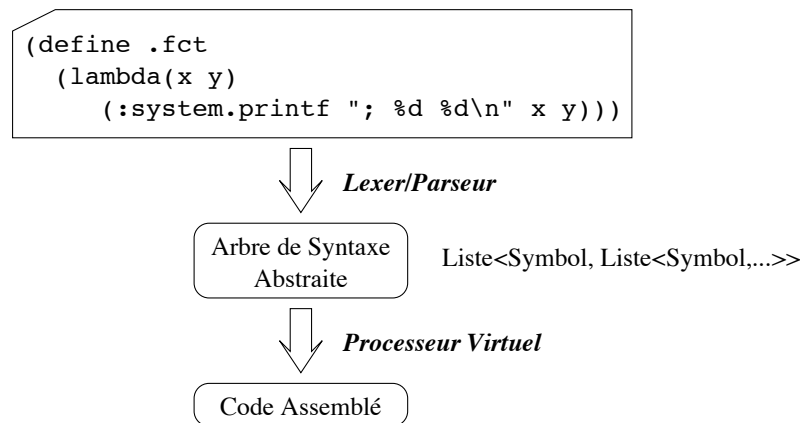


FIG. 3.3 – Compilation dans la Ynvm

Les AST sont ensuite compilés à l'aide d'un processeur virtuel appelé VPU (Virtual Processor Unit) [Piu04]. La VPU est une machine à pile abstraite transformant une séquence d'instruction sur la pile en une fonction assemblée en mémoire. La VPU est donc le générateur de code dynamique : la Ynvm transforme dans programmes en arbres et la VPU génère du code à partir de ces arbres.

Plusieurs implantations de la Ynvm pour différentes plate-formes ont été développées : une version au dessus de Think [FSLM02, SCS02], un exo-noyau à base de composants, et des versions pour systèmes d'exploitation standards (Linux, Windows et MacOS).

2.1 Les Arbres de Syntaxe Abstraite

Les arbres de syntaxe abstraite sont une représentation intermédiaire du code. Ils sont structurés sous la forme de listes représentant des applications fonctionnelles. Un AST est composé d'objets élémentaires compilables :

²Un langage fonctionnel proche de Lisp.

- des entiers codés sur 4 octets ;
- des chaînes de caractères ;
- des listes, elles représentent les applications fonctionnelles ;
- des symboles, ils servent à réaliser l'édition de lien ;
- des modules, ce sont des listes regroupant des symboles dans un espace de nom particulier.

La racine de module est appelée module global, noté `'.'`, et le séparateur de module est noté `'.'`. Ainsi, le symbole `:object.list` correspond au symbole `list` du sous-module `object` contenu dans la racine des modules.

2.2 Compilation et VPU

La VPU est le compilateur dynamique de la Ynvm : c'est un compilateur Just In Time (JIT). Elle transforme une séquence d'actions effectuées sur une pile abstraite (type Post-Script ou bytecode Java) en une fonction assemblée en mémoire. La VPU n'est pas byte-codée : c'est une classe écrite en C++ possédant des fonctions de manipulation de la pile abstraite.

FIG. 3.4 – Utilisation de la VPU

A titre d'exemple, les figures 3.4 et 3.5 montrent une utilisation (en Ynvm) de la VPU et le code produit par cette séquence. Cette fonction génère le code nécessaire à l'addition de deux variables locales (aux index 0 et 1 dans le tableau des variables locales) et à l'appel d'une fonction `f` avec le résultat de l'addition en paramètre. La variable locale à l'index 0 des variables locales est posée en pile, puis la variable locale à l'index 1, et la fonction `f` est appelée avec 1 paramètre (la somme des deux variables locales). Le résultat de cet appel est ensuite posé en pile. La séquence d'instruction VPU est transformée en une fonction assemblée lors d'un appel à la fonction de compilation de la VPU.

FIG. 3.5 – Code compilé en mémoire

La VPU génère un code compatible avec les appels C, c'est-à-dire que des fonctions C compilées peuvent appeler et être appelées par des fonctions générées par la VPU. Les fonctions générées par la VPU peuvent donc interopérer avec des fonctions de bibliothèques partagées (qui respectent une sémantique d'appel C). Cette possibilité est largement exploitée dans les versions de la Ynvm pour systèmes.

La VPU est accessible au niveau utilisateur : une MVLet peut adresser directement la VPU et ainsi construire son propre compilateur en ligne dédié en fonction de ses besoins. Cette possibilité est utilisée dans la MVLet Java (présentée dans le chapitre 5) pour construire un compilateur de bytecode Java Just In Time.

2.3 Réflexivité au niveau langage

Les arbres de syntaxe abstraite décrivent des applications fonctionnelles. Ce sont des listes contenant des objets. Ces listes peuvent être manipulées pour modifier ou simplement connaître le comportement des fonctions. Ce niveau de réflexion est donc comportemental : ce sont les algorithmes de l'application qui peuvent être connus ou modifiés.

La Ynvm utilise aussi un puissant mécanisme d'intercession : les syntaxes. Les symboles Ynvm possèdent trois champs :

- un champ valeur servant à stocker des données ou des fonctions ;
- un champ syntaxe permettant de prendre la main sur le compilateur interne de la Ynvm ;
- un champ objet permettant à une MVLet de stocker des méta-données.

Le champ syntaxe construit de nouveaux mots clés du langage. Si un symbole **f** possède une syntaxe non nulle, toute application fonctionnelle commençant par **f** sera compilée par la fonction rangée dans le champ syntaxe du symbole. Si la syntaxe renvoie un AST, celui-ci sera compilé à la place de l'AST original.

```

1 (define (syntax .f)
2   (lambda (form comp) ;; form référence l'arbre représentant '(f "Hello, World!!!")
3     ;; f est remplacée par :system.printf
4     (set! ( :object.list.at form 0 ) ' :system.printf)
5     form)) ;; renvoie l'arbre '( :system.printf "Hello, World!!!")
6
7 (f "Hello, World!!!") ;; l'application fonctionnelle

```

FIG. 3.6 – Utilisation des syntaxes

La figure 3.6 présente un exemple d'utilisation des syntaxes. La syntaxe **f** remplace le premier élément de l'AST par le symbole **:system.printf**. Ainsi, l'arbre **(:system.printf "Hello, World!!!")** sera compilé à la place de l'arbre original. Cette technique est proche de la notion de macro en Scheme et permet de manipuler ou modifier les applications fonctionnelles automatiquement.

Les syntaxes ne se résument pas uniquement à de la modification d'arbres, le paramètre **comp** de la syntaxe est une référence vers le compilateur qui s'occupe de compiler l'application fonctionnelle. A l'aide de ce paramètre, une fonction syntaxique peut accéder directement à la VPU utilisée par ce compilateur. Cette technique permet à une fonction syntaxique de générer elle même les instructions VPU nécessaires à la compilation de l'application fonctionnelle. La figure 3.7 présente un exemple d'utilisation du compilateur dans une syntaxe. Dans cet exemple, le symbole **f** est défini comme un nouveau mot clé du langage. Ce mot

clé charge automatiquement le nombre 10 sur la pile pendant la phase de *compilation* de l'application fonctionnelle. Le retour de la syntaxe vaut 0 et aucun autre arbre n'est compilé à la place de l'AST original.

```

1 (define (syntax .f)
2   (lambda (form comp) ;; l'arbre représentant "(f 1 2 3)" et le compilateur
3     (let ([vpu ( :compiler.vpu comp)]) ;; la VPU associée au compilateur
4       (ld-int vpu 10)                ;; charge le nombre 10 sur la pile
5       0))                            ;; l'arbre n'est pas remplacé
6
7 (f 1 2 3)                            ;; l'application fonctionnelle

```

FIG. 3.7 – Utilisation de la VPU dans une syntaxe

Un exemple d'utilisation des syntaxes est donné par le compilateur JIT de la MVLet Java : la fonction qui compile le bytecode Java n'est autre qu'une syntaxe qui s'occupe de transformer une séquence de bytecode Java en une séquence de manipulation de la pile de la VPU.

Tous les mots clés de la Ynvm sont définis sous la forme de symboles munis d'une syntaxe. Rien n'empêche une application de modifier ces syntaxes et donc de modifier la sémantique du langage. Ces possibilités du langage sont utilisées par les MVLets pour étendre l'ensemble de mots clés du langage. Ces nouveaux mots clés forment l'ensemble des mots clés de DSL (Domain Specific Language, voir la section 4.2 du chapitre 2) spécifiques à un domaine particulier : l'environnement défini par la MVLet.

2.4 Réflexion fonctionnelle

Tous les symboles C ou C++ utilisés dans le code source de la Ynvm sont accessibles par un programme Ynvm. La fonction `dlsym` trouve l'adresse d'une fonction à l'aide d'une chaîne de caractères représentant cette fonction. La fonction `dlsym` est connue : elle lie dynamiquement les symboles des bibliothèques partagées.

Avec cette introspection, une application active peut appeler directement les fonctions internes de la Ynvm. La plupart des fonctions de base et des mots clés de la Ynvm sont écrits en C++ directement dans le code de la Ynvm³. La fonction `dlsym` est utilisée au démarrage de la Ynvm pour lier dynamiquement les fonctions C++ avec les symboles associés dans la Ynvm. Les fonctions de manipulation de la VPU présentées dans la sous-section précédente sont, elles aussi, liées dynamiquement avec les fonctions C++ de la VPU.

La fonction `dlsym` est aussi utilisée par la Ynvm pour récupérer des pointeurs de fonction dans des bibliothèques partagées. Comme la VPU produit un code compatible avec les bibliothèques partagées, la Ynvm (et par extension les MVLets et les applications) peuvent interopérer dynamiquement avec n'importe quelle bibliothèque partagée.

³Les mots clés de base, comme `define` ou `set !`, et les fonctions de base, comme `dlsym`, ne peuvent pas être écrits dans le langage de la Ynvm sans avoir déjà accès à ces mots clés ou à ces fonctions.

3 La μvm

La Ynvm est un compilateur en ligne et permet donc d'insérer dynamiquement du code dans une MVV. La Ynvm offre aussi un premier ensemble de comportement réflexif avec la possibilité de manipuler les arbres de syntaxe abstraite, mais la Ynvm n'offre pas la réflexion structurelle nécessaire pour adapter les structures internes du moteur d'exécution. Cette lacune empêche une application active de modifier et de réutiliser au niveau applicatif les éléments internes de la Ynvm comme le lexeur, le parseur ou les objets compilables : seuls les objets de la Ynvm peuvent être compilés et le langage d'entrée ne peut pas être redéfini. L'absence de réflexion des structures dans la Ynvm empêche aussi une application active de construire des environnements adaptables dynamiquement : une fois un environnement (une MVLet) compilé et chargé dans la Ynvm, il ne peut plus être modifié car les structures définies par la MVLet sont opaques.

Cette constatation nous a donc conduit à modifier la Ynvm pour la transformer en prototype de μvm . La μvm est une machine virtuelle virtuelle minimale basée sur la Ynvm offrant en plus un modèle objets réflexifs et un protocole méta-objet (MOP). Le MOP est le moins intrusif possible : il ne nécessite aucun crochet dans le code ni aucune réification des structures de données (les structures internes sont directement manipulées par le MOP). La structuration des données pour mettre en place le MOP est très proche des structures générées par la compilation des objets en C++ : l'utilisation du MOP a le même coût en terme de vitesse d'exécution que des appels à des méthodes virtuelles en C++. Le MOP a toutefois un coût en mémoire pour stocker des méta-informations ce qui peut ralentir les phases de collection de la mémoire.

Outre l'absence de réflexion structurelle, la Ynvm présente aussi d'autres limites qui nous ont amené à modifier d'autres aspects du code :

- Les variables globales ne sont pas protégées : la Ynvm n'est pas thread-safe.
- Le ramasse-miettes ne prend pas en compte les processus mutateurs : le ramasse-miettes n'est pas thread-safe.
- L'allocateur du ramasse-miettes de la Ynvm est basé sur la fonction d'allocation de la bibliothèque C standard (le `malloc` de la `glibc`). La Ynvm ne peut donc pas différencier un objet d'un nombre : les fonctions compilées dynamiquement ne peuvent pas être allouées via le ramasse-miettes⁴ et une fonction redéfinie ne peut pas être automatiquement supprimée, ce qui peut conduire à des fuites mémoire.
- Les littéraux (les chaînes de caractères, les listes) utilisés par les fonctions compilées ne peuvent pas non plus être ramassés : toute fonction supprimée ou redéfinie entraîne une fuite mémoire.
- Seuls les entiers signés sur 4 octets sont reconnus par la Ynvm. Les flottants ou les entiers sur 8 octets ne sont pas gérés ce qui rend la Ynvm incompatible avec les fonctions de bibliothèques partagées utilisant ces types. Ce dernier point, faute de temps, n'a pas été résolu, il faudrait intégrer les flottants et les entiers longs directement dans la VPU.

Les modifications apportées pour la gestion multi-tâches sont assez simples (ajout de sémaphores et de pages de mémoire privées au thread). En revanche, deux ramasse-miettes multi-tâches ont été développés pour résoudre les problèmes de fuite mémoire et de collection multi-tâches.

⁴Le ramasse-miettes ne peut pas différencier un Program Counter dans une fonction d'un nombre.

La sous-section suivante présente les deux nouveaux ramasse-miettes, la sous-section 3.2 présente le protocole méta-objet de la μvm et la section 3.3 décrit précisément les composants lexeur et parseur de la μvm .

3.1 Les ramasse-miettes

Pour résoudre les problèmes de fuite mémoire et de concurrence, nous avons développé deux algorithmes de ramasse-miettes dans la μvm : un ramasse-miettes coloré, multi-tâches et incrémental possédant son propre allocateur, basé sur les algorithmes de Boehm [BDS91] et de Dijkstra [DLM⁺78] et un ramasse-miettes coloré, générationnel, multi-tâches et incrémental inspiré de l'algorithme proposé par Lieberman dans [LH83]. Ces deux ramasse-miettes reposent sur un même allocateur mémoire inspiré de l'algorithme proposé par Boehm dans [Boe] (voir section 3.1.2).

Les deux ramasse-miettes présentent des avantages et des inconvénients : si le nombre d'objets est faible ou si un grand pourcentage d'objets alloués restent longtemps en mémoire, l'utilisation du ramasse-miettes générationnel devient inadéquat. Si un grand nombre d'objets a une durée de vie courte, le ramasse-miettes générationnel est préférable.

Il serait possible de passer d'un ramasse-miettes à l'autre dynamiquement, mais ce travail demande un grand effort d'implantation car il faudrait transformer tous les objets du ramasse-miettes d'origine vers le ramasse-miettes de destination et reconstruire toutes les références. Le choix du ramasse-miettes doit actuellement être effectué lors de la compilation de la μvm .

3.1.1 Généralités sur les ramasse-miettes

Les deux ramasse-miettes de la μvm sont des marque-et-balayes (mark-and-sweep). Les objets racines se situent sur la pile d'exécution de la μvm et dans un tableau d'objets racines. Les marque-et-balayes appartiennent à une des trois grandes familles de ramasse-miettes : les marque-et-balayes, les compteurs de références et les stop-et-copies.

Marque-et-balaye. Les marque-et-balayes sont les algorithmes les plus simples à mettre en œuvre dans un programme écrit en C, ce qui est le cas de la μvm . Dans un marque-et-balaye, une tâche va s'occuper de ramasser les miettes à un instant donné. Un certain nombre d'objets sont marqués comme racines et sont marqués vivants et tracés. Ensuite, récursivement, les objets tracés sont marqués vivants et eux même tracés. Cette solution évite les cycles mais pose un léger problème vis-à-vis de la pile d'exécution : tous les objets dans la pile doivent être considérés comme des racines pour la collection. La pile est donc considérée comme un objet et est tracée. La pile est un objet particulier : elle contient des nombres qui peuvent être des références d'objets mais aussi des nombres quelconques (les variables locales entières par exemple). Comme il n'existe pas de description de la pile en C, il faut être capable de différencier les objets des valeurs, ce qui nous a amené à utiliser l'algorithme de l'allocateur mémoire de Boehm [BDS91, Boe] (voir section 3.1.2).

La figure 3.8 présente l'état d'une mémoire à un instant donné. Chaque flèche symbolise une référence. Les objets *A* et *I* sont des racines. Tous les objets atteignables à partir des racines sont marqués vivants et les objets *K*, *L* et *M* sont détruits à la fin de la collection.

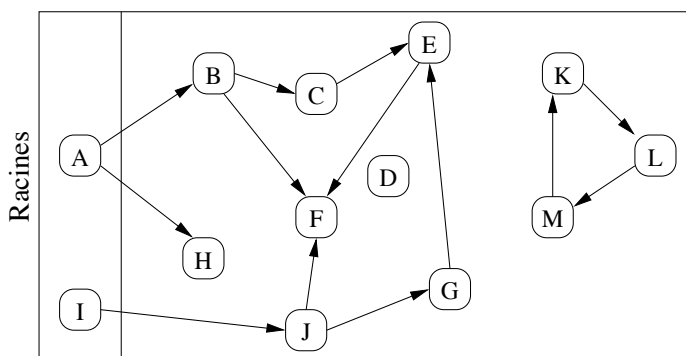


FIG. 3.8 – Une mémoire objet

Stop-et-copie. La deuxième classe d’algorithmes, les stop-et-copies [HB78] sont proches des marque-et-balayés, mais réduisent le temps de destruction des objets et défragmentent automatiquement la mémoire. Les algorithmes stop-et-copies parcourent aussi la mémoire à partir des objets racines, mais chaque morceau de mémoire atteint est déplacé dans une nouvelle zone (des pages différentes) ce qui accélère la phase de destruction des objets : il suffit de libérer les pages où se trouvent encore des objets non atteints. Cet algorithme défragmente aussi la mémoire : les objets atteignables sont regroupés à chaque collection ce qui diminue la dispersion de la mémoire.

Cette classe d’algorithmes présente un défaut pour un programme écrit en C ou C++ qui identifie naturellement les pointeurs et les références. Si les objets sont déplacés, il faut être capable de mettre à jour tous les pointeurs des objets ce qui entraîne une grande perte de temps.

Compteur de références. Dans un algorithme à compteur de références, chaque objet contient le nombre de fois où il est référencé. Sur la figure 3.8, tous les objets ont un compteur égal à 1 sauf *F*, *E* et *D* qui ont un compteur égal respectivement à 3, 2 et 0. Dans un algorithme à compteur de références, il suffit, à chaque affectation d’objet, de vérifier si le compteur tombe à 0. Si c’est le cas, l’objet est libéré. Cet algorithme présente l’avantage de ne pas bloquer l’application pendant une collection, mais pose deux problèmes qui nous ont fait opter pour le marque-et-balayé :

- Lorsqu’il y a un cycle d’objets (comme *K*, *L* et *M*), les objets de ce cycle ne peuvent pas être libérés puisqu’ils ont chacun un compteur de 1.
- Comme le but du ramasse-miettes est d’être utilisé dans la μvm (écrite en C++), à chaque fois qu’un objet est utilisé (via une variable locale ou un registre), il faut être capable d’incrémenter le compteur. Ceci ne peut pas être effectué de manière transparente : il faudrait trouver dans le code de la μvm toutes les variables locales et tous les registres contenant des composants, ce qui est un travail long et fastidieux.

3.1.2 L’allocateur mémoire

Comme décrit précédemment, il est nécessaire de différencier une référence d’un nombre pour pouvoir tracer la pile dans un algorithme marque-et-balayé utilisé dans un langage

identifiant naturellement les pointeurs et les références. L'algorithme d'allocation mémoire de Boehm [BDS91, Boe], basé sur une table de hachage, résoud ce problème. Il est utilisé pour les deux ramasse-miettes de la μvm .

Différencier les références des nombres permet aussi d'allouer les fonctions compilées dynamiquement dans des objets : la présence dans la pile de l'adresse de retour d'une fonction suffit au ramasse-miettes pour atteindre une fonction, même si celle-ci n'est plus référencée nulle part ailleurs. La collection des fonctions compilées dynamiquement automatise la gestion des littéraux et évite les problèmes de fuite mémoire de la Ynvm (voir section 3.1.6).

L'allocateur alloue de la mémoire (comme un `malloc` standard) et est capable de retrouver l'entête d'un bout de mémoire si il a été alloué par l'allocateur. La mémoire est allouée par zones de taille fixe, Z_n , où n est la taille des morceaux de mémoires gérés par cette zone. Lors d'une allocation de k octets, ce nombre est arrondi à l'entier supérieur n le plus proche géré par la zone Z_n . Cette technique entraîne une perte de mémoire puisque la taille allouée peut être supérieure à la taille requise. Trois ensembles de zones sont utilisés :

- Un ensemble exponentiel dans lequel les zones gèrent des puissances de 2 octets. Cet ensemble de zones gère de 0 à 32 octets.
- Un ensemble linéaire dans lequel les zones gèrent des multiples de 32 octets. Cet ensemble de zones gère de 32 à 8192 octets par pas de 32 octets.
- Un ensemble mappé qui ne contient qu'une seule zone et qui gère tous les morceaux de mémoire de plus de 8192 octets. Dans cet ensemble de zones, les morceaux de mémoire sont alloués directement avec la fonction système `mmap()`.

Ces paramètres peuvent être configurés lors de la compilation de la μvm . Cette séparation en trois ensembles minimise le pourcentage de mémoire perdue avec l'arrondi. En utilisant des zones de taille fixe, tous les morceaux de mémoire alloués par le ramasse-miettes peuvent être hachés pour un coût assez faible. Une zone est décrite par un descripteur contenant la taille des morceaux de mémoire gérés par cette zone, un pointeur vers la zone et un pointeur vers les entêtes associées aux morceaux de mémoires contenus dans la zone. Chaque descripteur de zone est inséré dans une table de hachage à deux niveaux indexée par les bits de poids forts pour le premier niveau et les bits suivants pour le second. Les bits de poids faibles du pointeur servent à retrouver l'entête (si elle existe) associée au pointeur dans la zone.

Les deux ramasse-miettes de la μvm sont basés sur cet allocateur. Il ne dépend à aucun moment de l'allocateur standard de la bibliothèque C (`malloc`) : toute la gestion mémoire interne du ramasse-miettes est basée sur des allocations de pages via la fonction système "`mmap`". S'affranchir de la bibliothèque C standard (`glibc`) permet à nos ramasse-miettes d'être utilisés dans la μvm au dessus de Think (qui possède son propre allocateur de pages).

Performances. Les mesures de notre allocateur ont été effectuées sur un PowerPC cadencé à 366MHz. L'allocateur a été comparé à l'allocateur par défaut de la `glibc` (`malloc`). Les résultats ont été calculés sur une séquence aléatoire d'allocations, de réallocations et de libérations. Après chaque allocation, il y a une chance sur dix de réallouer la mémoire, une chance sur dix de ne pas la libérer et huit chances sur dix de la libérer. Les tailles allouées sont aussi tirées de manière aléatoire : il y a une chance sur vingt d'allouer entre 257 et 65536 octets de mémoire et dix-neuf chance sur vingt d'allouer entre 1 et 256 octets. Ce type de séquence d'allocations montre le comportement de l'allocateur dans le cas d'une utilisation intensive.

Deux séries de 524288 allocations sont présentées. Dans la première, nous avons com-

mençé par `malloc`, dans la seconde par l’allocateur du ramasse-miettes.

	nombre de realloc	nombre de free	nombre d’objets	octets à la fin	octets alloués	mémoire du processus	temps
<code>malloc</code>	26021	498527	25761	3.3M	523M	-	1.07s
<code>gc</code>	26021	498527	25761	3.3M	523M	-	0.52s
<code>gc</code>	26374	498355	25933	3.4M	525M	7.2M	0.51s
<code>malloc</code>	26374	498355	25933	3.4M	525M	30.4M	1.12s

La différence de vitesse s’explique par l’utilisation de morceaux de mémoire de taille fixe : pour allouer de la mémoire, l’allocateur de la `glibc` est obligé de parcourir une table des morceaux de mémoire libres, alors que notre allocateur trouve rapidement la zone gérant des morceaux de mémoire de la bonne taille. Le fait de hacher les zones gérées par l’allocateur n’intervient que très peu sur les performances.

Au niveau de la mémoire utilisée, seule la seconde série est significative : en effet notre ramasse-miettes peut libérer toute la mémoire qu’il utilise pour voir le comportement de `malloc`. Notre allocateur est 4 fois plus léger que `malloc` : à part la table de hachage, aucune structure de données supplémentaire n’est nécessaire pour trouver des morceaux de mémoire libres.

3.1.3 Le ramasse-miettes incrémental coloré

Le ramasse-miettes incrémental coloré suit l’algorithme de Dijkstra/Lamport [DLM⁺78] pour être utilisé dans un environnement multi-tâches. Le processus qui collecte bloque la mémoire objet en écriture pour éviter les problèmes de synchronisation. Un bit unique est utilisé pour donner la marque : 0 ou 1. Ce bit est successivement activé et désactivé à chaque collection : tous les morceaux de mémoire atteints par la phase de traçage ont la même marque, ce qui permet d’inverser le bit de marque pour la collection suivante. Cette technique évite de parcourir tous les objets gérés par le ramasse-miettes au début d’une collection pour leur donner la même marque. Les seuls parcours effectués sont donc :

- un parcours linéaire des objets atteignables pendant la phase de marquage ;
- un parcours linéaire des objets à détruire pendant la phase de destruction.

Le temps d’une collection augmente donc linéairement avec le nombre d’octets gérés par le ramasse-miettes.

La gestion de la finalisation (dans Java par exemple, Spécification Java 1.2 [GJSB00], paragraphe 2.17) impose l’appel à une fonction de finalisation avant la destruction d’un objet. Cette fonction peut, d’après la spécification, rendre des objets atteignables. Pour implanter cet algorithme, les objets qui possèdent une fonction de finalisation sont alloués dans une liste spéciale du ramasse-miettes. Lorsque ces objets sont finalisés (avant la phase de destruction), ils sont automatiquement notés atteignables (ainsi que leurs sous-objets) et finalisés. Lorsqu’ils seront de nouveaux notés non-atteignables, ils seront directement détruits. Comme les composants de la μvm ne possèdent aucune fonction de finalisation et comme les classes Java possédant des fonctions de finalisation sont relativement peu nombreuses (classes de Fichier, de Socket principalement), cet algorithme est peu coûteux.

Cette version du ramasse-miettes n’est pas forcément optimale : le temps d’une collection augmente linéairement avec le nombre d’octets gérés par le ramasse-miettes. Il est clair qu’un grand nombre d’objets ont une durée de vie très longue, comme les modules et les symboles μvm ou les instances de classe Java. Une fois une MVLet chargée, une grande part des objets

a peu de chance d'être détruite et augmente considérablement le temps d'une collection, ce qui nous a entraîné à étudier un autre algorithme de marque-et-balaye basé sur une technique générationnelle.

3.1.4 Le ramasse-miettes générationnel

Les ramasse-miettes générationnels [LH83] séparent la mémoire en génération d'objets. Le principe est de collecter plus souvent les objets les plus jeunes et moins souvent les objets les plus vieux. Cette technique répond au problème posé par la durée de vie des objets : plus un objet est présent depuis longtemps plus ses chances d'être détruit diminuent.

La mémoire est séparée en deux générations (jeunes et vieux). Chaque génération est collectée en suivant un algorithme de marque-et-balaye de manière à rester compatible avec du *C*. La mémoire jeune possède une taille fixe. Dès que le nombre d'objets alloués dans cette génération atteint le maximum, la mémoire jeune est collectée. Si à la fin d'une collection un certain pourcentage du maximum reste utilisé (75% dans notre cas), tous les objets sont déplacés dans la zone vieille. La zone vieille est, quant à elle, collectée régulièrement lorsqu'elle augmente. Avec les constantes choisies, environ une collection jeune sur vingt entraîne une collection vieille.

La difficulté réside dans l'implantation. Sur la figure 3.9, un exemple de mémoire générationnelle est présenté. Chaque génération possède un certain nombre de racines qui servent de base à la phase de traçage. On constate sur la figure 3.9 que l'objet *E* n'est atteignable qu'à partir des objets vieux. Si on ne se servait que des racines jeunes (*A*) pour parcourir la mémoire, l'objet *E* ne pourrait pas être atteint et serait libéré.

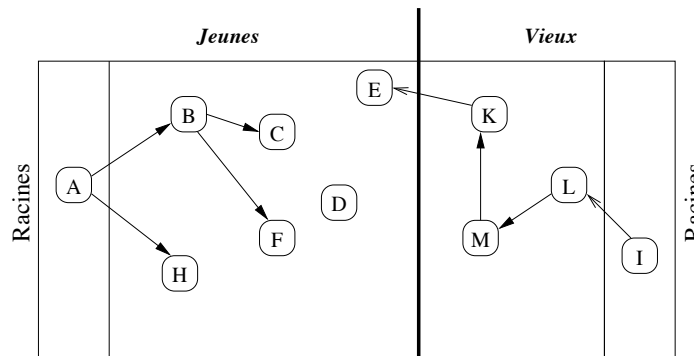


FIG. 3.9 – Problème de génération

Pour remédier à ce problème, il faut vérifier l'âge des objets chaque fois qu'une référence est stockée dans un objet. Lorsque *K* référence *E*, deux solutions sont possibles :

- soit *K* devient une racine temporaire pour *E* ;
- soit *E* et ses sous-objets deviennent vieux.

La première solution semble plus intéressante : si une boucle alloue des objets et les place dans le même champ d'un objet vieux, tous les objets alloués dans cette boucle seront déplacés dans la mémoire vieille alors qu'il suffit de marquer l'objet vieux comme racine temporaire.

A chaque affectation de référence de *E* dans *K*, l'algorithme devient : si *K* est vieux

et si E est jeune, alors K devient une racine temporaire pour les jeunes⁵. Cette technique ralentit l'exécution globale de l'application et n'apporte pas grand chose lorsque la plupart des objets alloués ont une grande durée de vie. Si la durée de vie moyenne des objets est courte (vis-à-vis des constantes choisies), l'algorithme devient de plus en plus performant.

Les performances du ramasse-miettes générationnel sont meilleures que le ramasse-miettes non générationnel lorsque la plupart des objets ont une durée de vie courte mais elles deviennent moins intéressantes lorsque les objets ont une durée de vie longue (voir les performances décrites dans la section 4.4 du chapitre 5) : les deux ramasse-miettes ont donc été conservés pour permettre à l'utilisateur de choisir l'algorithme adéquat en fonction de son application.

Améliorations possibles. Comme chaque affectation de référence entraîne, de fait, une action (la vérification du vieillissement d'un objet), passer à un algorithme à compteur de références chez les jeunes devient envisageable. Les cycles ne posent plus de problèmes car ils sont gérés par les collections dans la mémoire vieille. Cette technique a été utilisée dans la machine virtuelle Java de recherche d'IBM [jik, AP03]. La seule contrainte pour passer à ce type d'algorithme est un effort dans la μvm (écrite en C) pour incrémenter le compteur de références à chaque fois qu'un objet est mis en pile ou dans un registre.

Une autre amélioration utilisée dans la machine virtuelle d'IBM [DGK⁺02] est l'utilisation d'une zone jeune par tâche de manière à ne pas bloquer la mémoire en écriture pendant les phases de collection chez les jeunes objets.

3.1.5 Ramasse-miettes et compilation dynamique

Les performances des ramasse-miettes ont été améliorées par l'utilisation du compilateur en ligne de la μvm (la VPU). Il est possible de diminuer le temps de traçage des objets en n'essayant de marquer récursivement que les références. Si un objet o est constitué de 3 entiers n_i et d'une référence r , alors :

- Si le ramasse-miettes ne connaît aucune information descriptive de l'objet o , il faut (i) vérifier que les nombres contenus dans o sont bien des références à l'aide de l'algorithme décrit dans l'allocateur, (ii) tracer récursivement les références. Cette technique oblige donc le ramasse-miettes à vérifier que les n_i sont des références ce qui, d'une part fait perdre du temps au collecteur et, d'autre part, peut être sujet à confusion si par hasard la valeur d'un nombre est celle d'une référence.
- Si le ramasse-miettes connaît les informations descriptives de l'objet o , il peut ne tracer que la sous-référence r et éviter les deux problèmes précédents.

La seconde technique a été adoptée en laissant à la charge de l'objet de tracer ses propres sous-références. Pour se faire, un vecteur de fonctions est associé à un objet et une des entrées de ce vecteur est une fonction de traçage qui s'occupe de tracer récursivement l'objet. Sans technique de compilation en ligne, la fonction qui trace les objets doit être compilée de manière générique et est obligée de parcourir la description des champs de l'objet à chaque appel à la fonction de traçage. Ce parcours est en général plus coûteux que le fait de vérifier si un nombre est bien une référence.

Comme la μvm compile des fonctions en mémoire à la volée, les fonctions de traçage des objets sont compilées en même temps que la définition d'un objet, ce qui permet d'avoir

⁵Dès que les objets jeunes vieillissent, l'ensemble des racines temporaires est réinitialisé.

des fonctions *spécialisées* en fonction des types d'objets. Le temps de compilation de cette fonction spécialisée est rapidement amorti : il suffit d'une centaine d'appels à la fonction de traçage pour que l'utilisation d'une fonction compilée dynamiquement deviennent rentable.

Les performances des deux ramasse-miettes sont présentées dans la section 4.4 du chapitre 5.

3.1.6 Collection des fonctions et utilisation des objets

Les fonctions assemblées en mémoire par la VPU le sont dans des objets gérés par le ramasse-miettes. De cette façon, un utilisateur n'a pas besoin de gérer les problèmes mémoires lors de la redéfinition d'une fonction. Les littéraux associés aux fonctions sont stockés dans une méta-description de la fonction référencée par le code assemblé en mémoire. Lorsqu'une fonction n'est plus référencée, les littéraux sont alors libérés automatiquement ce qui résout le problème de fuite mémoire évoqué en introduction.

Les références allouées par le ramasse-miettes sont des pointeurs. Les champs d'un objet se trouvent toujours à l'offset 0 de l'objet : il y a identité entre une référence et son contenu. De cette façon, les objets alloués par le ramasse-miettes deviennent compatibles avec les fonctions des bibliothèques partagées. En particulier, une référence vers une fonction assemblée est identifiée à un pointeur C vers une fonction.

3.2 Réflexion des composants

Le modèle de composants et les mécanismes de réflexion développés dans la μvm sont la pierre angulaire des applications actives. En effet, pour modifier l'état d'une MVV, il faut être capable d'injecter du code, mais aussi de modifier la mémoire (les données) d'une MVV. La réflexion des composants est largement utilisée dans la machine virtuelle Java (voir chapitre 5) et dans les différentes extensions apportées à celle-ci (voir chapitre 6).

Le protocole méta-objet (MOP) a été développé dans le langage de la μvm sous la forme d'une MVLet MOP. La construction de ce MOP se sépare en deux parties (voir figure 3.10) :

- La structuration des éléments internes de la μvm sous forme de composants clairement définis et facilement réutilisables au niveau langage.
- La création d'une MVLet MOP offrant les mécanismes de réflexion structurelle des composants. Cette MVLet est définie avec deux sous-MVLets :
 - la MVLet de création de composants, elle génère de nouveaux composants et les méta-données associées à ces composants ;
 - la MVLet des méta-données des structures internes, générée pendant la compilation de la μvm .

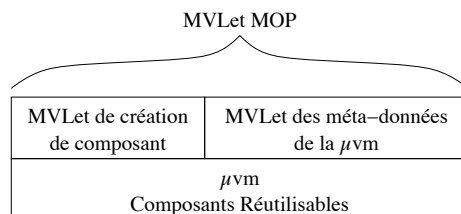


FIG. 3.10 – Structuration du MOP de la μvm

Les composants internes de la μvm (écrits en C) utilisent la structure des composants définis dans la MVLet MOP de manière à avoir une vision uniforme de la mémoire. Les méta-données de ces composants internes sont générées en même temps que le chargement de la MVLet MOP. La construction de la MVLet MOP utilise intensivement la réflexion langage (voir section 2.3). Dans la suite de la thèse, nous considérerons toujours que la MVLet MOP est chargée dans la μvm .

3.2.1 Structure d'un composant

Un composant possède une interface. Une interface est un ensemble de méthodes que doit implanter un composant. Un composant est alors composé d'un ensemble de données (appelées champs) et de l'implantation d'une interface. Les interfaces sont implantées sous la forme de vecteurs de fonctions, appelées tables virtuelles (VT). Pour modifier le comportement d'un composant, il suffit de modifier une des fonctions de ce vecteur. Cette architecture n'est pas forcément la plus adaptable : ajouter dynamiquement des méthodes à une interface revient à étendre la table virtuelle en la réallouant et donc à modifier tous les composants qui implantent cette interface pour mettre à jour leur référence vers leur table virtuelle. L'avantage de cette approche est la simplicité et l'efficacité.

La séparation entre interfaces et implantations donne un cadre composant classique : l'interface permet de faire interagir ensemble des composants ce qui rend leur implantation inter-changeable.

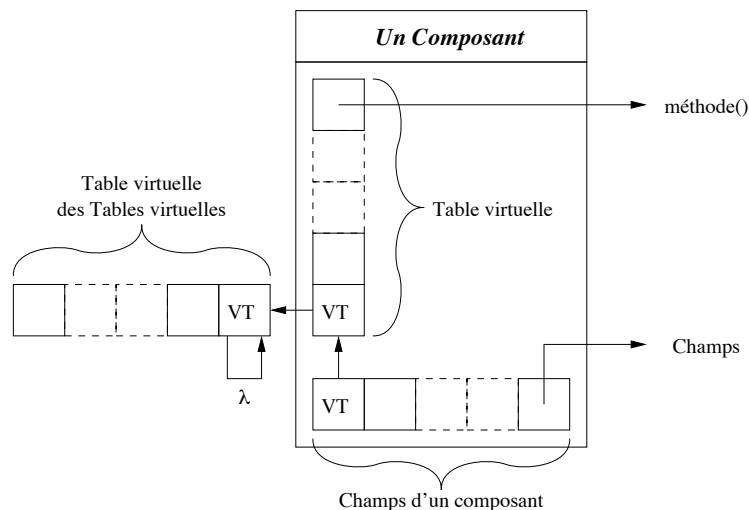


FIG. 3.11 – Structure d'un composant

La figure 3.11 présente un composant. Les tables virtuelles sont elles-mêmes des composants. De cette façon, elles sont collectées, ce qui évite au développeur de MVLet d'avoir à vérifier s'il faut libérer les tables virtuelles. Les tables virtuelles possèdent donc des tables virtuelles. La table virtuelle de la table virtuelle des tables virtuelles (référence λ de la figure 3.11) se référence donc elle-même.

3.2.2 Le MOP extensible

L'accès à la méta-donnée d'un composant est séparée en deux parties, une partie contenant la méta-donnée elle-même et une méthode d'accès à cette méta-donnée. En scindant en deux la donnée et l'accès à la donnée, le MOP gagne en adaptabilité : la méta-donnée peut être réallouée avec d'autres informations et la méthode d'accès peut toujours accéder à l'ancienne méta-donnée. Une méta-donnée est aussi un composant collectable (qui possède elle-même une méta-donnée).

Le contenu de la méta-donnée est laissé libre aux MVLets. Dans la MVLet Java, par exemple, la méta-donnée des objets Java est la description de la classe de l'objet. La MVLet MOP et les composants de base de la μvm proposent une structure par défaut pour les méta-données présentée sur la figure 3.12. Celle-ci se compose de deux parties : l'interface implantée par le composant et les champs du composant. Les structures composants de la méta-interface et du méta-composant ne sont pas représentées sur cette figure. La méta-interface et le méta-composant de la figure sont décrits dans la suite de cette section.

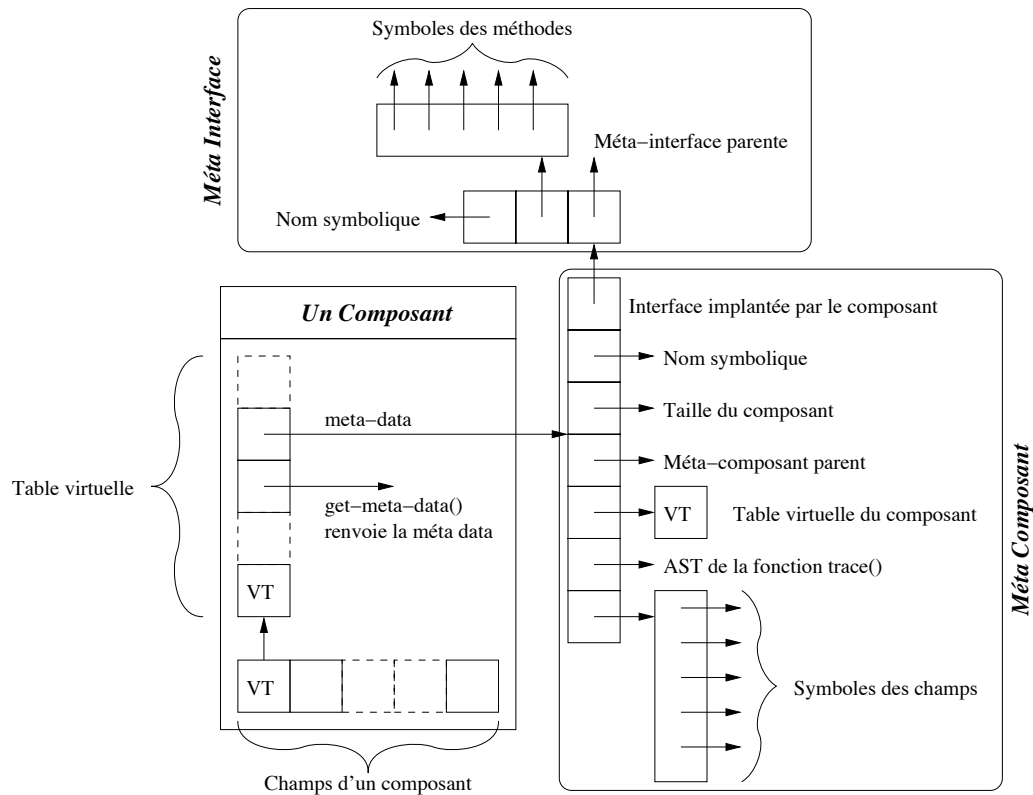


FIG. 3.12 – Architecture d'un méta-composant et de sa méta-interface

Méta-interface. La méta-interface contient une référence vers son interface parente, un nom symbolique (utilisé pour afficher le composant sous forme de chaîne de caractères, voir section 3.2.3) et la liste des symboles des méthodes de l'interface. Cette liste identifie à la fois la taille de l'interface (avec le nombre d'éléments de la liste) et les fonctions d'accès aux

méthodes de l'interface. La MVLet MOP construit automatiquement ces symboles. La réflexion langage héritée de la Ynvm est utilisée pour construire ces symboles. Le champ valeur de ces symboles contient l'offset où se trouve la méthode dans l'interface et le champ syntaxe contient une macro qui insère directement dans le code l'appel à la méthode d'interface.

La méta-interface est utilisée par la MVLet MOP pour implanter l'héritage d'interface. Les symboles des méthodes de l'interface forment le MOP de l'interface. Seul l'héritage simple d'interface a été implanté. Des travaux sont actuellement menés pour utiliser des tables virtuelles et de l'héritage multiple sur le modèle du C++.

Méta-composant. Les méta-composants sont, comme les méta-interfaces, utilisés pour implanter l'héritage de composant. Un composant possède une interface : la méta-interface de l'interface du composant se trouve dans son méta-composant. Un méta-composant contient aussi un nom symbolique pour afficher le méta-composant sous forme de chaîne de caractères (voir section 3.2.3). La liste des symboles du méta-composant tient le même rôle que la liste des symboles de la méta-interface. Ils permettent d'accéder aux champs du composant de manière transparente et de connaître leurs offsets dans le composant. Cette liste n'est pas suffisante pour connaître la taille du composant. En effet, les champs peuvent utiliser un nombre variable d'octets : la taille du composant doit donc être stockée dans le méta-composant.

Lors de la création d'un composant, la MVLet MOP génère automatiquement la table virtuelle du composant. Elle référence les fonctions implantant l'interface du composant. De plus, l'arbre syntaxique de la fonction de traçage (pour le ramasse-miettes) est automatiquement générée lors de la création d'un composant. Une forme compilée de cet arbre est placée dans la table virtuelle (voir les sections 3.1.5 et 3.2.3) et l'arbre est aussi préservé dans le méta-composant. Cet arbre est utilisé pour générer automatiquement les arbres des fonctions de traçage lors d'un héritage.

Le coût du MOP en temps d'exécution est nul : en effet, aucune réification des données n'est nécessaire pour accéder au composant ou pour le modifier. Les appels de méthodes et les accès aux champs sont mis en ligne dans le code : il n'y a pas de perte de performance, en dehors de l'utilisation de tables virtuelles. Les tables virtuelles du MOP sont assez similaires aux tables virtuelles C++. Les performances sont donc équivalentes à des appels de méthodes virtuelles en C++. En revanche, le coût en mémoire n'est pas nul. Pour chaque composant ou interface défini, une méta-donnée et quelques symboles sont générés. Ils utilisent de la place en mémoire. Comme le nombre de composants définis dans une MVLet est assez faible, le coût en mémoire reste relativement bas, mais augmente légèrement le travail que doit réaliser le ramasse-miettes.

La gestion mémoire est aussi totalement transparente. Les méthodes des tables virtuelles (voir section 3.1.6), les champs et les méta-données sont automatiquement collectés par le ramasse-miettes : ce sont des composants. Le développeur de MVLet n'a donc pas à prendre en charge cet aspect.

3.2.3 Composants internes

Les composants internes de la μvm respectent la structure des composants de la MVLet MOP. Ils possèdent une table virtuelle et des champs. Leurs méta-données sont générées dans une MVLet lors de la compilation de la μvm . Cette MVLet est chargée en même temps

que la MVLet MOP et construit automatiquement les méta-composants associés aux objets internes de la μvm : le compilateur, le lexeur, le parseur et les objets des AST.

De cette façon, la structure interne de la μvm est réfléchiée au niveau applicatif. Une MVLet peut donc modifier le comportement de la μvm dynamiquement. De plus, les composants internes de la μvm peuvent être hérités dans une MVLet. Cette possibilité a été exploitée lors de la construction de la MVLet XML (voir section 2.2 du chapitre 6).

Le ramasse-miettes et le MOP imposent à tous les composants d'implanter une interface minimale constituée de 5 entrées :

- **trace()** qui s'occupe de tracer le composant (utilisée par le ramasse-miettes) ;
- **finalize()** qui permet de finaliser un composant (utilisée par le ramasse-miettes) ;
- **print()** qui affiche le composant sous forme de chaîne de caractères (utilisée par les MVLets) ;
- **meta-data** qui stocke la méta-description du composant (utilisée par le MOP) ;
- **get-meta-data()** qui renvoie la méta-description du composant (utilisée par le MOP).

Les ramasse-miettes utilisent les tables virtuelles pour trouver les fonctions de finalisation et de traçage des composants (voir sections 3.1.3 et 3.1.5). La méthode **print()** est une méthode permettant d'afficher des informations sur un composant dans le composant chaîne de caractères des AST. Cette méthode est une facilité offerte au développeur de MVLets pour trouver les erreurs dans son programme. Les deux dernières entrées permettent de construire le MOP adaptable.

Les objets compilables formant les AST possèdent en plus une fonction de compilation (appelée **compile()**) utilisée par le compilateur de la μvm . Une MVLet peut donc modifier la manière dont un élément d'un AST est compilé.

3.3 Lexeur et Parseur

Le lexeur et le parseur sont structurés sous la forme de composants. Chaque table virtuelle associe des entiers codés sur 1 octet à des fonctions. Les fonctions du lexeur renvoient des numéros qui identifient le jeton (token) lu et remplissent une chaîne de caractères ou un entier contenant la valeur du jeton. Les fonctions du parseur associent les jetons du lexeur aux fonctions qui construisent les éléments des AST.

La figure 3.13 présente un exemple de fonctionnement du lexeur et du parseur. L'entrée (le composant Input sur la figure) lit le caractère 'd'. Dans la table virtuelle du lexeur, l'entrée du caractère 'd' correspond à la fonction de lecture d'un identifiant. Celle-ci remplit le texte en cours avec les octets lus, d, e, f, i, n, e, et renvoie le jeton des symboles (ID sur la figure). Dans le parseur, la fonction à cette entrée génère un symbole à partir du texte du lexeur : le symbole **define**. C'est cet objet qui est renvoyé par le parseur. De la même façon, l'entrée correspondant au caractère '0' remplit le nombre lu avec la valeur 0 et renvoie le jeton des entiers (Int sur la figure). La fonction stockée à l'index Int du parseur génère le composant entier 0.

A l'aide du MOP de la μvm , une MVLet peut modifier les fonctions des vecteurs du lexeur et du parseur et ainsi modifier le langage d'entrée de la μvm . En modifiant les fonctions de la table virtuelle du lexeur, une nouvelle syntaxe peut être mise en place. En modifiant les fonctions de la table virtuelle du parseur, de nouveaux objets compilables avec une nouvelle sémantique peuvent être générés. Dans les sections 2.1 et 2.2 du chapitre 6 deux exemples d'utilisation de ces possibilités sont présentées pour construire une entrée pour des arbres sérialisés et une entrée pour le langage XML.

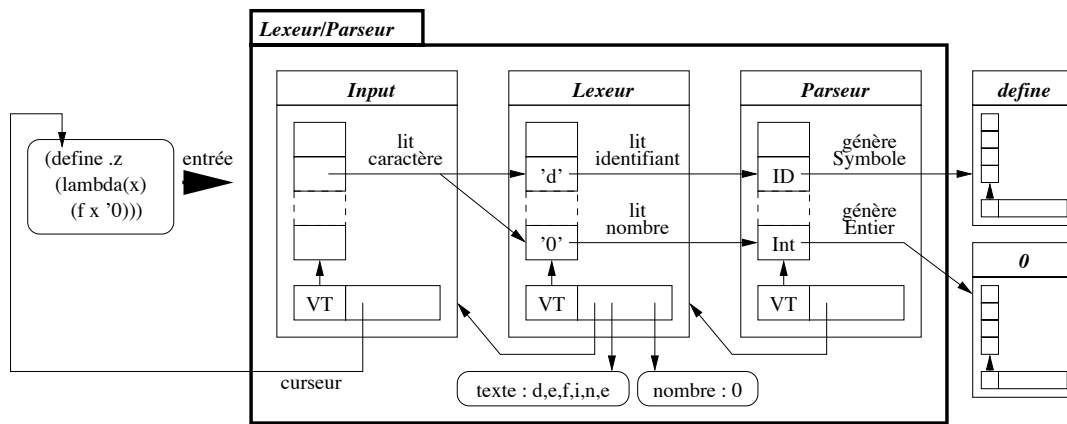


FIG. 3.13 – Fonctionnement du lexeur et du parseur

4 Récapitulatif

L'architecture des applications actives présentée dans ce chapitre répond aux besoins d'évolutivité des environnements lors de l'émergence de nouveaux besoins, tout en gardant une homogénéité des environnements d'exécution. Cette architecture répond aux problèmes posés dans le chapitre 2 et réconcilie adaptation, performances et portabilité. L'environnement d'exécution des applications actives, la μvm , offre aux MVLets un compilateur en ligne et trois niveaux de réflexion :

- Une réflexivité comportementale définie dans la Ynvm : les AST peuvent être manipulés par une MVLet.
- Une réflexivité structurelle définie dans la μvm : les structures internes de la μvm , les composants des MVLet et les composants applicatifs possèdent une méta-description, et une MVLet peut modifier leurs comportements (intercession) en modifiant un des pointeurs dans la table virtuelle ou un des champs des composants.
- Une introspection fonctionnelle définie dans la Ynvm : toutes les fonctions internes de la μvm peuvent être utilisées par une MVLet (introspection de la Ynvm).

La réflexion et le compilateur dynamique sont utilisés par les MVLets pour modifier, enrichir ou diminuer l'environnement d'exécution en fonction des besoins spécifiques de l'application. Deux MVLets majeures valident l'approche, une MVLet documents actifs (voir chapitre 4) qui offre la possibilité à un document de transporter avec lui son code de visualisation, et une MVLet Java (voir chapitre 5) spécialisable par d'autres MVLets (aspects, réflexivité, migration, analyse d'échappement, voir chapitre 6).

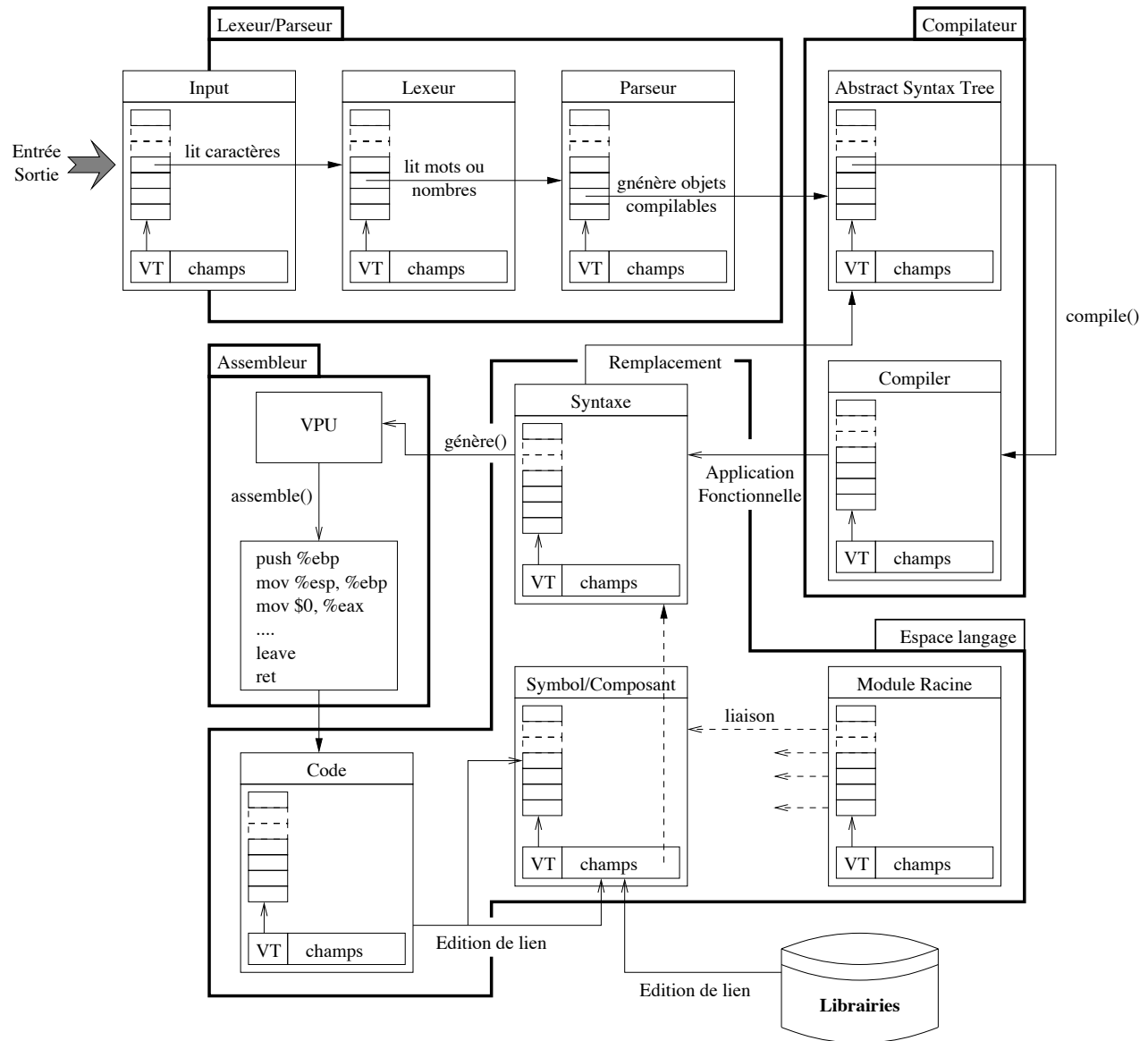
La figure 3.14 présente une vue d'ensemble des différents composants de la μvm . Le lexeur et le parseur sont décrits dans la section 3.3, les phases de compilation dans la section 2 et le modèle composants dans la section 3.2.2.

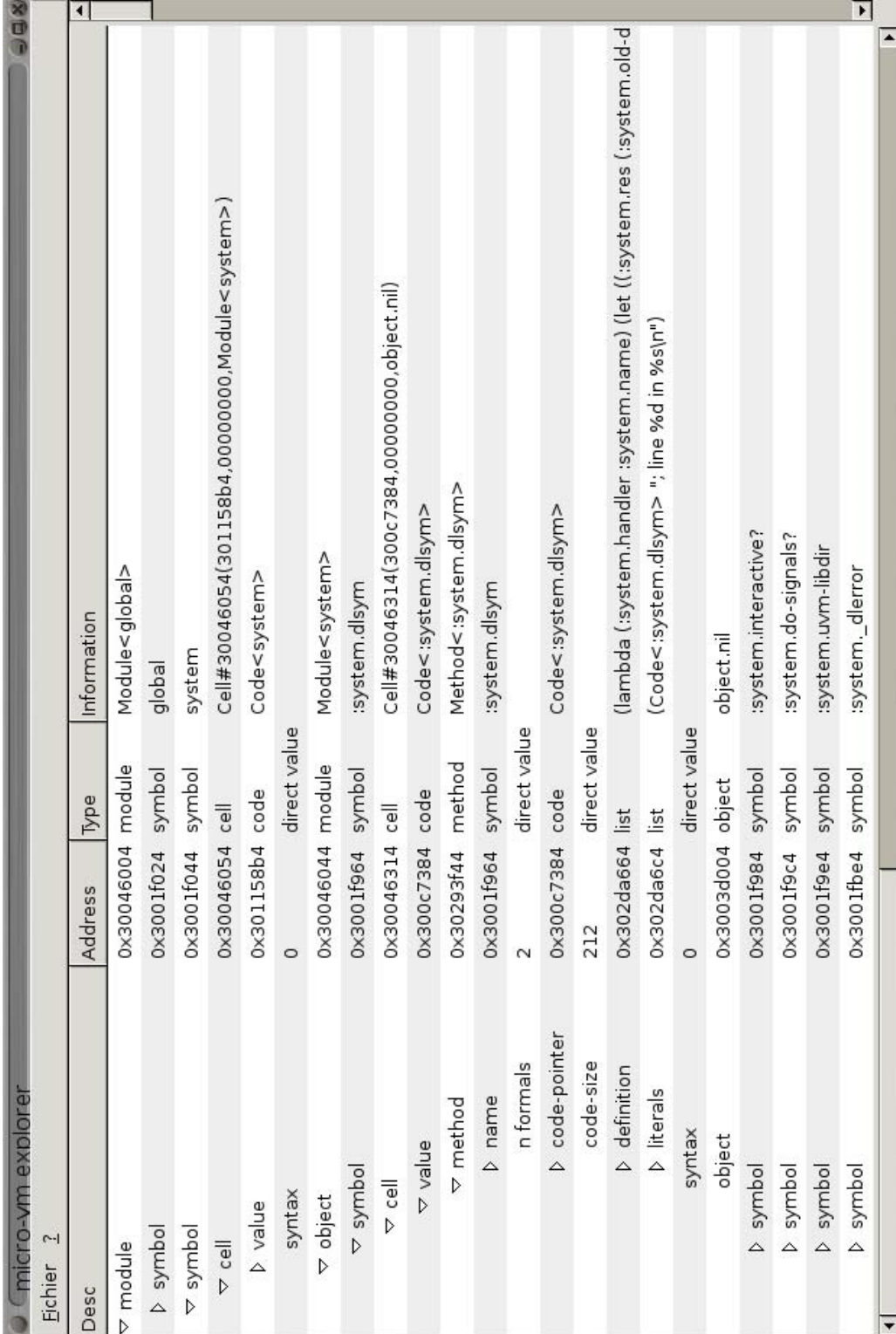
En conclusion, une MVLet est présentée : une interface graphique d'introspection de la μvm , appelée la MVLet explorer. Cette MVLet s'occupe d'afficher graphiquement l'état de la mémoire de la μvm . Une bibliothèque graphique (**gtk**) s'occupe de l'affichage. Elle est chargée et liée dynamiquement avec la μvm pendant l'exécution de la MVLet explorer. Cet aspect illustre les possibilités de réutilisation de bibliothèques externes.

La figure 3.15 donne un aperçu du rendu graphique. Le module `global` contient le symbole `:system` qui lui même contient une cellule⁶. Cette cellule référence le module `:system`. Le champ valeur du symbole `:system.dlsym` est présenté : c'est un composant `Code` (donc un pointeur de fonction) qui référence la méthode associée. On trouve dans cette méthode différentes informations, en particulier l'expression μvm qui a permis de compiler la fonction `:system.dlsym` dans le champ `definition`.

N'importe quelle application chargée dans une MVV peut utiliser cette MVLet pour du debuggage, ou simplement pour connaître l'état de la MVV à un instant donné. Cette interface graphique peut aussi être utilisée à des fins pédagogiques pour comprendre les structures internes de la μvm .

⁶Les 3 champs d'un symbole, `value`, `syntax` et `object`.

FIG. 3.14 – Architecture de la μvm



Desc	Address	Type	Information
▽ module	0x30046004	module	Module<global>
▷ symbol	0x3001f024	symbol	global
▽ symbol	0x3001f044	symbol	system
▽ cell	0x30046054	cell	Cell#30046054(301158b4,00000000,Module<system>)
▷ value	0x301158b4	code	Code<system>
syntax	0	direct value	
▽ object	0x30046044	module	Module<system>
▽ symbol	0x3001f964	symbol	:system.dlsym
▽ cell	0x30046314	cell	Cell#30046314(300c7384,00000000,object.nil)
▽ value	0x300c7384	code	Code<:system.dlsym>
▽ method	0x30293f44	method	Method<:system.dlsym>
▷ name	0x3001f964	symbol	:system.dlsym
n formals	2	direct value	
▷ code-pointer	0x300c7384	code	Code<:system.dlsym>
code-size	212	direct value	
▷ definition	0x302da664	list	(lambda (:system.handler :system.name) (let ((:system.res (:system.old-d
▷ literals	0x302da6c4	list	(Code<:system.dlsym> "; line %d in %s\n")
syntax	0	direct value	
object	0x3003d004	object	object.nil
▷ symbol	0x3001f984	symbol	:system.interactive?
▷ symbol	0x3001f9c4	symbol	:system.do-signals?
▷ symbol	0x3001f9e4	symbol	:system.uvm-libdir
▷ symbol	0x3001fbe4	symbol	:system._derror

FIG. 3.15 – Introspection dans la μvm

CHAPITRE 4

Machines Virtuelles Virtuelles pour documents actifs

Sommaire

1	Introduction	65
2	Dynamicit� des documents	67
3	MVLets pour document multim�dia	68
4	Deux adaptations des MVLets de documents actifs	72
5	Qualit� de service	75
6	Conclusion et perspectives	78

1 Introduction

Traditionnellement, un document est vu comme la donn e d'un programme de visualisation : il est manipul  par un programme externe et c'est   ce programme d'offrir les m canismes n cessaires au bon fonctionnement du document. Cette approche contraint donc un document   respecter les interfaces offertes par son programme de visualisation. Or, avec l'arriv e du num rique et des  changes massifs de documents multim dias, de nouveaux probl mes apparaissent lors de la conception, de la r alisation et de la visualisation de documents :

- L'h t rog n it  des formats des documents implique un grand nombre de programmes de visualisation. D'une part, tous les programmes de visualisation peuvent ne pas  tre pr sents sur une plate-forme, d'autre part, faire interagir des documents de nature diff rente pour obtenir des documents multim dias demande des plate-formes sp cifiques   chaque composition de documents.
- L'h t rog n it  des supports de visualisation a des cons quences sur le rendu du document. Par exemple, il est difficile d'adapter un document pr vu pour un ordinateur de bureau sur un PDA (Personal Digital Assistant).
- La qualit  de service (QoS) requise pour un type de document particulier d pend fortement de l'encodage et du contenu du document. Les plate-formes de visualisation offrent rarement la possibilit  de modifier la qualit  de service offerte par la plate-forme h te.

- Les droits d’auteur sont difficiles à gérer. Comme un document est une donnée, il ne peut pas se protéger du piratage. Les réseaux peer-to-peer type Gnutella [Gnu00] illustrent parfaitement le problème : une fois qu’un utilisateur possède un document, rien ne l’empêche de le donner à un autre utilisateur.
- La réplication et la cohérence posent des problèmes lors de la mise à jour des documents. Si un document est répliqué sur plusieurs sites, les mécanismes permettant de diffuser les modifications à partir d’une copie primaire existent pour les pages Web mais ne sont pas généralisés pour d’autres types de documents. Il est difficile, lorsqu’on consulte un fichier sur un site particulier, d’être sûr de lire la dernière version du document.

La solution proposée pour résoudre ces différents problèmes consiste à considérer le document comme un programme. Le document peut alors construire la plate-forme spécifique de visualisation nécessaire à son bon fonctionnement. Pour adopter cette approche, un document standard transporte avec lui du code exécutable qui s’occupe de spécialiser une plate-forme de visualisation générique. Le document prend alors en charge lui-même les problèmes non résolus de son programme de visualisation par défaut. Le document devient *actif* [TFP02] : il agit sur son environnement pour le spécialiser en fonction de ses besoins.

Les documents actifs réutilisent des documents existants. En effet, il ne s’agit pas de créer de toute pièce de nouvelles plate-formes de visualisation ou de nouveaux types de documents mais d’offrir la possibilité à des documents standards d’enrichir ce qui existe à l’aide du code exécutable du document actif. Plusieurs MVLets pour documents actifs ont été développées pendant ces travaux :

- Une MVLet pour document multimédia permettant de mélanger différents types de documents ensemble (images, vidéo, son, texte).
- Une MVLet de génération de MVLets de documents multimédia permettant à un utilisateur de décrire graphiquement son document multimédia.
- Une MVLet pour de l’ordonnancement de vidéo permettant à un document de modifier, en fonction de ses besoins en qualité de service, la façon dont le système lui attribue le processeur. Cette MVLet s’intègre avec la première pour obtenir des documents multimédia et auto-ordonnés.

Les prototypes présentés dans ce chapitre répondent aux problèmes de composition de documents hétérogènes et aux problèmes de qualité de service. Les autres problèmes évoqués, l’hétérogénéité des supports, les droits d’auteur et la cohérence entre réplicats, peuvent aussi être résolus avec les documents actifs.

Dans la section suivante, d’autres solutions actives existantes sont présentées et la notion d’activité dans les documents est explorée. L’architecture utilisée par les documents actifs multimédias et le prototype développé pendant ces travaux sont décrits en détail dans la section 3. La section 4 présente deux adaptations de cette MVLet, l’une construit un environnement de conception de documents actifs, l’autre produit des fichiers PostScript standards à partir de documents actifs PostScript. La section 5 présente des documents actifs auto-ordonnés et montre comment étendre le prototype de documents actifs avec une gestion particulière de la qualité de service. Enfin, la section 6 conclut ce chapitre.

2 Dynamicité des documents

La limite entre document passif et actif est difficile à donner : on ne peut que donner un degré d'activité. En effet, une expression comme : `` peut être vue comme une marque HTML (donc passive) ou comme une commande permettant de lancer un client mail (donc active). Plusieurs solutions actives ont déjà été proposées. Deux tendances se dessinent : l'approche script et l'approche plate-forme à agents (ou objets).

2.1 L'approche script

L'approche script permet de spécifier, via des scripts, le comportement d'un document comme dans HTML, XML ou SGML [EvOS96, vOES96]. Les scripts permettent aussi d'étendre l'expressivité d'un type de document.

Mobisaic [VB94] offre la possibilité à un document HTML de s'abonner à certaines variables d'environnement physique (dont la position géographique du lecteur et l'heure courante). L'abonnement à ces variables permet aux documents de s'adapter automatiquement lorsque la plate-forme de visualisation change de contexte. Un exemple typique de l'utilisation de Mobisaic est la mise à jour automatique d'une page Web en fonction du bureau dans lequel se trouve l'utilisateur pour afficher les informations pertinentes de ce bureau. Mobisaic présente l'avantage de ne pas avoir à modifier les serveurs Web : c'est directement le navigateur Web qui construit les requêtes en fonction des changements d'état des variables souscrites.

Les approches Scripts sont en général peu actives : les scripts ne sont pas exécutables. Elles permettent néanmoins de résoudre simplement un certain nombre de problèmes comme la composition de médias ou l'évolution temporelle d'un document (comme SMIL [smi] dans un document XML par exemple).

2.2 Les approches agents

Les solutions agents sont, quant à elles, une vision plus récente des documents actifs. Les agents semblent très bien adaptés pour rendre les documents plus autonomes et adaptatifs en y introduisant des buts et des méthodes. Les approches agents sont assez proches de l'approche document actifs : un agent est un programme qui contrôle l'état d'un document.

Parmi les méthodes agents (ou objets répartis), on trouve Globule [PvS01], construit au dessus de Globe [SHT99], qui permet aux documents de s'auto-répliquer ou le serveur vidéo [JBY00] construit au dessus de Bond [BM00] qui permet à un document de spécifier à son serveur la QoS dont il a besoin.

Ces solutions sont assez prometteuses car elles font intervenir plusieurs acteurs : des clients d'un document (les lecteurs) et les serveurs du document. De cette façon, le client peut agir sur le serveur et l'adapter en fonction de ses besoins. Les approches agents ne sont pas soumises aux contraintes traditionnelles des documents : le document est une donnée manipulée par un programme particulier, un agent. Toutefois, un document encapsulé dans un agent ne peut adapter que les mécanismes prévus par la plate-forme agent. Le nombre de points d'adaptation reste figé et ne peut pas être étendu en fonction des besoins particuliers du document.

2.3 Script versus Agent

Notre approche est assez différente des scripts puisque les documents embarquent avec eux du code exécutable, mais rentre aussi difficilement dans le cadre agent puisque les documents n'ont pas de buts : nos documents actifs sont des programmes dont les données manipulées sont des documents. Les documents actifs englobent les approches scripts et agents. Le code exécutable étend l'expressivité d'un document, comme avec les approches scripts, et un document actif peut embarquer avec lui une plate-forme agent.

3 MVLets pour document multimédia

Nos travaux sur les documents multimédias montrent comment composer et synchroniser des documents provenant de sources différentes (image, son, texte et vidéo). La composition de document peut déjà être réalisée dans des cas précis, comme l'ajout d'un type MIME [FBM⁺96] quelconque dans un document HTML, mais ne peut pas l'être avec des formats qui n'offrent pas les abstractions nécessaires à la composition.

Nous considérons que chaque média est exécuté avec son programme de visualisation standard : Ghostscript pour du PostScript ou du texte brut, Mplayer pour des vidéos, Display pour des images et Mpg123 pour de la musique eu format MP3. Chaque exécutable est encapsulé par une surcouche logicielle permettant au document d'interagir avec les autres. Ces surcouches logicielles permettent de voir les exécutables comme des composants auxquels on envoie des messages. La partie active du document s'occupe principalement d'implanter cette surcouche entre les différents exécutables intervenant dans la composition du document. A chaque surcouche correspond une MVLet spécifique. Ainsi, le nombre de formats de documents gérés peut facilement être enrichi en écrivant des MVLets pour de nouveaux types de documents.

Pour faire interagir des documents ensemble, il faut définir des points de jonction entre les différents documents. Le choix des points de jonction est arbitraire et dépend de ce que l'utilisateur attend de son document multimédia. Notre choix s'est porté sur la notion de page et de position sur la page. Une description temporelle aurait tout à fait été possible. Un document sert de support pour décrire la position des autres documents. Ce document privilégié est appelé document primaire et les autres sont appelés post-it : ce sont des notes insérées dans le document primaire. La notion de page et de position sur une page est liée aux documents contenant du texte : nous avons donc adopté un document texte, le PostScript, comme document primaire.

La PostScript est interchangeable avec tout autre document contenant la notion de page. Le travail à réaliser pour effectuer ce changement consiste à développer une nouvelle MVLet pour ce type de document. Ce travail est assez rapide : la MVLet pour PostScript ne fait que 410 lignes de code.

3.1 L'architecture des documents actifs

Les documents actifs sont conçus comme des applications actives (voir figure 4.1). Les MVLets sont chargées dans la μvm et s'occupent de faire interagir les différents documents entre eux en utilisant leur programme de visualisation standard.

La figure 4.2 présente l'architecture des documents actifs. Le document est composé de sous-documents : un document primaire PostScript, des documents texte, des images,

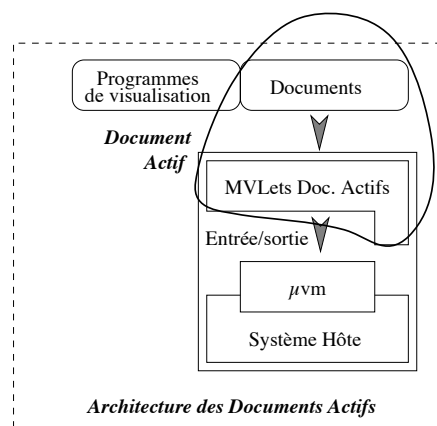


FIG. 4.1 – Architecture globale des Documents Actifs

des vidéos et des fichiers son au format MP3. Chaque sous-document est encapsulé avec son programme de visualisation dans un composant logiciel. Ce composant interagit avec le composant du document primaire pour synchroniser les documents entre eux. L'architecture est extensible : les composants doivent respecter une interface contenant une méthode de démarrage et une méthode d'arrêt du document. Pour ajouter un nouveau type de document, il suffit d'écrire le composant qui encapsule le programme de visualisation associé au document.

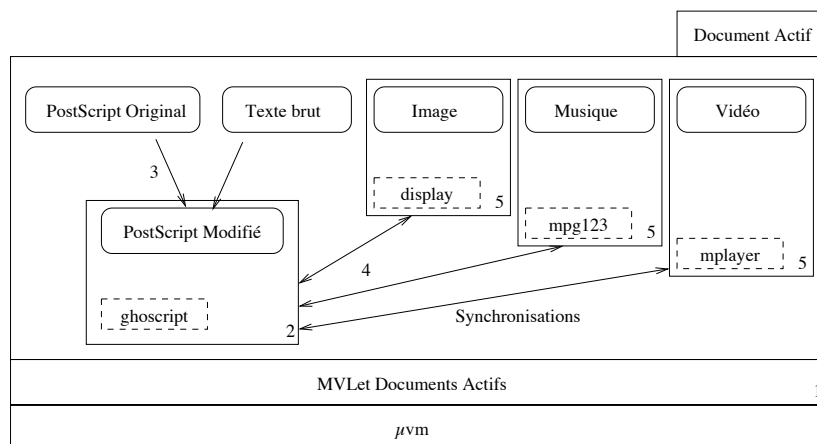


FIG. 4.2 – Architecture des documents actifs multimédias

Le texte brut et le document PostScript sont mixés ensemble pour donner un nouveau document PostScript contenant le texte. Ensuite, ce document est affiché par la MVlet PostScript et, en fonction de la page, les autres documents sont exécutés par leurs MVlets respectives à la bonne position.

3.2 Réalisation des Documents Actifs

Dans la suite de cette section, nous détaillons les différents éléments des documents actifs présents sur la figure 4.2.

3.2.1 La MVLet pour documents actifs

La MVLet de documents actifs (case 1 sur figure 4.2) est le pivot des autres MVLets. Elle contient la définition du langage de description de nos documents actifs et les mécanismes de synchronisation. Cette MVLet est enrichie par les autres MVLets (PostScript, vidéo, MP3, image et texte) pour effectivement exécuter le document actif multimédia. Les points de jonction utilisés par les documents actifs, la page et une position sur la page, sont imposés par cette MVLet.

La figure 4.3 présente un exemple de document actif sous la forme d'un programme. Le document actif charge la MVLet de documents actifs et les gestionnaires de types MIMES [FBM⁺96] audio, video et image. Ensuite, le document spécifie le nom du fichier PostScript à annoter : `main.ps`. La suite du fichier décrit les notes à insérer : à la ligne 8, une note texte ayant une police `Helvetica` et une taille de 18 points est ajoutée à la position (20%, 10%) de la page 1 du document. A la ligne 14, une note de type vidéo est insérée à la page 3. La vidéo est lancée à la position (20%, 60%) de la page sur une surface de (60%, 30%) de la page. La figure 4.4 montre une capture d'écran de la page trois pendant l'exécution de la vidéo.

```

1 (require-in "doc-actif" 'doc-actif-path)
2 (require 'doc-actif)
3 (require 'primaire-postscript)
4 (require 'mime-audio-mpeg)
5 (require 'mime-video-mpeg)
6 (require 'mime-media-image)
7 ( :da.file "main.ps")
8 ( :da.ajoute-note 1 "text/ps" "(Une note texte)" "/Helvetica" 18 200 100)
9 ( :da.ajoute-note 2 "text/ps" "(Petite pose musicale)" "/Helvetica" 18 500 50)
10 ( :da.ajoute-note 2 "audio/mpeg" "test.mp3")
11 ( :da.ajoute-note 2 "text/ps" "(Le chat)" "/Helvetica" 18 500 700)
12 ( :da.ajoute-note 2 "media/image" "lechat.jpg" 200 300 600 300)
13 ( :da.ajoute-note 3 "text/ps" "(La note video)" "/Helvetica" 18 500 100)
14 ( :da.ajoute-note 3 "video/mpeg" "rigolo.mpeg" 200 600 600 300)
15 ( :da.run :system.argv :system.argv)

```

FIG. 4.3 – Exemple de document actif

Les mots clés de la figure 4.3 sont définis dans la MVLet pour documents actifs. Le langage construit par cette MVLet reste syntaxiquement un dialecte de Lisp, mais la sémantique des mots clés forme un DSL (voir la section 4.2 du chapitre 2) dédié à la composition de documents multimédia. Comme tout DSL, ce langage est concis (quelques mots clés),

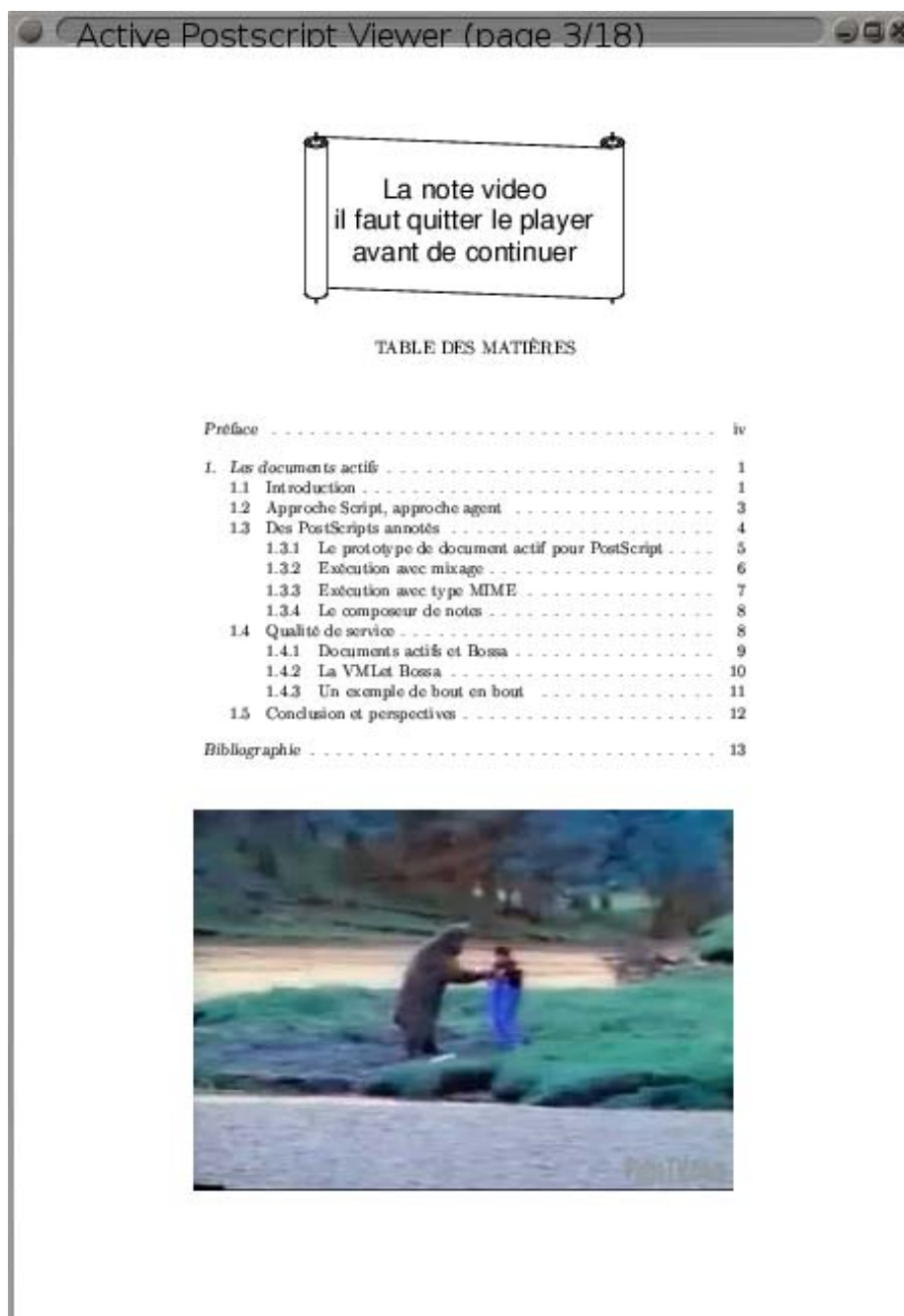


FIG. 4.4 – Une vidéo dans un PostScript

masque la complexité du domaine et est vérifiable pendant la compilation¹. Contrairement aux DSL présentés dans l'état de l'art, la compilation des fichiers de description de documents multimédia se fait au plus tard, c'est-à-dire dynamiquement pendant l'exécution.

Les différents éléments intervenant dans la définition du document sont accumulés dans des listes chaînées permettant aux autres MVLets de se synchroniser. Une fenêtre parente est créée. C'est elle qui sert de base pour l'affichage. Cette fenêtre ne contient rien : ce sont les MVLets des documents qui vont s'occuper de remplir cette fenêtre.

3.2.2 Les MVLets Documents

Chaque MVLet pour document (PostScript, texte brut, son, image et vidéo sur la figure 4.2) s'occupe d'encapsuler l'exécutable naturellement associé au type de média (respectivement GhostScript, GhostScript, Mpg123, Display et Mplayer). Les synchronisations entre la MVLet pour documents actifs et les MVLets pour documents sont effectuées en utilisant les outils systèmes adéquats en fonction du programme de visualisation : signaux, message entre fenêtre, tube ou ligne de commande. La tableau 4.5 récapitule les différents mécanismes².

Signal	Message	Tube	Ligne de commande
Display, Mplayer Mpg123	GhostScript	GhostScript	Display, Mplayer Mpg123, GhostScript

FIG. 4.5 – Synchronisation entre MVLets

La MVLet PostScript affiche les pages du document dans une fenêtre fille de la fenêtre de la MVLet de documents actifs. Ensuite, les autres MVLets de documents sont lancées. La MVLet texte brut insère directement le texte traduit sous forme PostScript dans GhostScript. La MVLet texte brut peut être vue comme une feuille de style pour texte brut transformant du texte en PostScript dynamiquement. Les MVLets image et vidéo sont lancées dans des fenêtres filles de la fenêtre de la MVLet pour documents actifs. Les MVLets image et son utilisent le passage d'une page à l'autre pour s'arrêter alors que la MVLet vidéo attend la fin de la vidéo pour autoriser l'utilisateur à changer de page³. Ces cinq MVLets montrent la souplesse de la synchronisation entre les différentes MVLets de documents.

La figure 4.6 présente le code complet de la MVLet image qui encapsule l'exécutable Display⁴. Display est lancé (si `stat ?` vaut `da :stat-start ?`) en fournissant à l'exécutable la fenêtre dans laquelle il doit produire la sortie. Lors du passage d'une page à une autre, l'exécutable est arrêté en envoyant un signal (si `stat ?` est différent).

¹Il suffit de vérifier que les coordonnées sont dans l'image.

²L'exécutable Gv encapsule GhostScript de la même façon, la documentation concernant les messages a été trouvée dans les sources de Gv.

³L'exécutable Mplayer peut être interrompu par l'utilisateur à l'aide de la touche d'échappement.

⁴Une seule image peut être affichée à la fois.

```

1
2 (define .img-pid -1)
3 (define .img-win 0)
4
5 (define .media/image
6   (lambda (stat? ags primaire file px py pdx pdy)
7     (if (= stat? :da.stat-start)
8       (let ([tx ( :da.ags.pix-width ags)]
9             [ty ( :da.ags.pix-height ags)]
10            [dx (/ (* tx pdx) 1000)]
11            [dy (/ (* ty pdy) 1000)]
12            [x  (/ (* tx px) 1000)]
13            [y  (/ (* ty py) 1000)])
14         (set! img-win (create-window ags x y dx dy))
15         (set! img-pid ( :system.fork))
16         (cond [(= img-pid -1) ( :system.error "fork() : %s" ( :system.strerror))]
17               [(not img-pid)
18                (let ([bwin ( :system.alloca 32)]
19                      [bxy ( :system.alloca 32)])
20                  ( :system.sprintf bwin "%p" img-win)
21                  ( :system.sprintf bxy "%dx%d" dx dy)
22                  ( :da.localize-ressource
23                    [f file]
24                    ( :system.execlp "display" "display" "-window" bwin "-size" bxy f 0))))])
25       (let ([res 0])
26         ( :system.kill img-pid ( :system.SIG_INT))
27         ( :system.waitpid img-pid (addr res) 0)
28         ( :x11.XDestroyWindow ( :da.ags.my-display ags) img-win))))))
29
30

```

FIG. 4.6 – La MVLet image

4 Deux adaptations des MVLets de documents actifs

Dans cette section, nous présentons deux MVLet d'adaptation des documents actifs. La première permet de générer des fichiers PostScript standards à partir de documents actifs PostScript et la seconde offre un outil de conception de documents actifs.

4.1 Production d'un PostScript Portable

L'exécution du PostScript dans l'environnement Machine Virtuelle Virtuelle n'est pas possible si l'utilisateur ne possède pas la μvm . La MVLet de PostScript portable (PP) permet de s'affranchir de cette limite. Cette MVLet modifie la précédente pour générer un nouveau fichier PostScript standard contenant les notes.

Il n'est, évidemment, pas raisonnable de demander des fonctionnalités multimédia aux interpréteurs standards, mais les notes texte peuvent être insérées directement dans le PostScript. La MVLet PostScript Portable change le point d'entrée d'exécution (`:da.run`, voir figure 4.3) de la MVLet de documents actifs. Cette nouvelle fonction d'exécution ne lance pas l'exécutable ghostscript mais envoie directement sur la sortie standard l'ancien PostScript enrichi du chargeur de PostScript et des notes converties en PostScript. Les fonctions de la MVLet documents actifs sont réutilisées pour effectuer ce travail. Le fichier PostScript généré contient donc les notes texte et les outils nécessaires à leur affichage avec un interpréteur PostScript Standard.

Cette MVLet montre comment la MVV de documents actifs peut être modifiée en fonction des besoins particuliers d'un utilisateur, sans avoir à modifier le code original présenté dans la section précédente.

4.2 Le compositeur de notes

Dans la section 3.2.1, la figure 4.3 présente un document actif sous la forme d'un programme. Le compositeur de notes permet de générer ce programme automatiquement à l'aide d'une interface graphique. Cette interface trouve automatiquement les coordonnées des post-it graphiquement.

Plutôt que d'écrire un programme de composition, la MVLet compositeur de notes adapte dynamiquement la MVV document actif pour lui apprendre à générer des fichiers de documents actifs : la MVLet de documents actifs est chargée dans la μvm et la MVLet compositeur de notes utilise les mécanismes mis en œuvre précédemment pour ajouter et afficher dynamiquement de nouvelles notes. Ensuite, la MVLet compositeur de notes construit un fichier de description du document actif indépendant.

La fonction de réception de message du serveur graphique est déroutée vers une nouvelle fonction qui s'occupe d'attraper les clics droits de la souris. Dès qu'un clic à lieu, la MVLet compositeur de notes attend le relâchement de la souris et calcule automatiquement la surface et la page sur laquelle doit être posée une note. Ensuite, un petit menu contextuel (écrit en TCL/TK) permet de spécifier le contenu de la note et le gestionnaire associé. La note est ensuite directement exécutée par le moteur. L'utilisateur construit donc de manière interactive son document actif.

Quand l'utilisateur appuie sur le bouton du milieu, le fichier μvm du document actif est généré à partir des informations stockées dans la liste des notes. L'entête est générée

en incluant les gestionnaires de types MIME adéquats. Ce document actif génère donc des documents actifs.

Le composeur de notes montre comment la machine virtuelle virtuelle de documents actifs peut être enrichie avec de nouvelles possibilités, tout en réutilisant le travail effectué précédemment sans le modifier.

5 Qualité de service

La gestion de qualité de service d'un document peut être déclinée sous de nombreuses formes : débit sur un réseau, ordonnancement, gestion mémoire etc... Dans le cadre de ces travaux, seule la politique d'ordonnancement d'une tâche sur une machine a été explorée. En suivant la même approche, d'autres aspects de la qualité de service pourraient être modifiés, en particulier en utilisant la notion de réseaux actifs (voir section 4.1, chapitre 2) pour gérer la qualité du transfert des documents.

Le système a pour rôle de s'occuper d'attribuer les ressources locales de la machine aux différentes tâches s'exécutant. Il existe peu de systèmes permettant à une application de modifier la gestion de ces ressources. En effet, cette notion est en contradiction avec la notion de système puisqu'une application pourrait s'attribuer toutes les ressources et empêcher la collaboration entre les différentes tâches. Toutefois, un système ne peut pas prendre en compte la nature des tâches s'exécutant dans le système. Dans un système Unix par exemple, le CPU est géré en suivant un algorithme de tourniquet (round robin), or certaines tâches ont des besoins précis en terme d'échéance, comme les lecteurs vidéo, alors que d'autres tâches n'ont pas de contraintes temporelles, comme des tâches de compilation.

Comme le système gère les tâches indépendamment de leur nature, il semble cohérent d'offrir à certaines applications la possibilité de modifier les mécanismes de gestion de ressources du système. Le système NEMESIS [RF97, LMB⁺96], par exemple, offre à une application des garanties sur l'utilisation des ressources système, comme le processeur, la mémoire ou la bande passante. En augmentant ces garanties, on diminue l'équité des applications mais les applications gagnent en qualité de service individuellement.

Notre solution pour offrir un ordonnancement adaptable au niveau applicatif repose sur Bossa [LMB02], un DSL (voir la section 4.2 du chapitre 2) permettant d'écrire des politiques d'ordonnancement dans un langage de haut niveau. La plate-forme Bossa permet aussi d'insérer de nouvelles politiques dynamiquement dans le noyau du système d'exploitation à l'aide de modules Linux. Bossa, avec les noyaux Linux 2.4.17/2.4.24 modifiés, offre la possibilité d'ajouter et de retirer de nouvelles politiques d'ordonnancement à un document actif.

5.1 Documents actifs et Bossa

La MVLet de document actifs permet à un document de gérer sa composition mais n'offre aucune garantie sur l'utilisation des ressources matérielles. Augmenter la MVV document actif avec Bossa permet au document actif de pouvoir, en plus, gérer sa politique d'ordonnancement et de s'assurer une certaine forme de qualité de service.

La figure 4.7 présente les différents éléments intervenant pour construire ce document actif :

- les politiques d'ordonnancement, écrites dans le DSL Bossa, permettent de générer des modules d'ordonnancement nécessaires au bon fonctionnement du document actif ;

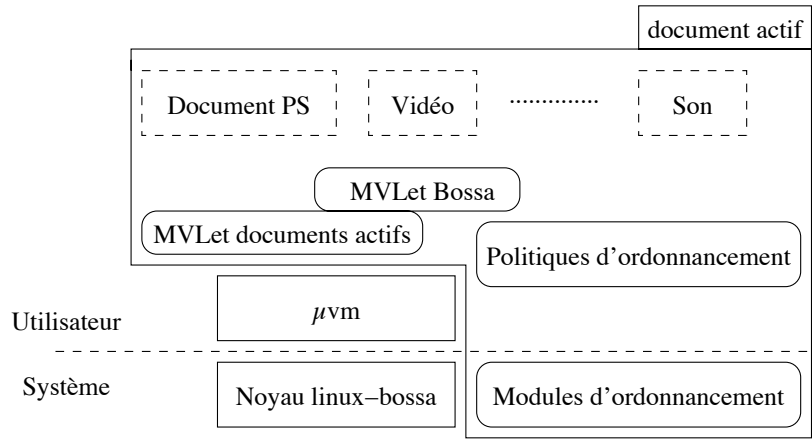


FIG. 4.7 – Architecture de l'intégration de Bossa aux documents actifs

- la MVLet Bossa fournit l'API de haut niveau pour que le document actif puisse instancier un ordonnanceur, insérer des tâches dans un ordonnanceur particulier et modifier les paramètres de cet ordonnancement ;
- la MVLet Documents actifs est la MVLet décrite dans la section précédente ;
- le document PostScript et les sous-documents sont les documents manipulés par le document actif multimédia.

L'utilisation de Bossa ne peut se faire qu'en mode super-utilisateur (pour insérer ou retirer les politiques d'ordonnancement du noyau) et avec un Linux patché avec Bossa. Ainsi, le document actif doit forcément être exécuté avec des privilèges de super utilisateur.

5.2 La MVLet Bossa

La MVLet Bossa est indépendante de la MVLet de documents actifs décrite dans la section précédente : elle peut être utilisée par n'importe quelle application active. Ces travaux ont été réalisés par C. Clement [Cle04] pendant son stage de DEA que j'ai co-encadré avec B. Folliot. La MVLet Bossa offre une interface entre une machine virtuelle virtuelle, le noyau Linux/Bossa et le compilateur Bossa.

La MVLet Bossa est composée de plusieurs parties :

- une partie générique qui permet au document actif de compiler une politique pour la plate-forme cible et d'ajouter/retirer un ordonnanceur dans le noyau ;
- de parties spécifiques à des ordonnanceurs particuliers permettant au document actif de modifier les paramètres d'ordonnancement d'un processus lié à un ordonnanceur particulier et d'attacher un processus à un ordonnanceur donné.

La partie générique de la MVLet sert à charger, décharger des ordonnanceurs et à construire des arborescences d'ordonnanceurs. Une arborescence d'ordonnanceurs repose sur un ordonnanceur virtuel qui s'occupe d'ordonnancer des ordonnanceurs. De cette façon, plusieurs ordonnanceurs différents peuvent cohabiter simultanément dans le noyau. Tous les processus Linux qui n'ont pas d'ordonnancement défini via Bossa sont dans l'ordonnanceur Linux : cet ordonnanceur doit toujours être présent dans le noyau. Si l'utilisateur veut ajouter de nouveaux ordonnanceurs, un ordonnanceur virtuel au moins est nécessaire pour

accueillir le nouvel ordonnanceur et l'ordonnanceur Linux.

Pour charger ou décharger un ordonnanceur particulier, deux fonctions sont nécessaires : `load-module` et `unload-module`. Ces fonctions chargent dans le noyau les modules d'ordonnancement grâce aux programmes `modprobe` et `rmmod`. Une fois un ordonnanceur chargé dans le noyau, il faut le connecter à l'arborescence des ordonnanceurs, ce que fait la fonction (`:bossa.connect_sched parent child`). Cette fonction attache l'ordonnanceur `child` à l'ordonnanceur virtuelle `parent`. De la même façon, un ordonnanceur peut être retiré de l'arborescence avec la fonction `:bossa.disconnect_sched`.

Les parties spécifiques sont spécialisées en fonction des ordonnanceurs utilisés. Pour une politique EDF (Earliest Deadline First), par exemple, quatre fonctions forment l'interface. La première est une fonction d'attachement à l'ordonnanceur qui prend en entrée les paramètres qui caractérisent son exécution : le pire temps d'exécution (WCET, Worst Case Execution Time) et la période requise pour le processus. La seconde est une fonction `get_missed_deadline` qui prend en paramètre un identifiant de processus et renvoie le nombre de périodes pendant lesquelles le processus n'a pas fini son exécution avant le début de la période suivante. La troisième fonction permet de connaître le nombre de fois où le processus a été interrompu parce qu'il avait consommé son temps imparti (le WCET). Enfin une fonction `sched_yield` permet au processus de quitter le processeur avant l'expiration de son WCET.

La MVLet Bossa permet donc à une application d'adapter dynamiquement sa politique d'ordonnancement en fonction de ses besoins : c'est l'application elle-même qui spécifie quelle est sa politique et comment elle doit être configurée. Cette MVLet permet à une application de modifier dynamiquement les paramètres d'ordonnancement : l'application devient donc auto-adaptative puisqu'elle n'a plus besoin de l'utilisateur pour être configurée. L'auto-adaptation évite le retard entre l'observation et la prise de décision : si un opérateur humain s'occupe de modifier l'ordonnancement de son application en fonction de ses observations, le temps qu'il modifie la politique peut suffire à rendre ces modifications inadéquates.

La MVLet Bossa suit exactement l'approche application active : c'est l'application qui construit l'environnement qui lui convient et qui l'adapte en fonction des paramètres qu'elle observe. L'observation et la prise de décision dépendent fortement de l'application.

5.3 Une vidéo auto-ordonnée dans un Document Actif

Pour évaluer la MVLet Bossa et les documents actifs, une vidéo auto-ordonnée dans un PostScript a été testée. L'ordonnancement de la vidéo est géré par la MVLet Bossa. Un nouveau fichier de gestion de type MIME est créé : `video/mpeg/bossa`. Celui-ci s'occupe de lancer une vidéo, d'observer son rendu et de prendre des décisions. La qualité de la vidéo se résume à une seule donnée : le nombre d'image par seconde (`fps`). On considère que la vidéo s'affiche correctement si ce nombre reste supérieur à une valeur seuil. Cette valeur est actuellement fixée par le document actif à 18 images par seconde, mais elle pourrait très bien être configurée dynamiquement par un utilisateur.

Une des difficultés de cet exemple réside dans l'utilisation d'un programme d'affichage de vidéo existant. La plupart de ces programmes ne donnent pas d'information sur la qualité de la vidéo. Pour obtenir le nombre d'images par seconde, nous avons utilisé un affichage passant par le framebuffer dans un serveur X. Le framebuffer permet de projeter (mapper) directement de la mémoire vidéo dans de la mémoire centrale. À chaque fois qu'une image est totalement affichée, une fonction utilisateur est appelée.

Cette fonction permet de calculer précisément le nombre d'images affichées par seconde. La première fois que cette valeur passe sous le seuil, un ordonnanceur virtuel proportionnel est créé dans le noyau (voir figure 4.8). Cette ordonnanceur attribue de manière proportionnel le CPU à plusieurs sous-ordonnanceurs. Tous les processus Linux sont placés dans un tourniquet, appelé tourniquet Linux. Comme un ordonnanceur virtuelle ne doit ordonnancer que des ordonnanceurs, un second ordonnanceur est ajouté à l'arborescence. Cet ordonnanceur, appelé ordonnanceur d'affichage, contient au maximum un seul processus : le processus d'affichage. Comme la répartition du temps est assurée par l'ordonnanceur virtuel, l'algorithme de l'ordonnanceur d'affichage n'a pas de besoin précis, c'est pourquoi nous utilisons un tourniquet.

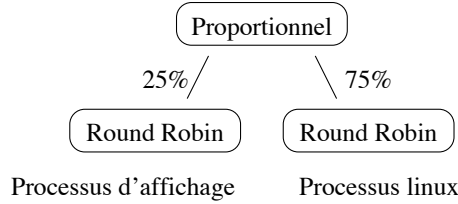


FIG. 4.8 – Ordonnancement en cas de famine

Une fois les ordonnanceurs mis en place, à chaque fois que le nombre d'images par seconde passe sous la valeur seuil, le processus d'affichage est placé dans l'ordonnaceur d'affichage. Comme notre processus tourniquet est seul dans son ordonnanceur, on est assuré qu'il aura le CPU un quart du temps. Dès que le nombre d'images par seconde repasse au dessus de la valeur seuil, le processus d'affichage est déplacé dans le tourniquet Linux.

L'algorithme d'ordonnancement utilisé reste rudimentaire : des algorithmes temps réel seraient préférables. Toutefois, cet exemple montre comment le document actif peut configurer certains paramètres système de manière à obtenir la qualité de service requise. Les problèmes de cohérence lors de la construction de l'arborescence d'ordonnanceurs ou de l'insertion de nouveaux processus dans un ordonnanceur temps réel ne sont pas abordés dans ces travaux mais pourraient être résolus en utilisant le module noyau Bossa.

6 Conclusion et perspectives

Les documents actifs permettent d'enrichir la sémantique des documents pour les transformer en documents multimédia. Pour construire des documents actifs multimédia, ni les programmes de visualisation, ni les formats des documents n'ont besoin d'être modifiés.

La figure 4.9 récapitule les différents MVLets présentés dans ce chapitre, ainsi que leurs dépendances et les exécutable liés dynamiquement avec la μvm . Ces MVLets montrent la flexibilité de l'approche : la MVLet de documents actifs a été enrichie avec d'autres MVLets spécifiques, une MVLet de production de PostScript portable, une MVLet de composition graphique de documents multimédia et une MVLet d'ordonnancement. Les documents actifs peuvent enrichir leur environnement en fonction de leurs besoins spécifiques. L'utilisation de la MVLet Bossa montre que le document actif peut non seulement agir sur son contenu et sur son programme de visualisation, mais aussi directement sur le système.

Ces MVLets montrent comment les documents actifs résolvent certains des problèmes

évoqués en introduction. Comme le document exécute du code, il n'est pas soumis aux contraintes traditionnelles des documents. Nos MVLets montrent comment résoudre les problèmes de composition de documents hétérogènes et comment gérer une forme de qualité de service. Les autres problèmes (réplication, hétérogénéité des supports de visualisation et droits d'auteur) pourraient aussi être résolus par cette approche. Par exemple, le code du document actif pourrait servir à gérer la validité du document auprès d'une copie primaire. Le document pourrait se protéger du piratage en s'auto-cryptant et en interrogeant un tiers de confiance pour obtenir une clé de décryptage. Le code exécutable d'un document actif pourrait aussi permettre au document de s'adapter à des plates-formes particulières, comme un téléphone portable, en générant son contenu grâce à une feuille de style (tout comme avec XML) [xml], mais il pourrait également s'adapter automatiquement à des utilisateurs ayant des besoins spécifiques, par exemple, à un mal voyant en changeant les polices de caractères.

L'architecture des documents actifs repose sur la notion de page et de position. Pour faire évoluer le prototype vers d'autres types de synchronisation, seule la MVLet de documents actifs présentée dans la section 3.2.1 doit être modifiée. En effet, les autres MVLets encapsulent les programmes de visualisation standards et offrent une API de haut niveau pour manipuler les exécutables. Le passage du PostScript à un autre type de document texte peut aussi se faire, à condition que le programme de visualisation du document texte se prête bien à de l'encapsulation (comme Ghostscript).

Toutefois, la construction des MVLets d'encapsulation demande de bien connaître le fonctionnement des exécutables. En particulier, la MVLet PostScript demande une connaissance fine des messages envoyés par Ghostscript. Si le programme de visualisation standard offre une documentation rudimentaire, l'encapsulation peut s'avérer impossible sans avoir accès aux sources. Par exemple, pour encapsuler des documents Microsoft Word, il faudrait savoir quelles sont les possibilités du programme MSWord, ce qui n'est pas possible actuellement.

CHAPITRE 5

La JnJVM : construction d'une machine virtuelle Java adaptable

Sommaire

1	Introduction	81
2	Conception de la JnJVM	83
3	Réalisation de la JnJVM	85
4	Performances de la JnJVM	93
5	Conclusion	99

1 Introduction

La prolifération d'environnements dédiés à des besoins applicatifs non couverts par les spécifications touche particulièrement le monde Java. Java masque l'hétérogénéité matérielle et possède de nombreuses bibliothèques, c'est pourquoi Java est devenu un standard dans les applications réparties, les intergiciels ou les services web. Avec l'émergence de besoins spécifiques (aspects, réflexion, migration, minimalité, temps réel), les machines virtuelles Java deviennent inadéquates. De nombreuses machines virtuelles Java dédiées, décrites dans la section 3 du chapitre 2 et dans la section 5 du chapitre 6, sont apparues depuis 95 : MetaXa [Gol98] ou Guaraná [OB99] pour des JVM réflexives, Steamloom [BHMO04] pour une JVM à aspects, Pjama et la machine virtuelle OPJ [JA99], pour de la persistance, les JVM qui suivent la spécification RT-Java [BBD⁺00] pour du temps réel, les JVM suivant la spécification CLDC (Connected, Limited Device Configuration) [Sun03] pour de l'embarqué (comme la KVM), MOBA [SM01] pour de la migration de fil d'exécution transparente, etc... Les applications Java écrites pour ces environnements spécialisés ne peuvent pas être exécutées sur d'autres machines virtuelles Java. Comme Java a d'abord été conçu pour masquer l'hétérogénéité des plate-formes d'exécution, on se retrouve face à une contradiction.

Plutôt que d'avoir une multitude de machines virtuelles Java incompatibles entre elles, nous proposons de construire une unique JVM générique basée sur le protocole méta-objet de la μvm et donc spécialisable par une application active. Nous permettons ainsi à l'application de construire la machine virtuelle Java adéquate pour son exécution.

Cette approche réconcilie les JVM dédiées avec les surcouches applicatives (voir la section 3 du chapitre 2). L'utilisation des applications actives dans les machines virtuelles Java offre à la fois :

- De bonnes performances : c'est la machine virtuelle Java elle-même qui prend en charge les mécanismes dédiés. Les indirections dues au surcouches applicatives sont ainsi évitées.
- Une diminution du temps d'intégration de fonctionnalités dédiées à des domaines applicatifs particuliers : le temps entre l'arrivée d'un besoin dédié et l'intégration dans les JVM standards est fortement diminué puisque c'est l'application qui implante ce mécanisme.
- Une homogénéité entre les environnements Java dédiés : les MVlets, et l'application Java sont portables. Le foisonnement d'environnements dédiés et incompatibles entre eux est ainsi évité.
- Une machine virtuelle Java générique : elle prend en entrée des programmes Java standards sous forme binaire (fichier de classe) et les exécute en suivant la sémantique de n'importe quelle machine virtuelle. L'application active spécialise ensuite le comportement de certains aspects de la machine virtuelle Java en fonction de ces besoins.

La MVLet Java développée pendant cette thèse, appelée JnJVM [TOG⁺05, OTF05, TFO03, OTFP03], est une implantation des spécifications 1.2 de la machine virtuelle Java [LY] permettant à d'autres MVlets de modifier dynamiquement le coeur de la machine virtuelle à l'aide des mécanismes de réflexion de la μvm . Dans la suite du chapitre, par souci de clarté, nous identifions la JnJVM (la MVLet) avec la Machine Virtuelle Virtuelle ainsi construite (la μvm et la JnJVM chargée).

De manière à s'intégrer avec l'existant et à suivre l'évolution des spécifications du langage Java (différentes des spécifications de la machine virtuelle Java), la JnJVM peut utiliser la bibliothèque de classes du projet GNU Classpath [gnu] qui implante actuellement les spécifications 1.2 à 1.4 Java. Le code de la JnJVM est indépendant de cette bibliothèque de classes de base. Une MVLet GNU-Classpath s'occupe d'implanter l'interface entre la bibliothèque GNU Classpath et la JnJVM : une application active peut donc redéfinir quelle bibliothèque de classes de base elle va utiliser.

L'architecture des applications actives Java est présentée sur la figure 5.1 : l'application active est constituée d'une application Java standard, de la JnJVM, de la MVLet GNU Classpath et de la bibliothèque GNU Classpath. Une fois ces quatre éléments chargés dans la μvm , le comportement de l'application active est exactement celui de l'application Java standard chargée dans toute autre machine virtuelle Java. Une application active spécialisée va alors charger ces quatre éléments et une MVLet de spécialisation qui s'occupe de modifier le comportement de la JnJVM pour qu'elle offre le mécanisme dédié requis. Par exemple, une MVLet de tissage d'aspects va contenir le code nécessaire au tissage d'aspects dans la JnJVM et le langage permettant d'exploiter ce code.

Le travail effectué pour construire la JnJVM est reproductible : tout environnement d'exécution peut être développé à partir de la μvm , par exemple une machine virtuelle C#, SmallTalk ou PostScript. Toutefois, ce travail peut s'avérer long, la JnJVM est une machine virtuelle Java complète et demande autant de travail d'ingénierie que le développement de n'importe quelle autre machine virtuelle Java.

Les différents algorithmes utilisés pour implanter la JnJVM sont des algorithmes connus : la principale originalité du travail réside dans l'utilisation du MOP de la μvm (voir la

section 3.2 du chapitre 3) pour construire la JnJVM. Les différents éléments constituant la JnJVM sont structurés sous la forme de composants réflexifs hérités de la μvm . Ces composants réflexifs rendent la JnJVM flexible et permettent à des applications actives de modifier en profondeur le comportement de la machine virtuelle Java.

Ce chapitre présente la JnJVM et ses performances. Le chapitre 6 présente une évaluation qualitative de la JnJVM et différents travaux utilisant la JnJVM pour résoudre des problèmes récurrents dans le monde Java : modification de l'allocateur mémoire, modification de la gestion des synchronisations, tissage d'aspects et migration de fil d'exécution.

La section 2 présente les concepts utilisés dans la JnJVM. La section 3 présente la réalisation de la JnJVM et décrits plus en détails quelques algorithmes utiles pour comprendre comment exploiter la flexibilité de la JnJVM. Les performances de la JnJVM sont étudiées dans la section 4 et la section 5 conclut ce chapitre.

2 Conception de la JnJVM

La JnJVM est une MVLet, elle est chargée, compilée et exécutée par la μvm (voir figure 5.1). Une fois la JnJVM chargée et compilée dans la μvm , la MVV devient une machine virtuelle Java implantant la spécification 1.2 de Sun [LY]. La MVLet GNU Classpath peut ensuite être chargée. Elle s'occupe d'implanter l'interface entre les classes dépendantes de la machine virtuelle et la JnJVM. La JnJVM peut alors charger un fichier de classe Java standard et l'exécuter. Le canal d'entrée/sortie de la μvm est ensuite utilisé pour modifier le fonctionnement de la JnJVM à l'exécution.

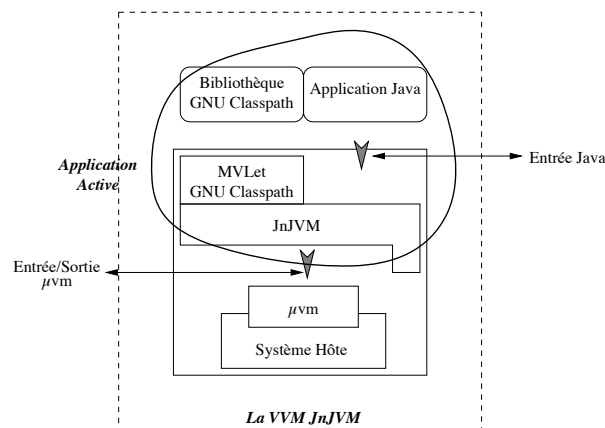


FIG. 5.1 – La machine virtuelle virtuelle Java

Les mécanismes d'adaptation de la JnJVM reposent principalement sur le MOP et sur la compilation en ligne de la μvm (voir chapitre 3). Le MOP est utilisé par une MVLet pour modifier dynamiquement l'état de la μvm alors que la compilation en ligne est utilisée par une MVLet pour injecter du code dynamiquement.

La JnJVM est structurée sous forme de composants. Le modèle de composants utilisé suit celui défini dans la μvm pour quatre raisons principales :

- Pour réutiliser le travail effectué lors de la conception de la μvm .

- Pour bénéficier de la réflexivité de la μvm et pour pouvoir adapter le comportement ou les données de la JnJVM à l'exécution.
- Pour donner à l'utilisateur une structure uniforme de la mémoire. Toutes les fonctions applicables aux objets de la μvm sont applicables aux composants de la JnJVM.
- Pour éviter d'avoir à définir de nouvelles racines de collection de la mémoire. Il suffit de stocker un composant JnJVM dans un composant collectable de la μvm pour qu'il soit automatiquement atteignable.

La vision uniforme de la mémoire de la JnJVM, de la μvm et de l'application permet au concepteur d'application et au concepteur de MVLets d'avoir un environnement de développement unique à appréhender et des mécanismes communs mettant en œuvre l'adaptabilité dynamique.

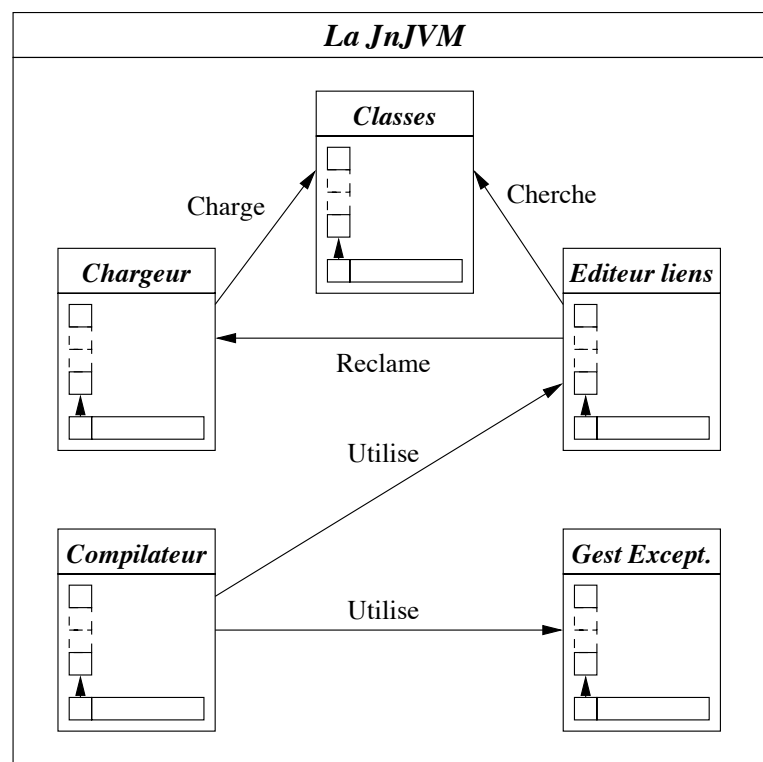


FIG. 5.2 – Les composants de la JnJVM

La figure 5.2 récapitule les principaux composants constituant la JnJVM. Le chargeur de classes charge de nouveaux fichiers de classes Java, l'éditeur de lien s'occupe d'effectuer les liaisons (au plus tard) entre les symboles Java et les entités représentées par ces symboles. Le gestionnaire d'exception prend en charge les exceptions. Le compilateur génère du code assemblé à partir du bytecode Java : c'est un compilateur Just In Time (JIT). Les classes de la figure sont la représentation interne des classes Java chargées dans la JnJVM.

3 Réalisation de la JnJVM

Dans cette section, la réalisation de la JnJVM est décrite. Fondamentalement, la JnJVM est une machine virtuelle Java comme une autre. Les particularités de notre plate-forme résident dans l'utilisation de la structure composant de la μvm qui offre aux applications actives des outils pour modifier dynamiquement le cœur de leur environnement d'exécution.

3.1 Composants JnJVM

Le composant de base de la JnJVM est la machine virtuelle Java. Ce composant contient les principales abstractions de manipulation de la JVM : chargement, compilation, édition de lien, gestion d'exceptions, descriptions de classes, de champs, de méthodes et de signatures (voir figure 5.3). Les différents éléments de la figure 5.2 se retrouvent dans la figure 5.3 et sont présentés plus en détail. Plusieurs instances de machines virtuelles peuvent coexister simultanément pour permettre à des MVLet de faire de l'isolation ou pour maintenir plusieurs versions d'une même classe à l'exécution (un exemple d'utilisation de deux JVM est donné dans la section 4.2 du chapitre 6).

Les composants interagissent entre eux via leurs interfaces : une MVLet est libre de remplacer l'implantation d'un composant par un autre, en particulier la machine virtuelle Java elle-même. Les classes, les champs et les méthodes Java sont aussi représentés par des composants. Les attributs de ces composants correspondent aux fichiers de classes Java. Ils sont enrichis avec les données valables pendant l'exécution, comme l'offset d'un champ dans un objet.

Le constant pool est un tableau contenu dans le fichier binaire Java. Il fait la correspondance entre un nombre et un élément du fichier de classe, comme un champ, une méthode, une signature, une chaîne de caractères ou une classe. Le constant pool en mémoire est aussi structuré sous la forme d'un composant μvm .

Les instances de classes Java sont elles aussi structurées sous forme de composants, toujours pour les mêmes raisons : les objets applicatifs, les composants internes de la JnJVM et les composants internes de la μvm possèdent le même format. Les méta-données des instances de classes sont simplement les descriptions de classes Java (voir la section 3.2 du chapitre 3).

Une MVLet peut ajouter de nouvelles méthodes à des classes chargées ou en supprimer d'anciennes en modifiant la structure de classe. En revanche, l'ajout dynamique de champs dans les classes déjà chargées ayant déjà des instances reste difficile¹. Il faut que la MVLet trouve tous les composants en mémoire, vérifie s'ils sont des instances de classes Java et s'ils héritent de la classe modifiée. Ensuite, la MVLet doit séparer ces composants en deux et les ré-allouer pour insérer le nouveau champ. Les références et les pointeurs sont identifiés, donc réallouer un composant revient à modifier sa référence. Il faut donc, pour chaque instance réallouée, parcourir l'ensemble des instances Java pour chercher toutes les instances qui référencent l'instance réallouée. Ce travail est toutefois réalisable (on possède une description des champs Java) mais pose aussi des problèmes de synchronisation : comme les instances de classes et la description de classes ne correspondent plus, la liaison ne peut plus être effectuée pendant l'adaptation².

¹Ajouter de nouveaux champs pendant le chargement ne pose aucun problème.

²Un sémaphore permet d'avoir un accès exclusif pendant la liaison.

En utilisant la structure sous forme de composant, les descriptions de classes, de champs et de méthodes sont collectées régulièrement. Une MVLet qui modifie le contenu de ces champs (comme la MVLet de tissage d'aspects) n'a donc pas à préoccuper de la gestion mémoire.

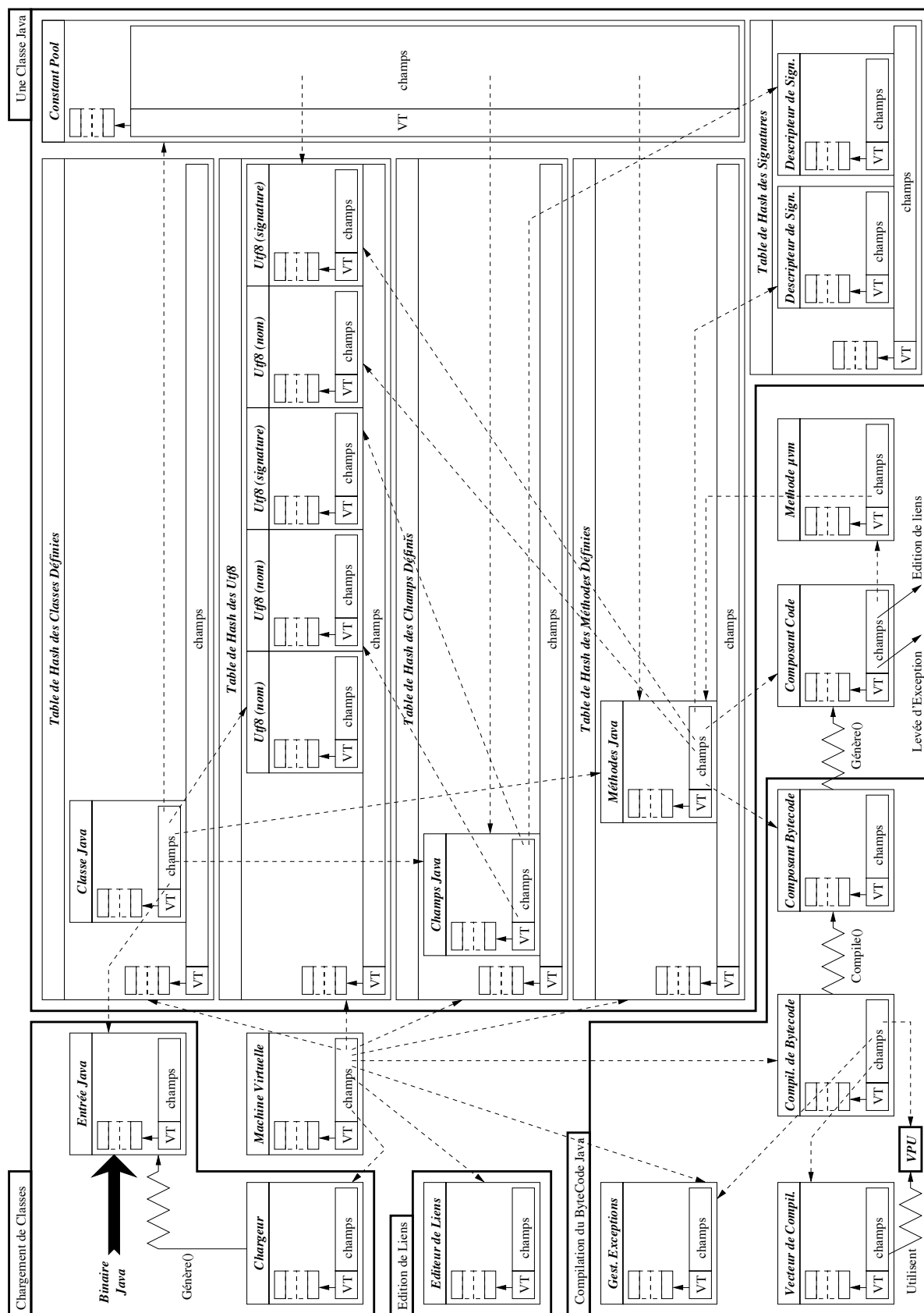


FIG. 5.3 – Réalisation de la JnJVM

Le bytecode Java est aussi compilé dans des composants de la μvm (voir la section 3.2 du chapitre 3 et la figure 5.3) : les méthodes sont collectées par le ramasse-miettes et sont manipulées comme tout autre composant du système. De plus, l'introspection de la pile est facilitée par l'utilisation de composants : à l'aide de l'allocateur du ramasse-miettes, un pointeur vers une fonction Java assemblée en mémoire peut être déréférencé vers la description de méthode Java.

3.2 Liaison

La liaison s'effectue au plus tard : les classes ne sont chargées et liées que si elles sont utilisées pendant l'exécution. Neuf types de liaison existent : les liaisons de champs statiques et d'instance, en lecture ou en écriture, quatres liaisons de méthode et la liaison de l'opération d'allocation d'une classe en mémoire.

Les quatres liaisons de méthode sont :

- statique : pour lier un appel à une méthode de classe ;
- virtuelle : pour lier un appel à une méthode d'instance en suivant l'algorithme classique Java ;
- spéciale : pour lier un appel à une méthode d'instance en suivant l'algorithme de **super** en Java ;
- interface : pour lier un appel à une méthode d'interface.

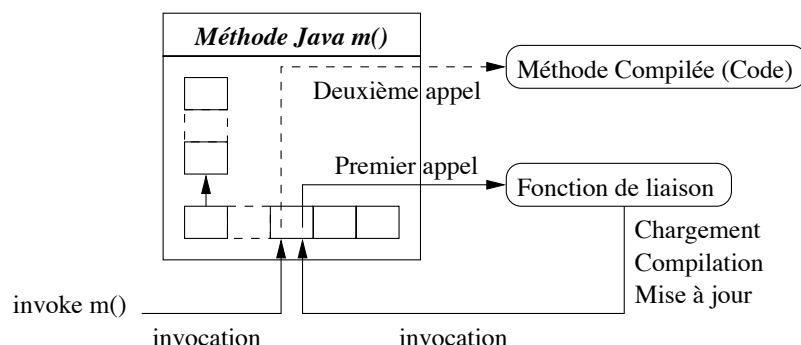


FIG. 5.4 – Liaison statique de méthode

Les liaisons statiques (voir figure 5.4) et spéciales sont proches l'une de l'autre : les appels dépendent de la classe passée en paramètre. Pour effectuer la liaison au plus tard, le premier paramètre d'une méthode statique est la description de méthode et le premier paramètre d'une méthode spéciale est un cache en ligne (voir ci-après).

Les liaisons de méthode virtuelle et d'interface sont plus complexes. La figure 5.5 présente un exemple de liaison virtuelle. La méthode `Base.test()` appelle la méthode `f()` de l'objet passé en paramètre. Cet appel à `f()` dépend de la classe de l'objet passé en paramètre (`Base`, `X` ou `Y`). A chaque exécution d'un appel de méthode virtuel ou d'interface, il faut lier la méthode en fonction de l'objet cible.

Plutôt que de recommencer la liaison à chaque exécution, un système de cache en ligne est utilisé (voir figure 5.6). La liaison n'est effectuée que si la cible change. Le cache d'appel

```

1 class Base {
2   public void f() { System.out.println("Base.f()"); }
3   public static void test(Base b) { b.f(); }
4 }
5 class X { public void f() { System.out.println("X.f()"); } }
6 class Y { public void f() { System.out.println("Y.f()"); } }

```

FIG. 5.5 – Appel de méthode virtuelle

se sépare en deux parties : une partie stockant les méta-données de l'appel (la méthode à appeler) et une liste chaînée de doublets (dernière cible, pointeur de code associé). L'utilisation d'une liste chaînée répond à deux difficultés :

- Lorsque la cible courante du cache change, un seul pointeur doit être mis à jour (noté **First** sur la figure). Les caches sont donc utilisables en environnement multi-tâches³.
- Chaque liaison est conservée dans la liste chaînée. Comme le nombre de classes cibles pour un site d'appel est relativement faible, les reliaisons sont accélérées.

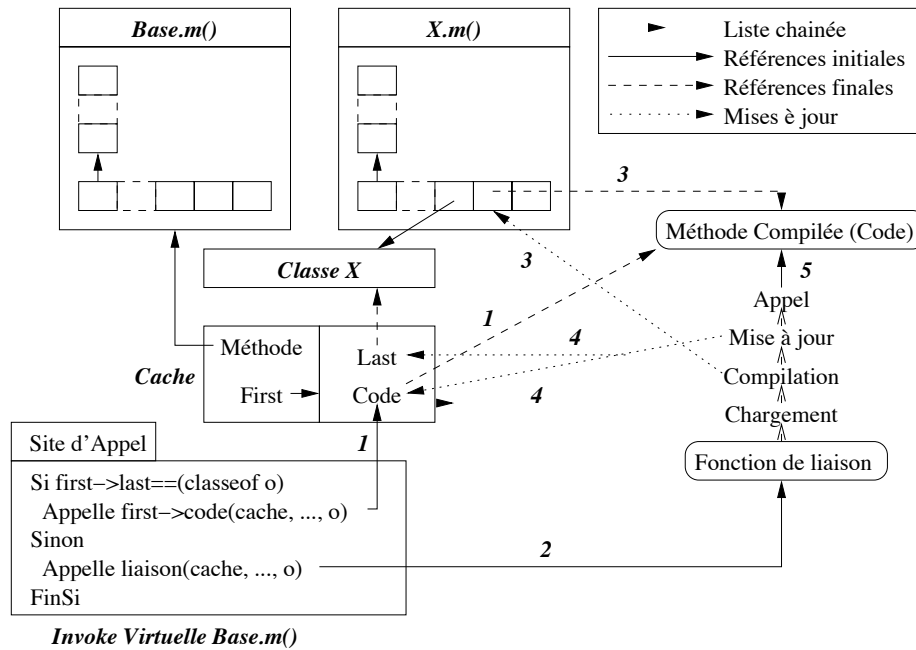


FIG. 5.6 – Liaison virtuelle de méthode

Dans la méthode appelante, la classe de l'objet cible est comparée à la classe de la dernière cible (1 sur la figure). Si ces deux classes sont égales, le pointeur de code du cache est directement appelé (tracé en pointillé vers "Méthode Compilée (Code)" sur la figure 5.6). Sinon, une fonction de liaison est appelée (2 sur la figure). Celle-ci s'occupe de charger la classe et de compiler la méthode cible (3) (si ce n'était pas déjà fait), de mettre à jour le

³Sans cette liste, il faut mettre à jour deux données : la dernière cible et le pointeur de code, ce qui n'est pas utilisable dans un environnement multi-tâches.

cache en ligne (4) et d'appeler le code assemblé de la méthode (5). Cet algorithme n'est efficace que si la plupart des appels sont non polymorphiques⁴ ce qui est le cas de Java. Le premier argument des méthodes virtuelles, spéciales et d'interface est ce cache en ligne.

Dans le meilleur des cas, un appel virtuel ou d'interface passe par la lecture du premier cache, la lecture du type de l'objet cible, la lecture de la dernière cible, la lecture du pointeur vers le bytecode compilé, un appel indirect et un saut (quatre lectures, un appel indirect et un saut). Un algorithme utilisant les tables virtuelles des composants serait plus efficace pour les appels virtuels (pas de vérification de l'objet cible) mais empêcherait une MVLet d'ajouter ou d'enlever des méthodes dynamiquement à une classe (voir la section 3.2.1 du chapitre 3). L'utilisation de tables virtuelles a aussi été partiellement implanté et augmente les performances des appels de 5% sur le test d'appels de méthodes du JavaGrande (voir section 4.5). En outre, le MOP de la μvm n'implante que l'héritage simple d'interface : l'utilisation des tables virtuelles de la μvm n'est donc pas adéquate pour implanter l'héritage multiples d'interfaces.

Les autres types de liaisons ne présentent pas un grand intérêt : elles sont effectuées au plus tard (c'est à dire lors de l'exécution) et ne sont pas décrites dans ce document.

3.3 Compilation

Chaque bytecode de méthode Java est compilé en appelant une fonction dans le vecteur de fonctions du composant de compilation. Une MVLet peut donc modifier la manière dont un bytecode particulier est compilé en changeant le pointeur dans ce vecteur. Un exemple utilisant cette possibilité est présenté dans la section 3 du chapitre 6 sur l'analyse d'échappement.

La compilation de bytecode Java utilise directement la VPU : le compilateur en ligne écrit dans la MVLet est donc portable sur n'importe quelle plate-forme possédant une μvm .

Le bytecode Java est exécuté dans une machine pile, par exemple la suite de bytecodes Java suivante :

```

1   iload 10           ; pile : <10>
2   iload 12           ; pile : <10 12>
3   isub              ; pile : <-2>

```

charge les entiers 10 puis 12 sur la pile et effectue une soustraction. Comme la VPU est aussi une machine à pile, traduire cette séquence revient à appeler les fonctions suivantes :

```

1 ( :compiler.vpu.ld-int vpu 10) ;; pile : <10>
2 ( :compiler.vpu.ld-int vpu 12) ;; pile : <10 12>
3 ( :compiler.vpu.sub vpu) ;; pile : <2>
4 ( :compiler.vpu.neg vpu) ;; pile : <-2>

```

Comme présenté sur cet exemple, l'ordre des paramètres des opérations sur la pile est inversé entre la VPU et le bytecode Java. Pour tout ce qui est calcul arithmétique, cette propriété ne pose aucune difficulté. En revanche, pour les appels de méthodes, cette propriété entraîne une inversion des paramètres des méthodes (en particulier **this** est le dernier

⁴La cible de l'appel reste relativement constante au court du temps.

paramètre) car inverser les éléments de la pile de la VPU peut être coûteux (allocation de registres, passage en pile etc...).

La compilation de la plupart des bytecodes revient simplement à traduire une instruction bytecodée Java en instruction pour la VPU. La compilation des bytecodes entraînant une liaison est plus compliquée car la VPU suit une sémantique d'appel C alors que la machine virtuelle Java suit les sémantiques présentées en section 3.2 avec liaison et chargement de classes au plus tard.

Comme les flottants, les doubles et les entiers sur 8 octets n'existent pas dans la VPU, ces calculs sont effectués en appelant des fonctions externes d'une bibliothèque écrite en C et liée dynamiquement à la JnJVM. Les fonctions de cette bibliothèque émulent les opérations avec ces types de données. Ces fonctions reçoivent des entiers, ils sont ensuite transformés dans le type de données voulues et mis dans les registres adéquates. L'opération est appelée et le résultat est transformé de nouveau en entiers. Cette transformation n'est pas un cast, le nombre entier reçu est directement la valeur binaire, par exemple, le flottant 1.0 a une valeur entière de `0x3f800000`. L'utilisation de ces fonctions diminue très nettement les performances globales de la JnJVM : les calculs flottants sont soixante-dix fois plus lents avec la JnJVM qu'avec la machine virtuelle d'IBM (voir section 4) alors que les calculs entiers ne sont que trois fois plus lents. Ces mauvaises performances sont dues à un manque de travail d'ingénierie qui n'a pas pu être effectué faute de temps.

Le compilateur de bytecode pourrait encore être amélioré en utilisant des techniques d'inlining : si une sous méthode est souvent appelée avec la même cible, il pourrait être rentable de compiler en mémoire un code spécialisé pour cette cible en mettant directement la sous méthode dans la méthode appelante. Cette technique est exploitée par la machine virtuelle d'IBM par exemple.

3.4 Exception et gestion d'erreurs

Les exceptions sont gérées en utilisant les fonctions de la bibliothèque C standard (glibc) `setjmp` et `longjmp`. Cette technique n'est pas optimale, mais la VPU ne fournit pas encore les abstractions nécessaires à cette gestion en interne. Pour rendre le code de la JnJVM portable, nous avons adopté cette approche lente.

En modifiant le composant de gestion d'exception, une MVLet peut attraper les appels aux fonctions `try`, `catch` et `throw`. Ces points d'adaptation peuvent être utilisés par effectuer du tissage d'aspects applicatifs (voir la section 3 du chapitre 6).

3.5 Les langages dédiés à l'adaptation

Les applications actives interagissent directement avec le cœur de la JnJVM. Il faut donc leur proposer des abstractions de haut niveau pour qu'elles puissent accéder aux différents éléments constituant la JnJVM et les applications Java (voir figure 5.3).

Pour modifier le comportement de la JnJVM elle-même, les mots clés générés par le MOP suffisent : ils permettent de remplacer des fonctions des tables virtuelles par de nouvelles fonctions et de modifier des champs des composants. Pour les vecteurs de fonctions, comme le compilateur de bytecode, le langage de la μvm a été enrichi pour permettre à une MVLet de modifier des entrées de ces vecteurs à l'aide de noms symboliques. Par exemple, `(rdef-opcode vm name body)` remplace la fonction de compilation du bytecode de nom `name` dans la machine virtuelle Java `vm` par la fonction `body`.

Une application active doit pouvoir connaître et modifier l'état d'une application pendant qu'elle s'exécute pour l'adapter. Un grand effort a donc été effectué pour permettre à une MVLet d'interagir simplement avec l'application. Le langage de la JnJVM a été enrichi avec de nouveaux mots clés permettant d'appeler des méthodes Java, consulter ou modifier des champs, trouver des définitions de classes, de champs, de méthodes, lever ou attraper des exceptions. Ces mots clés sont utilisés en interne par la JnJVM et en externe par les applications actives.

La grande différence entre ce langage dédié et les fonctions de la Java Native Interface⁵ [Lia99] (ou les classes de réflexion Java) est le nombre de niveaux d'appels. En effet, les nouveaux mots clés insèrent directement dans le code assemblé les appels, exactement comme le fait le composant de compilation de la JnJVM. Par exemple,

```

1 (define .f
2   (lambda (blackbox o)
3     ( :system.printf " ; j'appelle %s sur %s\n"
4       ( :object.print-string meth)
5       ( :object.print-string o))
6     ( :jnjvm.invoke.virtual meth 18 o))) ;;; appelle meth avec this=o et l'argument 18
7   ;;; equivalent \a "return o.meth(18);" en java

```

produit le même code que l'équivalent en Java noté en commentaire sur la dernière ligne.

Dans l'autre sens, une méthode Java assemblée en mémoire l'est par la VPU et suit une sémantique d'appel C. Aux paramètres près (présence d'un paramètre **cache** ou **methode** permettant la liaison au plus tard et l'ordre des paramètres des méthodes), des fonctions écrites en μvm peuvent être utilisées à la place des méthodes compilées Java. Un exemple très simple de tissage d'aspects est donné par :

```

1 (let ([M ( :jnjvm.method.lookup ;; cherche la méthode
2       vm ;; dans la jvm vm
3       ( :jnjvm.class.lookup vm ( :jnjvm.utf8 vm "Titi")) ;; dans la classe Titi
4       ( :jnjvm.utf8 vm "bip") ;; ayant le nom bip,
5       ( :jnjvm.utf8 vm "()I") ;; la signature int ...();
6       :jnjvm.access.ACC_VIRTUAL)]) ;; et l'access virtuel
7   ( :jnjvm.method.set-pointer M f)) ;; remplace le pointeur de int Titi : bip()

```

A chaque fois que la méthode **Titi.bip()** sera appelée dans une méthode Java, c'est en fait la méthode **f** qui recevra l'appel. Tel quel, l'exemple n'est pas fonctionnel car les caches en ligne d'appels ne sont pas mis à jour. Un exemple complet de MVLet de tissage d'aspects est donné dans la section 4 du chapitre 6.

3.6 Symboles Java, symboles μvm

Une option de la JnJVM crée automatiquement un équivalent d'un symbole Java dans l'espace de nom de la μvm : une MVLet écrite dans le langage de la μvm peut donc accéder aux symboles de l'application Java de la même façon qu'elle accède aux symboles de la μvm . Cette option peut être utilisée pour connaître facilement l'état des classes chargées, mais aussi pour de la réflexion : les méta-données des symboles Java sont stockées dans les

⁵JNI.

symboles μvm . Ces symboles traduits tiennent un rôle équivalent à celui des symboles des listes des méta-interfaces et des méta-composants du MOP interne de la μvm décrit dans la section 3.2.2 du chapitre 3.

Par exemple, la méthode Java `bip()` de la classe `Titi` est accessible en utilisant le symbole μvm : `jnjvm.classes.Titi.bip_<>I`. Ce symbole sert à appeler la méthode à partir d'une MVLet (en mettant en ligne l'appel), mais aussi à retrouver la définition de la méthode (stockée dans le champ objet du symbole).

Cette possibilité fait double emploi avec le langage dédié décrit dans la section précédente. En effet, il est toujours possible d'accéder, d'appeler, de consulter ou de modifier des méthodes ou des champs en utilisant le langage interne de la JnJVM décrit dans la section précédente. La mémoire dépensée est de 16 octets par symboles Java traduit : la perte de mémoire peut donc être assez lourde. L'utilisation de cette traduction n'est pas obligatoire, c'est pourquoi un drapeau de la JnJVM indique quand traduire les symboles Java en symboles μvm .

4 Performances de la JnJVM

L'adaptation est souvent utilisée pour augmenter les performances d'une application (voir chapitre 2). Comme les applications actives ont été conçues pour permettre à l'application d'adapter son environnement en fonction de ses besoins, il faut que cette approche offre de bonnes performances. Dans cette section, nous évaluons les performances de la JnJVM et de la μvm , en terme de vitesse d'exécution et de mémoire utilisée, et plus globalement, l'impact de l'architecture des applications actives sur ces performances.

Toutes les mesures présentées dans cette section ont été effectuées sur un Powerpc avec un Linux (Gentoo) cadencé à 1Ghz. Nous avons comparé la JnJVM à deux autres machines virtuelles Java :

- la JVM d'IBM (version 1.4.1) avec compilateur en ligne⁶ ;
- la JVM Kaffe (version 1.1.4) sans compilateur en ligne.

Pour les tests, la JnJVM utilise les classes de démarrage GNU ClassPath [gnu] (version 0.10). Ce paquetage implante les spécifications 1.2 à 1.4 [GJSB00] de Sun et est loin d'être terminé et optimisé. Le programme de test utilisé est le bench standard développé par le JavaGrande Forum [EPP03]. Il permet à la fois de comparer la JnJVM avec la JVM d'IBM et la JVM Kaffe, mais ce bench donne aussi une note qui classe la JnJVM par rapport à l'ensemble des machines virtuelles Java connues.

Les tests n'ont pas pu être effectués sur Pentium à cause du manque de maturité de la VPU pour cette plate-forme, c'est pourquoi nous n'avons pas pu comparer la JnJVM avec d'autres JVM avec JIT. Les résultats présentés dans [jit02] montrent que la machine virtuelle d'IBM est actuellement l'une des plus rapides du marché avec des performances équivalentes à la machine virtuelle actuelle de Sun (HotSpot).

La suite de tests JavaGrande donne une note globale mesurant les performances de la JnJVM par rapport aux autres machines virtuelles Java. Cette note est de 0.48, ce qui place la JnJVM au même rang que la première version de la machine virtuelle Java à compilateur JIT de Sun Microsystem (HotSpot 1.0). La version 1.2.0 de la machine virtuelle d'IBM (précédente de celle des tests) obtient une note de 3.28. Ce résultat global est plutôt positif compte tenu de la maturité de la JnJVM.

⁶Une des seules JVM pour PowerPc sous Linux avec compilateur JIT.

4.1 Démarrage

Comme la JnJVM est compilée lorsqu'elle est chargée dans la μvm , le temps de démarrage de la JnJVM et la place qu'elle utilise en mémoire sont deux données qui sont différentes des JVM standards.

4.1.1 Vitesse de chargement

Le démarrage de la JnJVM est beaucoup plus long que le démarrage d'autres machines virtuelles. En effet la JnJVM est entièrement compilée en mémoire par la μvm : la JnJVM fait environ 15000 lignes de code, soit 2798 fonctions assemblées en mémoire. Pour mesurer le temps de compilation de la JnJVM, nous exécutons un programme `HelloWorld` qui ne fait qu'afficher un message. La figure 5.7 donne les temps de démarrage des trois machines virtuelles Java étudiées.

	JnJVM	IBM	Kaffe
Chargement du binaire	0.24s	$\sim 0.24s$	$\sim 0.24s$
Compilation de la JVM	1.67s	0s	0s
Exécution de <code>HelloWorld</code>	0.21s	$\sim 0.47s$	$\sim 0.03s$
Total	2.12s	0.71s	0.27s

FIG. 5.7 – Exécution de `HelloWorld`

La machine virtuelle d'IBM compile les méthodes de démarrage et elles ne sont appelées qu'une seule fois : le temps de compilation de ces méthodes ne peut donc pas être amorti par le temps d'exécution, ce qui explique la différence entre la JVM d'IBM et la JVM Kaffe. L'exécution de `HelloWorld` dans la JnJVM se sépare en 3 parties : lancement de la μvm (0.24s, équivalent à la JVM d'IBM ou la JVM Kaffe), compilation de la JnJVM (1.67s) et exécution de `HelloWorld` (0.21s, équivalent à la JVM d'IBM). La JnJVM perd donc 1.67s au démarrage, soit un démarrage 6 fois plus lent que la machine virtuelle d'IBM. Ce résultat est plutôt positif car il montre qu'on peut compiler la MVLet pour un coût qui reste raisonnable vis-à-vis de l'utilisateur final. De plus, cette expérience montre que la μvm compile les fonctions relativement rapidement : en moyenne, la μvm compile une fonction en 597 μs .

La μvm n'est pas encore capable de générer des fichiers exécutables et il n'est donc pas possible de démarrer la JnJVM sous une forme pré-compilée. Une modification de la VPU pour générer des exécutables à partir des fonctions compilées en mémoire est actuellement en projet.

4.1.2 Mémoire de la JnJVM sans application

Comme la μvm utilise de la mémoire pour compiler les fonctions et garde un certain nombre de données à chaque compilation (méta-données, symboles), il est intéressant de savoir quelle est la perte due à la compilation de la JnJVM en mémoire. L'expérience menée consiste à exécuter une boucle infinie, juste après un appel à la méthode Java `System.gc()` qui collecte la mémoire.

La figure 5.8 montre les quantités de mémoire utilisées pendant l'exécution des tests. La taille mémoire physique utilisée par la JnJVM est supérieure aux autres machines virtuelles,

	JnJVM	IBM	Kaffe
Mémoire Virtuelle (en Mo)	37.57	429.21	14.23
Mémoire Physique (en Mo)	15.04	11.21	4.24

FIG. 5.8 – Utilisation de la mémoire

ce qui s'explique par la présence des méta-données générées lors de la compilation. En effet, 44555 composants résident en mémoire, soit 2.25 Mo (les classes, champs, méthodes et objets Java sont compris dans ces nombres). Ces composants n'existent que pour assurer la réflexion nécessaire à l'adaptation de la JnJVM par des applications actives.

Garder les méta-données des fonctions compilées n'est pas gratuit et peut être problématique pour des portages sur des machines ayant peu de ressources mémoire, comme les PDA (Personal Digital Assistant). Toutefois, cette perte mémoire est négligeable devant la puissance des machines de bureau actuelle et n'entraîne pas de pertes de performances avec un ramasse-miettes générationnel (voir section 4.4).

4.2 Calculs entiers

Les performances du calcul entier évaluent les performances bruts de la VPU. Le test d'addition effectue uniquement des additions avec des variables locales, donc ce sont principalement les performances de l'allocation de registre lors de la compilation qui sont mesurées. Les résultats obtenus montrent que la JnJVM est 7.6 fois plus lente que la JVM d'IBM et 23.3 fois plus rapide que la machine virtuelle Kaffe.

La JnJVM est bien une machine virtuelle à JIT, nettement plus rapide qu'une JVM sans JIT comme Kaffe. En revanche, les performances sont moins bonnes que la machine virtuelle d'IBM. Cette différence s'explique principalement par la portée des variables locales. La VPU alloue définitivement une variable locale soit dans un registre, soit en pile, alors que la JVM d'IBM peut migrer les variables de la pile vers un registre en fonction de l'utilisation. Comme la fonction de test du JavaGrande utilise déjà 30 variables locales, elles sont pratiquement toutes allouées en pile par la VPU (les registres sont plutôt utilisés pour stocker les résultats intermédiaires). Le temps d'aller chercher des données en mémoire centrale, de faire le calcul et de mettre à jour ces cases mémoire, même en utilisant le cache du CPU, est beaucoup plus lent que d'effectuer le calcul directement avec des registres.

Cette limite de la VPU est significative : comme le compilateur dynamique est de moins bonne qualité que celui d'IBM, toutes les autres performances de la JnJVM sont dégradées d'un facteur assez grand par rapport à la machine virtuelle d'IBM. Cette différence de performance n'est pas due à l'architecture adoptée mais bien à un problème d'ingénierie qui n'a pas été effectué faute de temps.

La figure 5.9 présente l'ensemble des résultats. Un point est aussi à noter : le rapport entre les performances du calcul de la JVM d'IBM et de la JnJVM reste relativement le même quelque soit le type de calcul entier. En revanche, la machine virtuelle Kaffe étant sans JIT, le rapport change en fonction du test et passe à 1/100 des performances de la JnJVM pour les tests `maxInt`, `absInt` et `minInt`.

	JnJVM	IBM	Kaffe
millions d'additions/s	85.72	653.56	3.69
millions de multiplications/s	69.36	329.62	3.67
millions de divisions/s	20.56	177.51	3.22
millions de absInt/s	32.04	321.27	0.34
millions de maxInt/s	29.95	251.10	0.25
millions de minInt/s	30.47	439.40	0.23

FIG. 5.9 – Arithmétique entière

4.3 Calculs flottants

La VPU ne gère pas les types double, flottant et entier long sur 8 octets (voir la section 3 du chapitre 3). La bibliothèque externe qui émule ces calculs (voir section 3.3) impose plusieurs appels de fonctions pour chaque opération sur ces types. Cette émulation est beaucoup plus lente que l'utilisation directe de l'unité de calcul flottante de la machine et dégrade très nettement les performances de la JnJVM. L'addition de doubles est en moyenne 68.3 fois plus lente dans la JnJVM que dans la JVM d'IBM et 1.95 fois plus lente dans la JnJVM que dans la JVM Kaffe. La JnJVM, bien que possédant un compilateur JIT, devient plus lente qu'une machine virtuelle interprétée. Ce problème est dû à un manque dans la VPU et non à l'architecture Machine Virtuelle Virtuelle. La figure 5.10 présente quelques résultats obtenus avec l'arithmétique double. Les résultats en arithmétique flottante ou avec des entiers longs codés sur 8 octets sont sensiblement dans les mêmes zones.

	JnJVM	IBM	Kaffe
millions d'additions/s	1.92	130.47	3.74
millions de multiplications/s	1.94	131.23	3.64
millions de divisions/s	1.95	131.84	3.69
millions de (int->double)/s	3.53	33.47	3.48
millions de (long->double)/s	1.53	4.82	2.09
millions de absDouble/s	1.68	394.80	0.32
millions de maxDouble/s	1.43	60.60	0.04
millions de sinDouble/s	0.68	4.19	0.22
millions de roundDouble/s	0.27	1.95	0.10

FIG. 5.10 – Arithmétique double

Ces résultats, particulièrement éloignés des performances de la JVM d'IBM, ont des conséquences sur tous les tests effectuant des calculs flottants, en particulier les sections deux et trois du bench JavaGrande (voir section 4.6). Pour récapituler, le calcul entier dans la JnJVM est, en moyenne, 7 fois plus lent que dans la JVM d'IBM et le calcul avec d'autres types numériques, 70 fois plus lent. Ces résultats sont à mettre sur le compte du prototype de compilateur en ligne utilisé et non sur le compte de l'architecture des applications actives. En effet, avec un compilateur en ligne adéquat, il n'y a aucune raison pour que les additions entières ou flottantes soient plus lentes que sur la machine virtuelle d'IBM.

4.4 Allocations d'objets

Deux séries de tests ont été effectuées pour mesurer la vitesse de l'allocateur d'objets de la JnJVM, le premier en utilisant le ramasse-miettes coloré, appelé JnJVM-c, le second en utilisant le ramasse-miettes générationnel, appelé JnJVM-g (voir la section 3.1 du chapitre 3). Ces mesures spécifiques avec les deux ramasse-miettes n'ont pas été réalisées pour les autres tests présentés dans cette section car les différences de performances ne sont pas significatives : le nombre d'allocations mémoire est trop petit.

Allocations (en 100000/s)	JnJVM-c	JnJVM-g	IBM	Kaffe
Tableaux de 4 entiers	9.64	11.46	54.63	3.38
Tableaux de 16 entiers	7.33	9.33	25.69	3.14
Tableaux de 128 entiers	1.64	3.80	2.38	1.99
Tableaux de 4 flottants	8.20	11.06	54.66	3.35
Tableaux de 16 flottants	4.81	8.30	25.66	3.11
Tableaux de 128 flottants	1.08	3.02	2.40	1.99
Tableaux de 4 objets	8.24	7.90	54.30	0.89
Tableaux de 16 objets	4.87	3.66	24.09	0.87
Tableaux de 128 objets	1.08	0.61	2.40	0.74
Objets de base (<code>Object</code>)	8.47	13.92	527.19	1.86
Objets simples avec ctr	11.65	13.53	23.56	1.26
Objets simples avec 4 champs	9.34	11.65	21.19	1.25
Objets complexes avec ctr	5.83	5.74	18.82	0.61

FIG. 5.11 – Test de la gestion mémoire

Les tests d'allocations d'objets mesurent deux performances simultanément, la vitesse d'allocation et la vitesse du ramasse-miettes. Le compilateur en ligne intervient moins dans cette série de tests que dans les précédentes : le rapport entre la JVM d'IBM et la JVM Kaffe est plus faible (25 à 30 fois plus lent en moyenne). La JnJVM est entre 1.5 fois plus rapide et 6 fois plus lente que la JVM d'IBM, même si la JnJVM reste globalement plus lente que la JnJVM.

La différence entre les ramasse-miettes générationnel et coloré provient de la durée de vie moyenne des objets alloués pendant ce test : tous les objets alloués ont une durée de vie courte. Ils peuvent donc être détruits relativement rapidement et seules de jeunes collections ont lieu, ce qui évite d'avoir à parcourir toute la mémoire objet à chaque collection. Il faut toutefois noter les résultats des allocations de tableaux d'objets et d'objets complexes. Ils sont moins bons avec le ramasse-miettes générationnel. Cette différence s'explique par l'algorithme de vieillissement d'objets présentés dans la section 3.1.4 du chapitre 3 : à chaque affectation d'objet, une fonction est appelée pour vérifier si l'objet doit vieillir. Cette fonction n'est pas optimisée, elle doit trouver l'entête des objets, vérifier leur âge respectif puis, éventuellement, faire vieillir l'objet cible. Tant que cette partie de l'algorithme n'aura pas été optimisée, il n'est pas rentable d'utiliser le ramasse-miettes générationnel.

Le résultat de l'allocation d'objets de base dans la JVM d'IBM est dû à une optimisation : comme l'objet de base ne possède pas de champ, une seule référence est nécessaire pour tous les objets de ce type.

Les optimisations présentées dans la section 3.1.4 du chapitre 3 permettent d'augmenter

les performances du ramasse-miettes, ce qui suffit à expliquer la différence de performances observée entre la JnJVM et la machine virtuelle d'IBM. La présence de nombreux composants de description de l'état de la JnJVM n'intervient pas dans les tests d'allocation avec le ramasse-miettes générationnel : ces objets n'augmentent pas la durée des collections dans ce cas. En revanche, avec le ramasse-miettes coloré, ces objets sont parcourus à chaque collection ce qui ralentit le ramasse-miettes.

4.5 Appels de méthodes

Les tests d'appels de méthodes mesurent le nombre d'appels que peut effectuer la machine virtuelle sur la même méthode par seconde. La figure 5.12 récapitule les résultats. Une première constatation se dégage de ce test : la machine virtuelle Kaffe avec son interpréteur est très nettement plus lente que les machines virtuelles avec compilateur en ligne (sauf avec les appels synchronisés de la JnJVM). Bien que les méthodes appelées soient très petites (juste une addition), la présence du compilateur en ligne permet d'obtenir de meilleurs performances.

Nombre d'appels (en millions/s)	JnJVM	IBM	Kaffe
méthode d'instance de la même classe	24.92	45.78	0.23
méthode d'instance synchronisée de la même classe	0.67	12.27	0.20
méthode d'instance finale de la même classe	23.27	83.48	0.22
méthode de classe de la même classe	30.46	88.04	0.22
méthode de classe synchronisée de la même classe	0.67	12.91	0.20
méthode d'instance d'une autre classe	22.31	59.80	0.24
méthode abstraite d'une autre classe	23.72	59.57	0.23
méthode de classe d'une autre classe	30.83	75.32	0.25

FIG. 5.12 – Appels de méthodes

La JnJVM utilise la VPU pour effectuer ses appels. La VPU produit un code moins bien optimisé que la machine virtuelle d'IBM sur les allocations de registres (voir section 4.2), ce qui explique la différence de performances entre la machine virtuelle d'IBM et la JnJVM qui est deux à trois fois plus lente. Les résultats des appels synchronisés de la JnJVM sont mauvais. Aucun effort n'a été fait pour améliorer ces performances : les mutex POSIX (`pthread_mutex`) sont utilisés et deux indirections sont nécessaires pour appeler ces fonctions. Une optimisation de cet aspect de la JnJVM sera à effectuer.

Les appels aux méthodes finales permettent d'éviter d'avoir à gérer le polymorphisme car la méthode cible est clairement définie. En utilisant cette propriété, la JVM d'IBM optimise ces appels, ce qui n'est pas le cas de la JnJVM.

Les appels de la JnJVM, aux optimisations près, restent dans le même ordre de grandeur que les appels de la JVM d'IBM. Pour obtenir des résultats similaires, il faudrait optimiser le code généré par la VPU, les appels synchronisés et la gestion des classes finales. Ces performances restent toutefois acceptables dans l'ensemble (la plupart des appels sont non synchronisés) et montrent que l'architecture de la JnJVM n'entraîne pas de pertes de performances notables sur ce point.

4.6 Conclusions sur les mesures

Les résultats des autres mesures effectuées sur le bench JavaGrande, exception, affectation et calcul mathématique, sont du même ordre de grandeur. Les différences de performances entre la JnJVM et la machine virtuelle d'IBM s'expliquent principalement par les algorithmes utilisés qui sont loin d'être optimisés et par le code produit par la VPU qui n'est pas au même niveau de maturité que le compilateur en ligne de la JVM d'IBM. En revanche, il n'y a pas de contradiction entre de bonnes performances et l'architecture de la JnJVM, mais seuls des prototypes optimisés de μvm et de JnJVM pourraient prouver définitivement cette assertion.

Outre les performances des opérations de base, le bench JavaGrande évalue aussi les performances des machines virtuelles Java avec des applications tests représentatives des applications Java. Ces tests n'apportent pas d'informations supplémentaires sur les performances de la JnJVM : le rapport des performances entre la JVM d'IBM et la JnJVM reste proche des rapports obtenus avec les opérations de base. La note globale de la JnJVM est de 0.48 ce qui la place au même niveau que la première machine virtuelle de Sun (HotSpot 1.0 datant de 1995) et à un septième des machines virtuelles commerciales d'IBM ou de Sun.

Ce résultat global est plutôt positif compte tenu du temps d'implantation de la JnJVM (une personne pendant six mois pour la JnJVM et deux personnes pendant six mois pour la μvm). En effet, IBM travaille sur l'optimisation de leur machine virtuelle Java depuis 1996 [jav]. Ces neuf années d'optimisation et d'accumulations d'expériences expliquent largement les différences de performances observées : d'un facteur 2 à 10 pour la plupart des opérations et d'un facteur allant jusqu'à 100 pour les exceptions et jusqu'à 70 pour les calculs flottants.

5 Conclusion

Dans ce chapitre, nous avons présenté la JnJVM, une machine virtuelle Java adaptable dynamiquement. La JnJVM est une machine virtuelle Java générique, dans le sens où elle respecte les spécifications J2SE [LY]. La JnJVM est spécialisable par une application active. Ces spécialisations reposent sur le MOP de la μvm : les champs et les fonctions des composants peuvent être remplacés dynamiquement en chargeant d'autres MVlets. La JnJVM répond aux besoins d'homogénéité des environnements et permet d'intégrer rapidement des innovations technologiques, ce qui ne peut actuellement pas être fait dans les JVM standards sans passer par un processus de normalisation et de développement long.

L'évaluation quantitative de la JnJVM montre que les performances sont acceptables compte tenu de la maturité de la JnJVM et place la JnJVM au même niveau que la machine virtuelle Java à JIT Hotspot 1.0. Ce point est essentiel : un haut degré d'adaptabilité n'est pas en contradiction avec de bonnes performances, pourvu qu'on dispose d'un MOP rapide et d'un compilateur dynamique. Il est donc possible de construire des environnements flexibles tout en utilisant la flexibilité pour augmenter les performances.

Dans le chapitre suivant, une évaluation qualitative de la JnJVM et de la μvm est présentée. Elle démontre le haut degré d'adaptabilité dynamique de notre environnement d'exécution. Cette évaluation passe par la réalisation de MVlets de spécialisation répondant à des problèmes réels dans les environnements Java : adaptabilité dynamique, tissage d'aspects, analyse d'échappement et migration de fil d'exécution.

CHAPITRE 6

Réalisation de quatre applications d'évaluation de la JnJVM

Sommaire

1	Introduction	101
2	Adaptation du langage d'entrée	102
3	La JVM à analyse d'échappement	107
4	Tissage dynamique d'aspects	119
5	La JVM à migration de fil d'exécution	123
6	Récapitulatif	129

1 Introduction

Ce chapitre présente une évaluation qualitative du degré d'adaptabilité de la JnJVM et de la μvm . Les exemples choisis montrent comment adapter les différents niveaux d'une Machine Virtuelle Virtuelle (voir le chapitre 1 d'introduction) : le langage, la μvm , les MVLets et les applications passives. L'ensemble des adaptations présentées dans ce chapitre ne nécessite pas la moindre modification dans le code de la μvm ou de la JnJVM. Toutes les adaptations se font pendant l'exécution : le protocole méta-objet et le compilateur en ligne de la μvm suffisent à modifier dynamiquement le comportement de la μvm , de la JnJVM ou de toute autre MVV. Les applications actives présentées dans ce chapitre sont donc portables dans le sens où elles ne requièrent pas d'environnement dédié.

La figure 6.1 résume les différentes MVLets présentées dans ce chapitre et le niveau qu'elles adaptent dans l'environnement (flèches continues sur la figure). Ces applications se séparent en quatre grandes thématiques :

- langage (voir la section 2) ;
- tissage d'aspects (voir la section 3.2.4 du chapitre 2 pour la notion de tissage d'aspects et la section 4 de ce chapitre pour la MVLet) ;
- migration de fil d'exécution (voir section 5) ;
- analyse d'échappement [CGS⁺99, Bla03] (voir section 3).

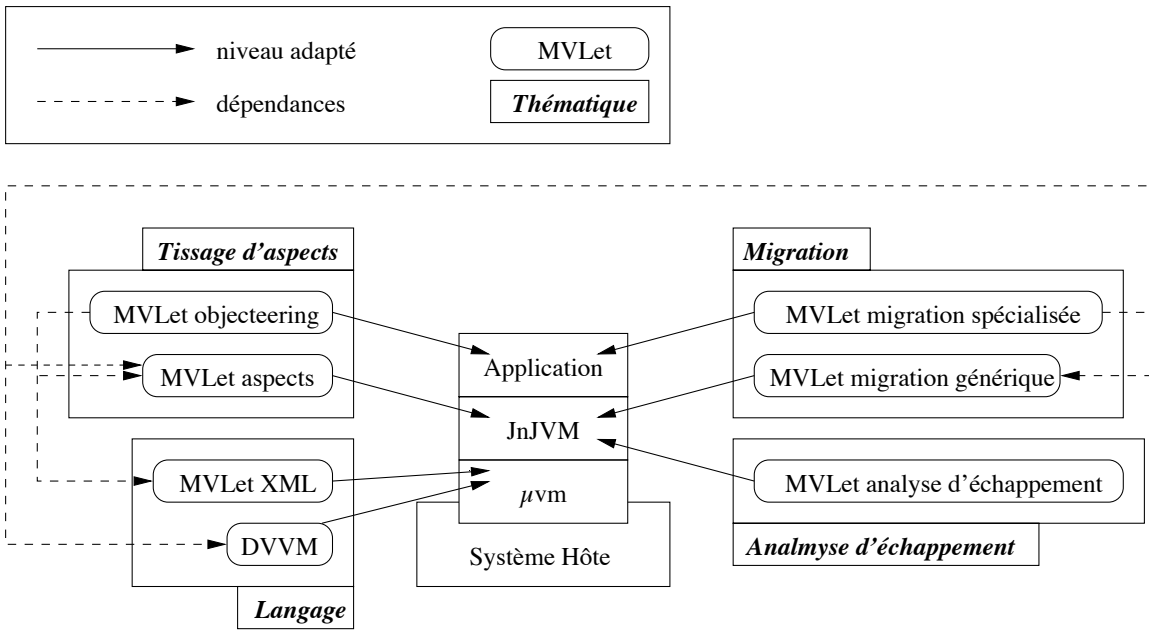


FIG. 6.1 – Thématiques abordées dans ce chapitre et niveau adapté dans une MVV

Les flèches en pointillé sur la figure 6.1 symbolisent les dépendances entre ces travaux. Ces dépendances montrent que les différents niveaux d'adaptation sont liés. Les deux MVLets d'adaptation de l'application dépendent d'au moins une adaptation de la JnJVM et d'une adaptation du langage d'entrée, et les adaptations du langage d'entrée nécessitent des adaptations dynamiques de la μvm . Les quatre niveaux d'adaptation décrits dans le chapitre 1 d'introduction sont intimement liés : plus l'adaptation est proche de l'application, plus le nombre de niveaux couverts peut être grand.

Les différents travaux présentés sur la figure 6.1 sont décrits dans la suite de ce chapitre. Plutôt que de structurer la suite de ce chapitre en fonction du niveau d'une machine virtuelle virtuelle adapté, le plan prend en compte la proximité entre ces travaux pour simplifier la compréhension des différents sujets. La suite du chapitre se décompose en cinq sections. La première section présente les MVLets d'adaptation du langage d'entrée de la μvm . La MVLet d'analyse d'échappement est présentée dans la section 3 et nos travaux autour du tissage d'aspects en Java dans la section 4. La section 5 présente notre étude de la migration dans les environnements Java et la section 6 conclut et récapitule ce chapitre.

2 Adaptation du langage d'entrée

La langage d'entrée de la μvm est un dialecte de Lisp proche de Scheme (voir le chapitre 3). Ce langage présente l'avantage d'être simple à lexer et à parser mais n'est pas adapté pour communiquer sur un réseau (les programmes se présentent sous forme de source) et ne peut pas être utilisé avec des programmes écrits dans des langages préexistants (le dialecte utilisé ne correspond à aucune spécification de langage standard). Pour comprendre d'autres formes de langage, rien n'empêcherait une MVLet de construire son propre lexer, parseur et compilateur au niveau applicatif, indépendamment de la μvm , mais cette approche ne

réutilise pas l'effort de développement de la μvm .

De manière à réutiliser au maximum le cycle de compilation mis en place dans la μvm , les composants internes lexeur et parseur de la μvm présentés dans le chapitre 3 sont adaptés dynamiquement par une application active à l'aide du MOP de la μvm .

2.1 La JVM adaptable à distance

Une application active reposant sur la JnJVM construit son environnement, mais une fois l'application Java lancée, il n'y a plus aucun moyen d'interagir dynamiquement avec le cœur de la MVV : la porte de communication avec la MVV est fermée. Une fois cette porte de communication fermée, plus aucune adaptation n'est possible. De plus, la porte de communication d'une MVV hérite des propriétés de la μvm et ne peut que lire des programmes via l'entrée standard ou via un fichier d'une partition montée localement, ce qui demande d'avoir un accès local à la machine.

Pour remédier à ces deux problèmes, nous avons construit une MVLet d'adaptation à distance appelée DVVM (pour Distributed Virtual Virtual Machine). Ces travaux ont été effectués conjointement par C. Martin [Mar03] pendant son stage de DEA et moi-même. Cette MVLet est exécutable directement dans la μvm : elle peut être utilisée simultanément avec la JnJVM ou toute autre MVV.

La DVVM modifie dynamiquement le lexeur et le parseur de la μvm et réutilise la chaîne de compilation native de la μvm . La structure des composants de la μvm , réfléchi au niveau applicatif, est utilisée pour adapter en profondeur le langage d'entrée.

2.1.1 La MVLet d'adaptation à distance

La MVLet d'adaptation à distance repose sur de la migration de code. Le code est échangé entre MVVs sous forme d'arbres de syntaxe abstraite (AST) sérialisés. La DVVM se sépare en deux parties :

- une partie serveur : cette partie est chargée sur la MVV qui envoie le code ;
- une partie cliente : cette partie est chargée sur la MVV qui reçoit le code.

La figure 6.2 présente l'architecture de la DVVM. Le serveur, ou console d'administration, charge la partie serveur. Elle lit un fichier μvm , le parse, construit les AST des applications fonctionnelles et, au lieu de les compiler et de les exécuter, la console d'administration sérialise les AST et les envoie au client.

Le client est constitué d'une MVV (la Machine Virtuelle Virtuelle dans laquelle est chargée la DVVM) et d'un processus de contrôle exécuté dans un autre processus léger. Ce processus de contrôle reçoit les arbres sérialisés émis par le serveur, les compile et les exécute. Cette exécution modifie l'état de la mémoire partagée entre la MVV d'origine et le processus de contrôle. La réception des AST sérialisés modifie l'état de la MVV exactement comme le ferait le chargement d'un fichier.

2.1.2 Le prototype

Le prototype ne fait aucune assertion sur le canal utilisé pour faire communiquer le serveur et le client. Pour abstraire le prototype de l'envoi et de la réception, un composant (le composant Comm. sur la figure 6.2) encapsule la communication. Celui-ci est constitué de deux fonctions d'interface : une fonction d'émission de caractères et une fonction de réception

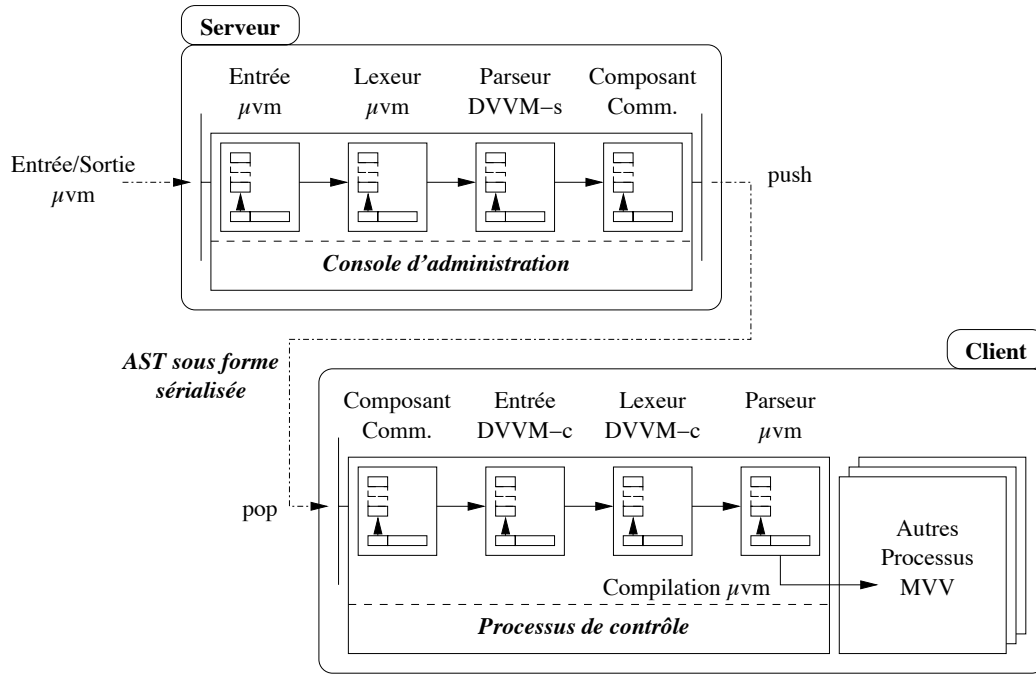


FIG. 6.2 – Fonctionnement de la DVVM

de caractères (**push** et **pop** sur la figure 6.2). Le serveur et le client sont libres d'utiliser l'implantation qui leur convient et donc d'utiliser le média de communication adéquat.

Le prototype de DVVM réalisé utilise le MOP interne de la μvm pour modifier les composants lexeur et parseur chez le client et chez le serveur (voir les section 3.2 et 3.3 du chapitre 3).

Le serveur. Le serveur réutilise l'entrée et le lexeur standard de la μvm pour lire et lexer un fichier ou l'entrée standard. Le parseur du serveur est légèrement modifié par la DVVM pour permettre la sérialisation, car, par défaut, les symboles sont résolus par le parseur dans le module courant. Si le serveur se trouve dans le module **A** et si le symbole ayant pour chaîne de caractères **.a** est rencontré, le lexeur renverra le symbole **A.a**. Si le client se trouve dans un module **B**¹, il ne faut pas envoyer directement le symbole **A.a** qui ne correspond pas à la sémantique attendue par le développeur, mais une forme intermédiaire contenant la chaîne **.a**.

Les pseudo-symboles sont définis dans la DVVM et sont une représentation sérialisable des symboles parsés. Ce sont des composants qui héritent du composant interne chaîne de caractères (**String**) de la μvm . Lorsque le lexeur trouve un symbole, la fonction de construction de symbole du parseur est automatiquement appelée. La DVVM change donc le pointeur de cette fonction vers une fonction qui s'occupe de générer un pseudo-symbole à la place du symbole. Cette manipulation ne pose pas de problème en multi-tâches car chaque tâche possède son propre lexeur/parseur : les autres tâches utilisent toujours le lexeur, le parseur et le compilateur natif de la μvm .

¹Parce que, par exemple, un AST précédent a déplacé le module courant.

Une fois l'arbre syntaxique produit, il est sérialisé. Les arbres ne peuvent contenir que 4 types d'objets : des entiers, des chaînes de caractères, des pseudo-symboles ou des listes. Une liste est sérialisée en émettant un identifiant de liste, la taille de la liste et les objets constituant cette liste. Les autres objets sont sérialisés en émettant un identifiant suivi de la donnée associée (une chaîne ou un entier). Les entiers sont automatiquement sérialisés au format réseau pour être portable.

Le client. Le client reçoit de la DVVM serveur les arbres sérialisés par le composant de communication et reconstruit les arbres de syntaxe abstraite. L'entrée du processus de contrôle reçoit ses données via le composant de communication et envoie les caractères reçus au composant lexeur du client. Celui-ci est spécialisé pour recevoir des arbres sérialisés : à chaque identifiant de type d'objet sérialisé est associé une fonction dans le lexeur de la DVVM cliente (lexeur DVVM-c sur la figure 6.2). Le lexeur du client produit les mêmes données que le lexeur natif de la μvm : le parseur du client n'a pas besoin d'être modifié. Les arbres sont reconstruits et compilés, exactement comme si ils avaient été construits avec la chaîne de chargement native de la μvm .

2.1.3 Utilisation de la DVVM dans la JnJVM

La DVVM est utilisée dans la JnJVM pour charger des MVLets de modification via un réseau. La MVLet JnJVM adaptable à distance utilise TCP pour communiquer. Ce choix est le plus naturel compte tenu de l'utilisation actuelle de la JnJVM adaptable à distance, mais rien n'empêche une application active de réutiliser le travail effectué et de l'adapter à d'autres types de communication.

Chez le client, une fonction s'occupe d'attendre la connexion d'un serveur et de lancer un processus léger par serveur connecté. Le serveur quand à lui initie la connexion vers le client². Le serveur peut aussi posséder une liste de clients et envoyer à chaque client un AST. De cette façon, la console d'administration ne s'occupe plus d'une MVV unique mais d'un réseau de MVV. Dans ce cas, les connexions réseaux et les MVVs sont considérées comme fiables : il n'y a pas de protocole de détection de fautes. Une MVLet de tolérance aux fautes, au dessus de la DVVM, pourrait s'occuper de gérer cet aspect.

La MVLet JnJVM adaptable à distance commence par charger la DVVM puis la JnJVM. L'exécution s'effectue donc de manière normale, mais une porte de communication avec la JnJVM est préservée pendant l'exécution de l'application Java. Cette porte de communication sert à injecter du code dynamiquement dans la JnJVM pour l'adapter dynamiquement. La DVVM est utilisée dans la MVLet de tissage dynamique d'aspects dynamique pour injecter dynamiquement de nouveaux aspects (voir section 2.2) et dans la MVLet de migration (voir section 5) pour échanger du code.

Les aspects sécurités sont en projet dans la DVVM et la JnJVM adaptable à distance. Les différentes solutions permettant de sécuriser le prototype sont principalement des techniques d'identification de serveurs connus, de cryptage des arbres sur le réseau, de systèmes d'isolation et de suppression de symboles jugés dangereux de la μvm .

La MVLet d'adaptation à distance montre la flexibilité du cœur de la μvm : les composants internes de la μvm peuvent être manipulés de la même manière que les composants

²Les sémantiques client/serveur dans la JnJVM adaptable à distance sont inversés par rapport aux sémantiques client/serveur en TCP.

applicatifs à l'aide du MOP développé pendant ces travaux. Cette flexibilité permet de réutiliser les mécanismes de la μvm et de les adapter à un besoin particulier, à savoir séparer la partie lexeur/parseur de la partie compilation/exécution.

2.2 Une entrée XML pour la μvm

La μvm ne peut que lire des fichiers écrits dans un dialecte de Lisp proche de `scheme`. Cette restriction empêche la réutilisation de fichiers écrits dans d'autres langages. De nombreux outils utilisent le XML comme format d'échange de document. Réutiliser ces fichiers permet à la μvm de s'intégrer facilement avec ces outils. La MVLet XML change le langage d'entrée de la μvm pour du XML. Elle a été développée par N. Geoffray [Geo04] pendant son stage de deuxième année de magistère, encadré par B. Folliot et moi-même. La section 4.1 montre une utilisation concrète de cette MVLet avec l'outil de modélisation Objectteering [obja].

2.2.1 Un texte XML

Un texte XML est formé de balises. Chaque balise possède des attributs, des sous-balises et du texte. Un attribut associe une clé à une valeur sous forme de texte. La figure 6.3 montre la syntaxe d'un fichier XML.

```

1 <personne id="1">
2   <nom> Luck Skywalker </nom>
3   <age> 22 </age>
4   Des mots peuvent se trouver ici...
5 </personne>

```

FIG. 6.3 – La syntaxe XML

Le sens donné aux balises est indépendant du fichier XML : un fichier XML est juste une description d'arbre XML.

2.2.2 Le lexeur XML

On constate qu'une balise XML est assez proche d'une liste μvm contenant des balises et du texte. Toutefois, les listes μvm ne peuvent pas exprimer les attributs de balise, c'est pourquoi un nouveau type de composant **Balise**, héritant des listes μvm , a été créé en utilisant le MOP. Une balise XML hérite d'une liste μvm avec trois nouveaux attributs : une liste μvm d'attributs, une balise parente³ et un module courant qui indique le module dans lequel doivent être placés les symboles des sous-balises⁴.

Le lexeur et le parseur de la μvm sont modifiés pour comprendre la syntaxe XML en utilisant le MOP de la μvm . De nombreuses possibilités ont été implantées pour pouvoir

³Qui est remplie à la demande par le lexeur/parseur.

⁴Le parseur peut, au choix, créer les symboles des sous-balises dans ce module ou dans le module courant.

lexer et parser un fichier XML. Le rapport de stage de N. Geoffray [Geo04] donne de plus amples renseignements sur ces possibilités.

L'utilisation qui est faite du lexeur/parseur XML dans notre cas réutilise les symboles μvm . Le premier élément d'une balise (vue comme une liste) est le symbole associé à la balise. La fonction μvm de compilation des listes est alors directement utilisable : une balise est une liste. La fonction de compilation va d'abord chercher si il existe une syntaxe associée au symbole. Si c'est le cas, celle-ci est appelée avec la balise et le compilateur courant en argument. Si ce n'est pas le cas, la fonction stockée dans le champ valeur du symbole est exécuté.

```

1 (define (syntax .personne)
2   (lambda (balise compiler)
3     ( :system.printf "; compiling balise : %s\n" ( :object.print-string balise))
4     ( :system.printf "; attributs : %s\n"
5       ( :object.print-string ( :xml.attributs balise)))
6     '0))

```

FIG. 6.4 – Une feuille de style XML

La figure 6.4 présente une feuille de style pour le fichier XML de la figure 6.3. La syntaxe **personne** reçoit la balise **personne** lorsque celle-ci est compilée. La syntaxe ne fait qu'afficher des informations sur la balise et ne compile rien à la place (elle renvoie l'arbre '0).

2.2.3 Utilisation

En utilisant la MVLet XML et une MVLet associée à un modèle de document XML, tout fichier XML peut être exécuté dans la μvm . Dans le cadre de la MVLet Objectteering, le code XML produit par un outil de modélisation est directement exécuté pour intégrer les modifications apportées au modèle de l'application (voir section 4.2). Dans le cadre du tissage d'aspects (voir section 4.1), des travaux sont actuellement en cours pour utiliser un langage de description d'aspects basé sur XML.

Ce second exemple d'adaptation du langage d'entrée de la μvm montre la souplesse des composants internes de la μvm . En exposant au niveau applicatif ces structures, la chaîne de compilation de la μvm est réutilisée, ce qui diminue très fortement le travail de développement.

3 La JVM à analyse d'échappement

La MVLet d'analyse d'échappement vise à optimiser les performances du ramasse-miettes et à éliminer les synchronisations inutiles pour des objets utilisés dans une seule tâche. Cette MVLet [TOG⁺05] a été conçue pour tester les travaux d'A. Galland [Gal05] sur le même sujet.

3.1 L'analyse d'échappement

L'analyse d'échappement [CGS⁺99, Bla03] est l'étude systématique de la durée de vie des objets alloués. Cette analyse indique à partir de quel point (en Java, quel bytecode) certains objets peuvent être libérés. Un objet est dit capturé si l'analyse d'échappement donne une borne connue à la durée de vie d'un objet. Dans le cas contraire, un objet est dit s'échappant. Les informations fournies par l'analyse d'échappement peuvent être utilisées pour optimiser la gestion mémoire et la gestion de la synchronisation entre processus légers.

3.1.1 Gestion mémoire

Un outil d'analyse d'échappement indique le point à partir duquel un objet n'est plus utilisé. Un objet capturé peut donc être libéré automatiquement en ce point, sans avoir à recourir au ramasse-miettes. La charge de travail à effectuer par le ramasse-miettes est donc diminuée.

Pour effectuer cette optimisation dans une machine virtuelle Java, il faut allouer les objets capturés indépendamment du ramasse-miettes, avec un autre système d'allocation mémoire⁵, et libérer ces objets après l'exécution du bytecode à partir duquel l'objet n'est plus utilisé.

Le point où l'objet i est alloué est noté a_i et le point, fourni par l'outil d'analyse d'échappement, où l'objet i peut être libéré est noté b_i . La durée de vie d'un objet capturé, i , s'exprime sous la forme $d_i = [a_i, b_i]$: elle regroupe toutes les instructions pendant lesquelles l'objet i est vivant. Si quels que soient les objets capturés i et j , on a $d_i \cap d_j \in \{\emptyset, d_i, d_j\}$, c'est-à-dire si les durées de vie des objets ne peuvent pas être entrelacées, alors les objets peuvent être libérés dans l'ordre inverse de leur allocation : une pile peut alors être utilisée pour allouer où libérer les objets. L'outil d'analyse d'échappement développé par A. Galland et utilisé pour ces travaux satisfait à ce critère.

3.1.2 Gestion des synchronisations

Les outils d'analyse d'échappement ne sont pas capables de trouver quand un objet affecté dans une variable globale (en Java, un champ statique d'une classe) peut être libéré. Par conséquent, tout objet affecté à une variable globale (ou une variable atteignable à partir des variables globales) est considéré comme s'échappant.

Cette propriété peut être utilisée pour diminuer le nombre de synchronisations. En effet, un objet capturé ne va jamais être atteignable via les variables globales et ne peut donc pas être échangé avec une autre tâche : tout objet capturé reste dans les variables locales d'une tâche. Les synchronisations effectuées sur ces objets n'ont alors plus de raison d'être. En utilisant les informations fournies par l'outil d'analyse d'échappement, la machine virtuelle Java peut supprimer toutes les synchronisations sur les objets capturés.

3.1.3 L'outil d'analyse statique

L'analyse d'échappement de bytecode Java menée par A. Galland permet de savoir si un objet alloué dans une méthode va s'échapper de cette méthode. Si l'objet ne s'échappe pas, il peut être automatiquement détruit à la sortie de la méthode. La durée de vie d'un

⁵Ou via le ramasse-miettes, mais en indiquant à celui-ci que l'objet ne peut pas être libéré

objet est incluse dans une méthode et la borne supérieure de cette durée est la fin d'une méthode : les durées de vie des objets ne sont donc jamais entrelacées et une pile peut être utilisée pour allouer les objets.

```
1 class Escape {  
2     public Object f() {  
3         Object a = new Object();  
4         System.out.println("a vaut " + a);  
5         return new Integer(22);  
6     }  
7 }
```

FIG. 6.5 – Un exemple d'échappement en Java

La figure 6.5 présente un exemple de fonctions Java allouant deux objets. Le premier objet alloué, `a`, ne s'échappe pas de la méthode, alors que le second, `Integer(22)` s'échappe. L'outil d'analyse d'échappement statique est capable de repérer que l'objet `a` ne s'échappera pas et la machine virtuelle Java peut exploiter cette information pour détruire cet objet directement à la sortie de la méthode.

Pour exploiter l'information donnée par l'outil statique d'analyse d'échappement, deux problèmes doivent être résolus :

- cette information doit être transmise à la machine virtuelle Java ;
- cette information doit être exploitée par la JVM.

Pour résoudre le premier problème, il faut créer un nouvel attribut de code indiquant les index des bytecodes qui allouent des objets capturés. L'outil d'analyse statique enrichie donc le binaire Java avec ce nouvel attribut, appelé `EscapeMap`. Il se présente sous la forme d'une liste d'index dans le bytecode de la méthode. Chaque index correspond à un bytecode d'allocation (`new`, `newarray`, `anewarray` ou `multianewarray`) et indique que l'objet alloué en ce point peut être détruit à la sortie de la méthode.

Le second problème est plus difficile à résoudre : il faut pouvoir modifier une machine virtuelle Java pour qu'elle puisse exploiter cette information. L'analyse ne peut alors être exploitée que sur des machines virtuelles spécifiques. L'architecture adoptée pour construire la JnJVM permet de s'abstraire de ce problème : l'application active embarque avec elle les mécanismes permettant d'exploiter l'information d'analyse d'échappement et transforme dynamiquement la machine virtuelle Java en machine virtuelle Java à analyse d'échappement. Le problème du déploiement de la technologie ne se pose donc plus : c'est l'application qui apprend à la plate-forme comment exploiter l'analyse d'échappement et non l'application qui doit attendre que les mécanismes existent sur toutes les machines virtuelles Java.

3.2 La MVLet d'analyse d'échappement

La MVLet d'analyse d'échappement [TOG⁺05] est chargée après la JnJVM dans la μvm . L'adaptation est dynamique, en particulier, la MVLet d'analyse d'échappement peut être chargée en utilisant la DVVM pendant l'exécution de l'application.

3.2.1 Suppression des synchronisations et héritage

La MVLet à analyse d'échappement supprime les synchronisations sur les objets capturés et les alloue indépendamment du ramasse-miettes. Pour supprimer les synchronisations, deux solutions sont possibles :

- placer une clause conditionnelle autour des synchronisations de méthode Java pour vérifier si la cible de l'appel est un objet capturé ;
- construire une sous-classe associée à chaque classe d'objet capturé qui redéfinit les méthodes synchronisées pour les rendre non-synchronisées.

La première approche est économique en terme de mémoire mais lente à l'exécution. En effet, une clause conditionnelle est exécutée en plus de chaque synchronisation. La seconde approche est plus lourde en terme de mémoire (une classe est créée pour chaque classe d'objet capturé) mais présente l'avantage d'être plus rapide. La seconde approche a été adoptée dans ces travaux car les contraintes mémoires sont faibles.

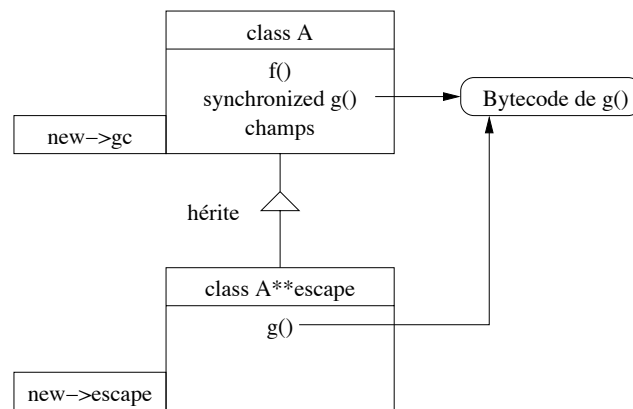


FIG. 6.6 – Héritage et analyse d'échappement

La figure 6.6 présente le fonctionnement de l'héritage lorsqu'un objet capturé de la classe *A* doit être alloué. Une classe, appelée classe d'échappement (la classe notée *A**escape* sur la figure) est créée pendant la compilation du bytecode d'allocation. Cette classe d'échappement redéfinit les méthodes synchronisées (uniquement *g()* sur la figure) et hérite des champs et des autres méthodes. Les nouvelles méthodes redéfinies ne sont plus synchronisées et partagent les attributs de la méthode originale, en particulier le bytecode. Les instances de *A* et de *A**escape* sont donc identiques, mais celles de *A**escape* n'ont plus aucune méthodes synchronisées.

De manière à effectuer la liaison au plus tard, un champ du composant de description de classe est utilisé pour placer la fonction d'allocation. Avant la liaison, cette fonction charge et alloue la classe alors qu'après la liaison, cette fonction ne fait qu'allouer une instance (ce qui évite d'avoir à vérifier si la classe est chargée à chaque allocation). Ce champ est modifié pour les classes d'échappement. En effet, le chargement d'une classe d'échappement se fait en chargeant la classe parente et en redéfinissant les méthodes synchronisées et l'allocation d'un objet capturé se fait sans passer par le ramasse-miettes.

Les description de classes possèdent deux noms : un nom clé (utilisé pour hacher les classes) et un nom d'usage (celui que renvoie la fonction `forName()` de la classe). Les classes

d'échappement gardent le même nom d'usage que les classes originales de manière à rester compatibles avec les fonctions de l'API Java.

Le coût en terme de mémoire des classe d'échappement est relativement faible pour des machines de bureau mais inadéquat pour des machines à faibles capacité mémoire : une description de classe fait 112 octets et chaque classe synchronisée redéfinie utilise 56 octets.

3.2.2 Allocation des objets capturés

Les durées de vie des objets capturés par l'outil statique d'analyse d'échappement ne sont pas entrelacés : une pile est donc utilisée pour allouer les objets capturés. Deux solutions ont été envisagées pour cette allocation :

- allouer les objets capturés dans la pile d'exécution ;
- allouer les objets capturés dans une pile dans le tas.

La première solution présente l'avantage d'être simple à mettre en œuvre : comme le pointeur de pile est automatiquement mis à jour lorsqu'une méthode est quittée ou lorsqu'une exception est levée, aucun travail de libération n'est nécessaire. Toutefois, cette approche présente deux défauts : la taille de la pile d'exécution est limitée, et il faut tout de même effectuer un travail lors de la levée d'une exception et lors du retour d'une fonctions pour gérer la finalisation,

La seconde solution est plus difficile à mettre en œuvre mais permet d'avoir une pile de la taille du tas. Les premiers tests avec cette approche ont montré que si un grand nombre d'objets était alloués en pile dans le tas, les performances décroissaient avec la taille de la pile. En effet, une pile trop grande (de l'ordre de quelques Mo) entraîne une activation du fichier d'échange (swap). Pour palier à ce problème, une limite au nombre d'objets alloués en pile a été fixée.

La seconde approche a été adoptée car elle permet de donner une taille de pile indépendante de la taille de la pile d'exécution. De manière à simplifier le travail, les objets capturés possédant une méthode de finalisation sont tout de même alloués de manière standard via le ramasse-miettes.

Trois points doivent être pris en compte pour allouer les objets capturés :

- Les allocations d'objets capturés doivent être effectués dans la pile d'allocation. Il faut donc modifier la fonction d'allocation des classes d'échappement.
- La pile doit être mise à jour à la sortie des méthodes possédant des bytecodes qui allouent des objets capturés. Un code doit donc être injecté autour de ces méthodes pour mémoriser le pointeur de pile au début de ces méthodes et le remettre à sa précédente valeur à la fin.
- Il faut prendre en compte les exceptions et mettre à jour le pointeur de la pile d'allocation lorsqu'une exception est levée. En effet, la fin des méthodes n'est alors plus exécutée et le pointeur n'est plus mis à jour ce qui peut conduire à une fuite mémoire. Pour prendre en compte ce problème, il faut modifier la fonction de levée d'exception.

3.2.3 L'aspect analyse d'échappement

L'adaptation effectuée dans la JnJVM pour prendre en compte l'analyse d'échappement revient à tisser un aspect sur la gestion mémoire. La coupe de cet aspect est constitué de six points de jonction (voir figure 6.7). Pour mettre à jour les pointeurs de pile au début et à la fin de méthodes possédant des allocations d'objets capturés, il faut modifier la fonction de

compilation générale (appelée compilation de l'entête et de la fin de méthode sur la figure) pour qu'elle prenne en compte cet aspect. Les quatre bytecodes d'allocation doivent aussi être modifiés pour allouer les objets capturés en pile, et enfin, la fonction de levée d'exception du composant de gestion d'exception doit prendre en compte la mise à jour du pointeur de pile d'allocation lors de la levée d'une exception.

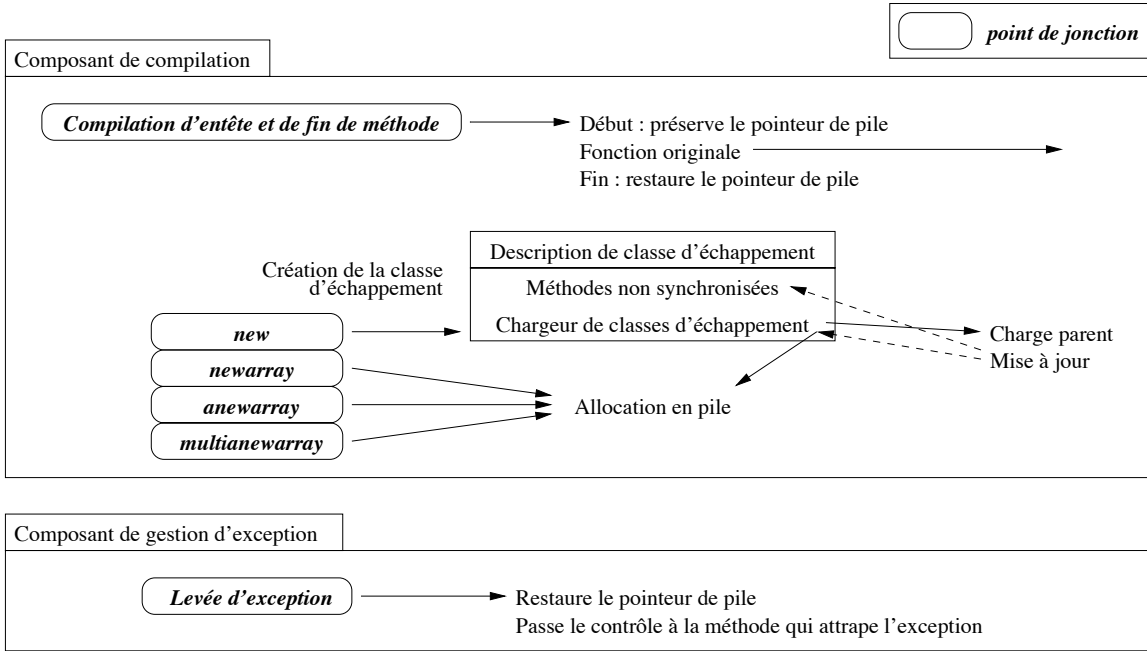


FIG. 6.7 – L'aspect analyse d'échappement

La MVLet d'analyse d'échappement préserve les anciennes valeurs de ces points de jonction : la MVLet d'analyse d'échappement peut donc être déchargée par un administrateur.

3.3 Mise en œuvre de l'allocation

L'algorithme utilisé pour allouer les objets capturés prend en compte deux points : l'allocation elle-même et la collection des objets capturés. En effet, les objets capturés peuvent contenir des références vers des objets s'échappant et le ramasse-miettes doit donc parcourir les objets capturés à chaque collection.

3.3.1 Fonctionnement de la pile

La pile d'allocation contient les objets capturés. Une seconde pile, appelée pile de description, est utilisée pour stocker les valeurs de la pile d'allocation à l'entrée des méthodes possédant des objets capturés.

La figure 6.8 résume l'algorithme à partir d'un exemple. Lors de l'entrée dans une méthode, le pointeur de pile d'allocation est préservé dans la pile de description. Ce pointeur est restauré à la sortie de la méthode. Lorsqu'une exception est levée, le pointeur de la pile d'allocation est mis à jour en calculant le nombre de méthodes ayant des attributs **EscapeMap** à partir de la méthode qui attrape l'exception. Par exemple, sur la figure 6.8, si une exception

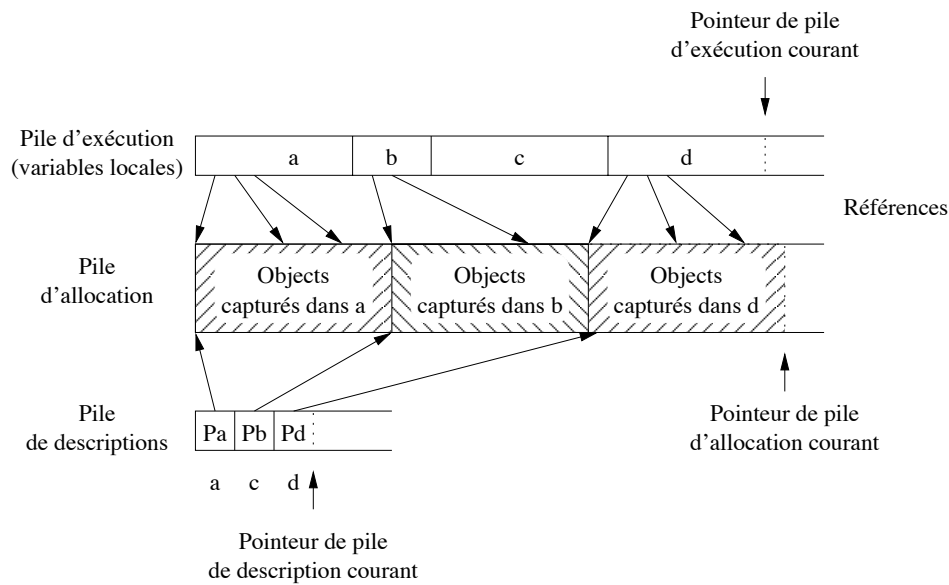


FIG. 6.8 – Algorithme de la pile d'allocation

est levée dans la méthode `d` et attrapée dans la méthode `b`, le nombre de méthodes restant sur la pile d'exécution et possédant un attribut `EscapeMap` est de 2 (`a` et `b`) : le pointeur `Pd` est donc utilisé comme pointeur de pile d'allocation lors de la reprise de l'exécution.

Cet algorithme présente l'avantage de centraliser la gestion de la pile d'allocation lors de la levée d'une exception dans une unique fonction de la `JnJVM`. De cette façon, les méthodes compilées avant d'injecter la `MVLet` d'analyse d'échappement n'ont pas besoin d'être recompilées spécifiquement pour prendre en compte la mise à jour de la pile.

Chaque objet alloué dans la pile est précédé de sa taille et de sa table virtuelle. La table virtuelle maintient la compatibilité avec les objets alloués normalement et la taille est utilisée pour les collections (voir sous-section suivante). Lorsque la pile d'allocation arrive à saturation, l'allocation est effectuée avec l'allocateur normal. De plus, les gros objets sont alloués directement avec l'allocateur par défaut pour éviter de dépasser trop vite la limite de la pile. Un objet qui ne s'échappe pas va donc être alloué dans la pile si la pile n'est pas saturée et si l'objet a une petite taille. Les constantes choisies donnent de bons résultats pour les tests (4Mo pour la limite de pile et 1024 octets pour la taille d'objets) et dépendent de la quantité de mémoire de la machine.

La finalisation n'a pas été gérée mais ne poserait pas de problèmes : il suffit, à chaque fois que le pointeur de pile d'allocation est restauré, de parcourir les objets dans la pile d'allocation en utilisant la taille des objets.

3.3.2 Compilation des bytecodes d'allocation

Les objets alloués en pile peuvent être des racines pour les objets alloués par le ramasse-miettes : si une collection est démarrée, il faut tracer les objets de la pile. La pile est un composant (on a une pile par tâche) avec une fonction spécifique qui trace la pile. La figure 6.9 présente cette fonction `trace()`. Elle commence par la base de la pile et trouve le

premier objet (au 8^e octet). La fonction `trace()` stockée dans la table virtuelle de cet objet (vt) est appelée avec l'objet en paramètre et la taille de l'objet. La fonction `trace()` de la pile passe ensuite à l'objet suivant à l'aide de la taille de l'objet. Tant que le pointeur de pile d'allocation n'est pas dépassé, l'opération est recommencée.

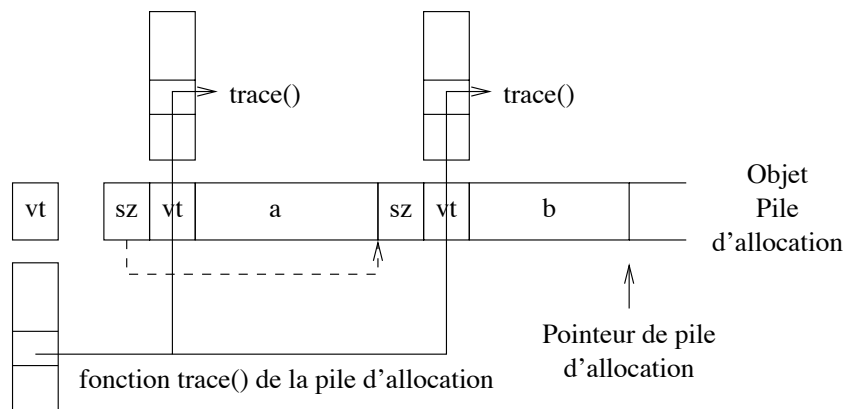


FIG. 6.9 – Collection de la pile

L'utilisation d'une pile externe est meilleure pour les collections que l'utilisation de la pile d'exécution pour allouer les objets. En effet, la fonction de collection du ramasse-miettes n'est pas capable d'utiliser les tables virtuelles des objets dans la pile d'exécution et parcourrait toutes les cases mémoires dans la pile. Avec l'algorithme présenté, seuls les sous-objets sont parcourus.

3.4 Performances de l'analyse d'échappement

Le but de l'analyse d'échappement est d'augmenter les performances de l'application. Les tests ont été effectués sur un PowerPC avec un Linux (Gentoo) cadencé à 1Ghz . Pour tester l'impact de l'analyse d'échappement, nous avons exécuté la section 3 du bench Java-grande [EPP03] avec et sans analyse d'échappement. La JnJVM est exécutée avec le GNU Classpath [gnu] (voir la section 4 du chapitre 5).

3.4.1 Objets capturés

Le tableau 6.10 présente le nombre d’objets qui s’échappent dans le GNU Classpath et dans la section trois du Javagrande. L’outil d’analyse statique ne peut pas savoir ce que font les méthodes natives. C’est pourquoi, dès qu’une méthode native est appelée et reçoit un objet, aucune assertion ne peut être faite sur l’échappement de cet objet. Dans la colonne “sans natives”, les méthodes natives sont considérées comme non fiables (i.e. tout objet transmis à une méthode native s’échappe) et dans la colonne “avec natives”, les méthodes natives sont considérées comme fiables. Pour les tests, les méthodes natives sont considérées comme non fiables.

Programme	Nombre d'allocations capturées		Total
	sans natives	avec natives	
GNU ClassPath	543 (4%)	800 (6%)	13226
JavaGrande Forum	363 (46%)	386 (49%)	774
-> Euler	18 (31%)	18 (31%)	57
-> MolDyn	3 (33%)	3 (33%)	9
-> Montecarlo	4 (4%)	6 (6%)	104
-> Raytracer	11 (19%)	11 (19%)	57
-> Search	7 (24%)	7 (24%)	29

FIG. 6.10 – Analyse d'échappement statique

3.4.2 Performances en allocation

Description des résultats. La figure 6.11 récapitule les mesures observées pour les tests *Search*, *MonteCarlo* et *Euler*. Les deux autres tests de la section 3 du JavaGrande ne sont pas présentés car ils n'apportent pas d'informations supplémentaires. Pour chaque test, deux séries de mesures ont été effectuées en variant le pas de collection (colonnes 512ko et 4096ko). Le pas de collection correspond aux nombres d'octets alloués avant de commencer une nouvelle collection⁶. Ces deux mesures montrent l'impact du matériel sur l'analyse d'échappement. En effet, plus les ressources en terme de mémoire sont faibles, plus le pas de collection doit être petit de manière à éviter l'activation du fichier d'échange (swap) et plus l'allocation en pile accélère les performances. Le choix des deux valeurs (512ko et 4096ko) dépend de la machine : un pas de collection de 4096ko est tout à fait adéquat sur la machine de test qui possède 1Go de mémoire et le pas de 512ko est idéal pour une autre machine de test moins puissante (un PowerPC à 366MHz et 128Mo de mémoire).

Deux versions de l'outil d'analyse statique d'analyse d'échappement ont été développées. La première marque tous les objets capturés par l'outil (colonne **boucles gardées**) alors que la seconde ne marque que les objets qui se trouvent en dehors de boucles (colonne **boucles retirées**). L'intérêt de cette seconde version est discuté en même temps que le test *Euler* (voir paragraphes suivants). La colonne **normal** donne les mesures observées avec la JnJVM sans la MVLet à analyse d'échappement.

En ligne, la figure 6.11 donne différentes mesures observées. La ligne **Collection** indique le nombre de collections effectuées pendant le test (le pourcentage est relatif au nombre de collection sans analyse d'échappement). Les lignes **alloués en tas** et **alloués en pile** indiquent la quantité de mémoire alloué dans le tas et dans la pile d'allocation. Les pourcentages sont relatifs aux octets alloués en tas de la colonne **normal**. La ligne **Résultats/seconde** indique les résultats du test et mesure la vitesse d'exécution de l'application. Chacune de ses mesures a été effectuée trois fois. Le pourcentage est relatif à la colonne **normal**, c'est à dire que ce pourcentage indique l'accélération obtenue avec la MVLet d'analyse d'échappement. L'écart type est celui observé sur les trois mesures précédentes.

⁶Le pas de collection est l'inverse de la fréquence de collection.

	Search					
	512ko			4096ko		
	normal	boucles gardées	boucles retirées	normal	boucles gardées	boucles retirées
Collections	619	5 (0,9%)	5 (0,9%)	80	3 (3,8%)	3 (3,8%)
Octets alloués en tas	327Mo	5Mo (1,5%)	5Mo (1,5%)	327Mo	5Mo (1,5%)	5Mo (1,5%)
Octets alloués en pile		322Mo (98,5%)	322Mo (98,5%)		322Mo (98,5%)	322Mo (98,5%)
Taille maximale de pile		1772 octets	1772 octets		1772 octets	1772 octets
1000 Résultats/seconde	97,1	120,5 (+24,12%)	120,3 (+23,88%)	115,6	119,7 (+ 3.62%)	120,0 (+ 3.86%)
Écart type	0,07%	0,15%	0,54%	0,20%	0,12%	0,08%
	MonteCarlo					
	512ko			4096ko		
	normal	boucles gardées	boucles retirées	normal	boucles gardées	boucles retirées
Collections	576	575 (100%)	575 (100%)	73	73 (100%)	73 (100%)
Octets alloués en tas	292Mo	292Mo (100%)	292Mo (100%)	292Mo	292Mo (100%)	292Mo (100%)
Octets alloués en pile		0,1Mo (0%)	0,1Mo (0%)		0,1Mo (0%)	0,1Mo (0%)
Taille maximale de pile		8 octets	8 octets		8 octets	8 octets
10 ⁻³ Résultats/seconde	2.89	2.90 (+0.4%)	2.90 (+0.4%)	3.10	3.13 (+ 1.22%)	3.13 (+ 1.00%)
Écart type	0,06%	0,03%	0,07%	0,14%	0,14%	0,25%
	Euler					
	512ko			4096ko		
	normal	boucles gardées	boucles retirées	normal	boucles gardées	boucles retirées
Collections	1616	940 (58%)	877 (54%)	203	119 (59%)	111 (55%)
Octets alloués en tas	842Mo	488Mo (58%)	455Mo (53%)	842Mo	488Mo (58%)	455Mo (53%)
Octets alloués en pile		354Mo (42%)	387Mo (47%)		354Mo (42%)	387Mo (47%)
Taille maximale de pile		1Mo	444 octets		1Mo	444 octets
10 ⁻⁴ Résultats/seconde	7.34	7.74 (+5.61%)	7.83 (+6.66%)	8.15	8.28 (+ 1.60%)	8.30 (+ 1.82%)
Écart type	0,05%	0,07%	0,03%	0,07%	0,03%	0,00%

FIG. 6.11 – Impact de l'analyse d'échappement

Search. Le test **Search** est celui qui donne les meilleurs gains avec la MVLet à analyse d'échappement. Pratiquement tous les objets alloués pendant ce test sont capturés et mis en pile et le nombre d'octets présents à un instant donné dans la pile d'allocation est très faible (1772 octets). Les résultats montrent que 99% (resp. 96%) des collections sont évitées avec une fréquence de collection de 512ko (resp. 4Mo). L'impact de l'analyse d'échappement augmente lorsque le pas de collection diminue (ce qui équivaut à une machine à faibles ressources mémoire) : l'application est accélérée jusqu'à 24%. Cet impact est beaucoup plus faible avec une machine avec de fortes ressources matérielles : le gain avec un pas de collection de 4Mo n'est plus que de 3.7%. Une série de mesure a été effectuée avec un pas de collection de 128ko et montre alors une accélération de l'ordre de 80%.

Ces résultats mettent en avant deux propriétés importantes de l'analyse d'échappement :

- Dans le meilleur des cas, tous les objets d'une application peuvent être alloués dans la pile d'allocation avec une taille de pile assez faible (quelques ko). L'usage du ramasse-miettes peut donc être totalement éliminé.
- L'impact de l'analyse d'échappement dépend fortement du matériel sous-jacent : plus la quantité de mémoire est grande, plus la fréquence de collection peut être grande et plus l'accélération obtenue avec la MVLet d'analyse d'échappement est faible.

MonteCarlo. Le test **MonteCarlo** est un exemple où l'analyse d'échappement ne modifie pas les performances de l'application. Ce type de cas se présente lorsque les objets capturés sont des objets d'initialisation : les objets alloués pendant l'exécution de l'application ne sont pas capturés et sont donc alloués via l'allocateur objet normal. Malgré quelques allocations en pile, l'analyse d'échappement ne fait pas économiser une seule collection au test **MonteCarlo**. Ce test met en avant une nouvelle propriété : l'analyse d'échappement n'est pas forcément adéquat pour toute application mais dans ce cas, elle ne dégrade pas les performances.

Euler. Le test **Euler** est le seul test où les colonnes **boucles gardées** et **boucles retirées** indiquent des résultats différents. Lorsque les objets capturés alloués dans des boucles sans gardés (i.e ils sont marqués capturés), la pile d'allocation peut arriver à saturation et contenir jusqu'à 1Mo d'objets. La figure 6.12 présente un exemple qui sature la pile d'allocation. Les objets **BadEscape** sont alloués en pile (car capturés) et ne peuvent être libérés qu'à la fin de la méthode **f()**. La pile d'allocation est donc rapidement saturée et à chaque collection mémoire, la pile est parcourue ce qui ralentit le temps d'exécution. Dans le test **Euler** avec boucles gardées, le ralentissement n'apparaît pas directement car 42% des objets alloués le sont dans la pile d'allocation (soit 354Mo) ce qui accélèrent plus l'application que ce que la pile saturée la diminue.

Pour éviter de ralentir les applications qui font de nombreuses allocations capturées dans des boucles, une autre version de l'outil d'analyse statique d'échappement a été développée de manière à éliminer les objets capturés dans des boucles de la liste des objets capturés. Avec cette version, les objets capturés alloués dans les boucles ne sont plus marqués par l'outil d'analyse statique et la pile n'est plus saturée (la taille maximale de la pile n'est plus que de 444 octets dans la colonne **boucles retirées**) ce qui évite de ralentir l'application. L'accélération passe alors de 5.61% à 6.66% avec un pas de collection de 512ko et de 1.60% à 1.82% avec un pas de 4096ko.

Ce test met en avant une nouvelle propriété de l'analyse d'échappement : si la pile d'allocation est saturée, les performances de l'application peuvent être dégradées. Comme la pile

possède une borne supérieure, l'impact d'une pile saturée reste toutefois limitée. En éliminant les objets capturés alloués dans les boucles, l'outil d'analyse d'échappement statique évite la plupart des cas où la pile d'allocation peut être saturée. Toutefois, il est possible de saturer la pile d'allocation sans utiliser de boucle (par exemple, avec une méthode récursive) et il reste donc de rares cas où l'allocation en pile peut dégrader les performances de l'application.

```

1  class BadEscape {
2      static String cst = "Chaine";
3      String str;
4
5      BadEscape() { str = cst; }
6      public void f() {
7          for(int i=0; i<1000000; i++)
8              new BadEscape();
9      }
10 };

```

FIG. 6.12 – Une fonction qui ralentit la JVM à analyse d'échappement

3.4.3 Élimination de synchronisations

L'élimination de synchronisations a été mesurée en appelant la méthode synchronisée `append()` 200 fois sur un objet `StringBuffer` capturé. Cette opération est répétée 100000 fois pour obtenir des résultats significatifs. Le temps d'exécution sans enlever les synchronisations est de 64.9s contre 27.7s en retirant les synchronisations, soit un gain de 57%. Ce test montre un très bon gain, mais ce résultat est à mettre en parallèle avec les mauvaises performances des appels de méthodes synchronisées (voir la section 4.5 du chapitre 5). Avec des appels synchronisés de meilleure qualité, les gains seraient beaucoup plus faibles.

3.4.4 Impact des modifications

L'impact des modifications apportées à la JnJVM par la MVLet d'analyse d'échappement a été mesuré dans un test. Cette expérience, réalisée sur la même plate-forme avec le test Search sans attributs `EscapeMap` montre que l'exécution de la JnJVM est de 93.25s et celle de la JnJVM avec analyse d'échappement est de 93.3s. Ce test ne mesure que l'impact des modifications apportées par la MVLet d'analyse d'échappement : aucune allocation en pile n'est effectuée. La différence de temps d'exécution entre les deux tests n'est pas significative ce qui montre que les modifications dues à la MVLet d'analyse d'échappement n'ont pas de conséquences visibles sur les temps d'exécution. Le même test a été réalisé sur le démarrage de la JnJVM, avec et sans analyse d'échappement, et les différences observées sont aussi inexistantes.

3.4.5 Récapitulatif

Ces tests montrent plusieurs propriétés de l'analyse d'échappement :

- L'analyse d'échappement est un mécanisme qui peut accélérer la vitesse d'exécution d'une application. En éliminant les objets capturés dans des boucles, l'analyse d'échappement ne présente que des gains.
- L'analyse d'échappement n'est pas forcément adéquate pour toute application puisque l'allocation en pile peut ne pas modifier la vitesse d'exécution.
- Lorsque l'analyse d'échappement est en adéquation avec le travail réalisé par une application, les performances peuvent être nettement augmentée (jusqu'à 24% avec le bench standard JavaGrande).
- Les interceptions de fonctions du compilateur de bytecode pour pouvoir gérer l'analyse d'échappement n'ont pas d'impact sur les performances.
- L'élimination de synchronisations ne présente que des gains qui peuvent aller jusqu'à 57%.

3.5 Conclusion

La MVLet d'analyse d'échappement illustre la flexibilité de la JnJVM et les possibilités de l'architecture des applications actives. Comme l'application modifie l'environnement en fonction de ses besoins, l'application peut spécialiser des mécanismes internes de la machine virtuelle Java tout en restant portable. La taille de la MVLet d'analyse d'échappement est relativement petite, 508 lignes de codes, et sa présence ne change pas les performances de la machine virtuelle. Ce code modifie un aspect de la JnJVM dynamiquement : la gestion mémoire. La technique présentée dans cette section peut être utilisée pour déployer d'autres technologies, par exemple, le Proof Carrying Code [Nec97, RR98] (PCC) pour accélérer la phase de vérification de bytecode dans la machine virtuelle Java.

Sur l'analyse d'échappement elle-même, les expériences montrent que l'impact dépend fortement de l'application : les applications qui allouent de nombreux objets en dehors de boucles bénéficient d'une accélération pouvant aller jusqu'à 24% et les applications qui appellent de nombreuses méthodes synchronisées sur des objets capturés peuvent espérer être accélérées jusqu'à 57%. L'analyse d'échappement est donc un outil pertinent particulièrement adapté aux applications ayant un usage intensif de la mémoire ou de méthodes synchronisées.

4 Tissage dynamique d'aspects

Comme présenté dans la section 3 du chapitre 2, le tissage dynamique d'aspects est un problème difficile à résoudre dans le monde Java. La gestion du tissage d'aspects peut être effectué soit par une surcouche logicielle, soit par une JVM dédiée.

Une surcouche logicielle, comme dans JBoss AOP [jbo], AspectWerkz [Bon04] ou Jac [Objb, jaca], engendre des pertes de performances et reste limitée en terme d'adaptabilité dynamique puisque seuls les points de jonction définis pendant le chargement de classe peuvent être exploités. L'avantage de l'approche surcouche logicielle est la portabilité. En effet, toute JVM implantant les classes de réflexion Java peut exécuter un programme aspectisé avec l'un des trois outils.

La seconde solution pour palier à ces deux problèmes, performance et manque d'adaptabilité dynamique, est de gérer le tissage d'aspects directement dans la machine virtuelle Java, comme dans Steamloom [BHMO04]. L'inconvénient de l'approche est la perte de compatibilité avec l'existant.

Notre solution est d'intégrer directement le tissage d'aspects au cœur de la machine virtuelle Java, mais en laissant à l'application le soin de construire son propre environnement dédié de manière à maintenir la compatibilité avec l'existant. La JnJVM est donc enrichie par une MVLet de tissage dynamique d'aspects qui offre :

- une grande flexibilité, comme dans Steamloom, en créant des points de jonction dynamiquement ;
- de bonnes performances puisqu'il n'y a plus de points de jonction sur lesquels aucun aspect n'est tissé et puisque les appels au “code advice”⁷ ne nécessitent aucune indirection ;
- une intégration dans l'existant beaucoup plus rapide : c'est l'application active qui spécifie quels sont ses besoins.

Nous avons effectué deux séries de travaux autour des aspects dans la JnJVM :

- la réalisation d'une MVLet de tissage dynamique d'aspects ;
- la réalisation d'une MVLet d'intégration de la JnJVM avec un outil de modélisation pour adapter l'exécution au modèle au fur et à mesure des changements.

Ces deux MVLets montrent comment transformer la JnJVM en machine virtuelle Java dédiée au tissage dynamique d'aspects et présente un exemple d'application utilisant ce tissage.

4.1 La MVLet aspect

La MVLet aspect offre un langage de haut niveau pour manipuler les points de jonction et l'association entre points de jonction et “code advice”. Cette MVLet illustre les possibilités d'introspection et d'intercession au niveau applicatif dans la JnJVM. La MVLet aspect encapsule des appels aux fonctions internes de la μvm pour offrir des abstractions de plus haut niveau.

4.1.1 Le langage

Le langage utilisé pour tisser des aspects est un des langages que la μvm peut prendre en entrée (actuellement XML ou μvm). La MVLet aspect offre une interface pour rechercher des classes, des champs et des méthodes, dupliquer des méthodes et tisser des aspects sur des méthodes ou des champs.

Le tissage d'aspects sur les champs est coûteux : chaque méthode accédant à un champs doit être stockée dans une structure de donnée associée au champs pour pouvoir retrouver tous les sites de consultation ou de modification de ce champs et régénérer le code de ces sites. Pour ce faire, la MVLet de tissage d'aspects modifie la manière dont les bytecode d'accès aux champs sont compilés (`getfield`, `setfield`, `getstatic` et `setstatic`) pour préserver la méthode contenant le site d'appel dans la description de champs. Cette manipulation peut être coûteuse en mémoire et impose de charger la MVLet aspect avant de commencer à compiler des méthodes Java. Le tissage d'aspects sur les champs est donc optionnel. La définition de points de jonction sur les exceptions est actuellement en projet.

Bien souvent, lorsqu'un nouvel aspect est tissé, il faut pouvoir ajouter de nouvelles méthodes ou de nouveaux champs aux classes. Comme expliqué dans la section 3.1 du chapitre 5, l'ajout de champs est lourd à implanter et n'a pas été effectué. En revanche, l'ajout de méthodes a été implanté et est assez léger.

⁷Le code implantant l'aspect.

4.1.2 Un exemple de tissage d'aspects

```

1 (define .c ( :aspect.get-class (jvm) "Hello"))
2 (define .m ( :aspect.get-method c "m" "(I)V" :aspect.ACC_VIRTUAL))
3 (define .o ( :aspect.dup-method m " m-orig"))
4 (define .fct
5   (lambda (cache value this)
6     (let ([meth ( :aspect.deref-cache cache)])
7       ( :system.printf " ; call : %s\n" ( :object.print-string meth))
8       ( :aspect.invoke-virtual o value this)
9       ( :system.printf " ; end of : %s\n" ( :object.print-string meth))))))
10
11 ( :aspect.wave-on-method m fct)

```

FIG. 6.13 – Un exemple de tissage d'aspects

La figure 6.13 présente un exemple complet de tissage d'aspects. Tout d'abord, la classe `Hello` est placée dans le symbole `c` à la ligne 1. Ensuite la méthode d'instance `void m(int)` est cherchée et mise dans le symbole `m` (ligne 2). Une copie de cette méthode est effectuée pour pouvoir l'appeler une fois le nouvel aspect tissé (ligne 3). La fonction `fct` est le “code advice” : elle affiche simplement quelle méthode est appelée et appelle la méthode originale. Ensuite l'aspect est tissé à la ligne 11.

L'appel à `:aspect.dup-method` charge automatiquement la classe de la méthode : il faut connaître la méthode originale pour la copier. La fonction `:aspect.wave-on-method` s'occupe de mettre à jour les caches en ligne pour que la nouvelle fonction soit prise en compte. Une fonction `:aspect.wave-on-method-with-bytecode` permet aussi de tisser directement un aspect dans une méthode Java à partir du bytecode d'une méthode Java.

4.1.3 Conclusion

La MVLet aspect fournit des abstractions de plus haut niveau pour exprimer le tissage d'aspects mais n'offre pas encore le haut niveau des outils commerciaux comme AspectJ ou JBoss AOP. En particulier, une syntaxe permettant d'appeler l'équivalent du `proceed` de Jac (voir la section 3.2.3 du chapitre 2) est en projet et une réflexion sur l'ordre d'appel des “codes advices” pour le tissage multiple et sur le dé-tissage est menée.

La MVLet aspect utilise les mécanismes internes de la μvm pour générer et remplacer le code. Le résultat est donc optimal puisque aucune indirection n'est nécessaire pour atteindre le nouveau code. Une série de mesures de performances a été effectuée et montre que le tissage d'aspects dans la machine virtuelle Java n'a pas d'impact. Toute application active contenant la MVLet aspect spécialise bien la JnJVM avec des outils de tissage d'aspect, tout en restant portable : les avantages des environnements dédiés et des surcouches logicielles sont bien réunis.

4.2 Intégration de la JnJVM avec un outil de modélisation

Les outils de modélisation permettent de décrire la structure d'un programme à l'aide d'outils graphiques et de générer le squelette des applications pour une plate-forme particulière. Lorsque le modèle ou les algorithmes d'une application change, il faut redémarrer l'application pour prendre en compte ces changements. Pour éviter ce redémarrage pour des applications critiques, nous proposons d'utiliser la JnJVM, la MVLet XML et la MVLet de tissage d'aspects pour refléter les changements apportés au code sans interruption du service.

L'outil de modélisation utilisé est Objecteering [obja]. Des travaux ont été menés par X. Blanc pour que cet outil génère des fichiers XML indiquant les changements apportés au modèle pour la plate-forme Java. La MVLet développée pendant ces travaux, appelée MVLet Objecteering, prend en entrée ce fichier et tisse les nouveaux aspects dynamiquement dans la JnJVM. La MVLet Objecteering s'occupe simplement de traduire les balises XML produites par Objecteering en appels aux fonctions de la MVLet aspects. Cette MVLet peut aussi être vue comme une feuille de style spécifique aux fichiers XML d'Objecteering.

Le fichier XML contient principalement la classe sur laquelle on souhaite ajouter ou modifier des méthodes, la signature de ces méthodes et le code en Java des méthodes. Il est laissé à la charge de la MVLet Objecteering de générer le fichier binaire Java. L'architecture de la JnJVM complique l'ajout dynamique de champs (voir la section 3.1 du chapitre 5) : cette possibilité n'a pas été développée faute de temps. Actuellement, seules des méthodes peuvent être ajoutées, retirées ou modifiées.

Cette génération passe par une introspection de la classe : un squelette vide de la classe est généré en excluant les méthodes modifiées. Ensuite les méthodes modifiées et les nouvelles méthodes sont ajoutées au squelette. Le squelette est alors envoyé au compilateur Java standard de la machine.

Le chargement de la nouvelle classe est plus délicat. En effet, deux "constant pool" différents⁸ doivent être maintenus en mémoire. Le premier est utilisé par les méthodes de la classe d'origine qui n'ont pas été modifiées et le second par les nouvelles méthodes ajoutées ou modifiées. De manière à maintenir n versions d'une même classe, une seconde machine virtuelle Java est instanciée (le composant machine virtuelle Java décrit dans la section 3.1 du chapitre 5). Celle-ci sert à préserver les différentes versions d'une même classe.

La fonction de chargement de classes de cette machine virtuelle Java s'occupe de charger les nouveaux bytecodes. Ces nouveaux bytecodes sont insérés par remplacement ou ajout dans la classe d'origine à l'aide de la MVLet de tissage d'aspects. Les méthodes modifiées référencent alors la nouvelle classe et le compilateur utilise le bon "constant pool".

La figure 6.14 décrit graphiquement le fonctionnement de la nouvelle JVM (appelée JVM n-version). Les méthodes sont partagées entre les deux descriptions de classes. Les méthodes d'origine référencent toujours la classe de la JnJVM alors que les méthodes ajoutées ou modifiées référencent la classe chargée dans la JVM n-version. Les champs sont aussi partagés entre les deux versions de la classe.

4.3 Conclusion

Ces travaux sur le tissage d'aspects montrent comment ajouter dynamiquement un tisseur dynamique d'aspects applicatif dans la JnJVM et une utilisation de ce tisseur d'aspects

⁸Le tableau indexant les symboles Java utilisés par une classe.

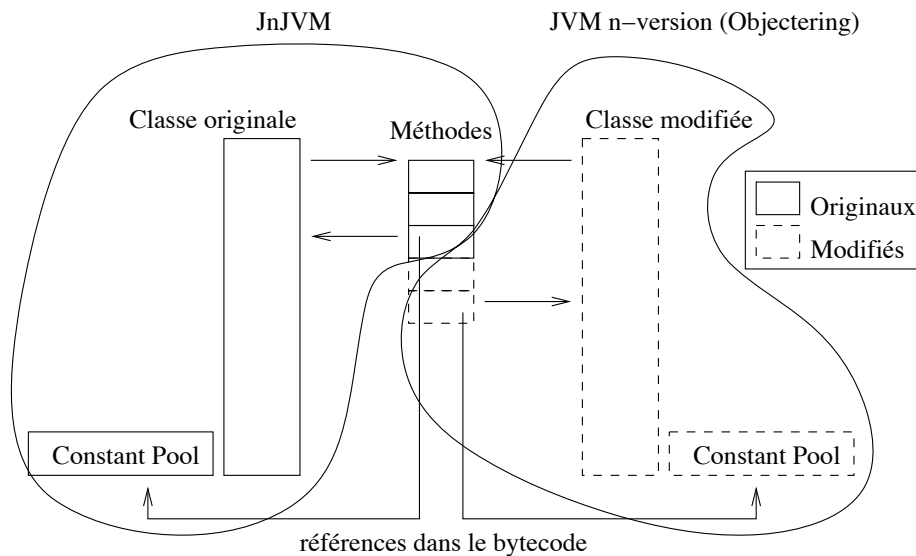


FIG. 6.14 – Structure des classes modifiées

pour coupler l'exécution d'une application à son modèle. Les applications actives permettent à l'application de construire un environnement spécialisé dans le tissage d'aspects à partir d'une machine virtuelle standard, sans avoir à toucher au code de la machine virtuelle. Les applications actives apportent une solution au problème de l'évolutivité logicielle en évitant les environnements dédiés et en enrichissant l'existant.

5 La JVM à migration de fil d'exécution

La migration d'un fil d'exécution consiste à prendre une tâche s'exécutant sur une machine A et à la déplacer, sans interruption de service, sur une machine B. La migration vise soit à augmenter la disponibilité en répliquant des applications sur un réseau, soit à équilibrer la charge des machines en déplaçant les applications en fonction de la charge des machines sur un réseau [SM01].

Les solutions actuelles permettant de faire de la migration sont de deux natures différentes :

- migration système : c'est l'environnement qui prend en charge la migration ;
- migration applicative : c'est l'application qui contient dans son code les algorithmes permettant de migrer l'application.

Aucune des deux solutions n'est satisfaisante. La migration système pose le problème des ressources locales : la liaison entre l'application et les ressources de sa machine ne peut pas être migrée de manière générique. En effet, les fichiers ouverts, les sockets ou les fenêtres graphiques sont dépendants de l'application, du système et du matériel utilisé et le système de migration ne peut faire aucune hypothèse sur l'état dans lequel doivent se trouver ces liaisons matérielles sur la machine cible. Ce point cantonne donc les systèmes de migration à des applications ayant très peu d'interactions avec les ressources, comme les processus de calcul.

La gestion de la migration au niveau applicatif permet de palier au problème des ressources. En effet l'application sait parfaitement ce qu'elle peut faire de ses fichiers, de ses fenêtres ou ses sockets ouvertes, et peut, en conséquence, prendre les décisions adéquates pendant une phase de migration. Ce type de migration pose toutefois deux problèmes : (i) il n'y a pas de séparation entre code fonctionnel et code non-fonctionnel, (ii) si l'environnement n'offre pas de mécanismes d'introspection suffisants, le fil d'exécution ne peut pas être déplacé. Ce problème est particulièrement vrai en Java car il n'y a pas de mécanismes d'introspection de la pile d'exécution.

Nous proposons, à l'aide des applications actives, de garder les bonnes propriétés des deux approches et de résoudre les problèmes soulevés. Notre solution consiste à laisser l'environnement prendre en charge la migration de fil d'exécution, mais en laissant à l'application active la charge de spécialiser la migration en fonction de son environnement. Notre approche sépare bien le code fonctionnel du code non fonctionnel : le programme Java n'a pas besoin d'être écrit spécifiquement pour pouvoir être migré. Le contexte peut être migré en utilisant la migration spécifique et on est sûr que l'environnement offre les bon mécanismes de migration puisque c'est l'application qui les injecte dans la JnJVM.

Notre MVLet de migration se sépare donc en deux parties :

- Une MVLet de migration générique qui implante les mécanismes de migration système (déplacement de la mémoire et du fil d'exécution).
- Une MVLet de migration spécialisée en fonction de l'application qui enrichit la précédente pour migrer le contexte d'exécution de l'application. Cette MVLet peut aussi prendre en charge une adaptation de l'application en fonction de la plate-forme cible.

Ces travaux ont été partiellement réalisés par C. Mescam pendant son stage de DEA [Mes04] encadré par B. Folliot et moi-même, et ont été complétés par la suite par moi-même.

5.1 Travaux similaires

Les machines virtuelles Java offrent des mécanismes de migration d'objet grâce à l'interface `java.io.Serialize`. Les objets implantant cette interface peuvent être sérialisés sous la forme d'un flux et reconstruits de l'autre côté. En revanche, les JVM ne permettent pas de migrer les fils d'exécution : aucune information n'est disponible sur le bytecode en cours d'exécution (le compteur de programme) ou sur les valeurs des variables locales.

Deux approches principales permettent de migrer les fils d'exécution Java :

- L'approche machine virtuelle dédiée, comme MOBA [SM01] : c'est la JVM qui prend en charge la migration.
- L'approche préprocesseur : un préprocesseur s'occupe d'injecter le code de migration directement à partir du source [Fun98, SMY99] ou du binaire [SSY00, TRV⁺00] du programme qui va devoir être migré. Le code injecté permet de capturer l'état des méthodes en cours d'exécution sur la pile.

Les systèmes à base de préprocesseur offrent l'avantage d'être portables sur toute machine virtuelle Java, mais le code injecté (sous forme de gestion d'exception ou de code spécifique autour des invocations) ralentit le temps d'exécution de l'application. L'approche machine virtuelle Java dédiée présente les défauts des environnements ad-hocs : ils sont incompatibles avec l'existant. Une autre approche a été proposée à l'Inria par S. Bouchenak et D. Hagimont [BH02] sans surcharge du code et sans machine virtuelle dédiée. Toutefois, cette approche repose sur une fonction de la version 1.3.1 de la machine virtuelle de

Sun n'appartenant pas aux spécifications, ce qui rend le code incompatible pour d'autres machines virtuelles Java.

Les machines virtuelles dédiées à la migration et les systèmes à base de préprocesseur n'offrent pas de mécanismes particuliers pour migrer les liens avec les ressources locales de la machine source (sockets, fichiers ouverts, interface graphique etc...). Ces limites sont dues à l'approche : un système de migration transparente ne peut pas choisir pour l'application ce qu'il faut faire des liens avec les ressources locales.

5.2 Migration générique

La migration générique repose sur les mécanismes d'introspection de la JnJVM : les classes sont déplacées vers la machine cible avec leur état. Migrer l'état d'une classe consiste à migrer la partie statique de la classe.

Pour les valeurs primitives, (entiers, flottants etc..), les nombres sont sérialisés dans le format réseau. Les objets sérialisables sont migrés via des `ObjectOutputStream` : les mécanismes existants sont réutilisés. Les objets non sérialisables ne peuvent pas être migrés par cette MVLet car ce sont des objets qui dépendent des ressources utilisées par l'application. Dès qu'un objet non sérialisable est trouvé, la MVLet cherche une méthode d'instance de sérialisation, appelée `serializeSpecific(int)`, dans la liste des méthodes de la classe de l'objet. Si cette méthode n'est pas trouvée, une erreur est levée, sinon, cette fonction est appelée avec un descripteur DVVM (voir section 2.1) pour pouvoir faire communiquer la machine source avec la machine cible. L'implantation de ces méthodes `serializeSpecific(int)` est laissée à la charge de la MVLet de spécialisation. La MVLet de migration spécialisée peut utiliser la MVLet de tissage d'aspects pour créer ces méthodes de sérialisation.

La migration du contexte d'exécution passe par deux phases :

- introspection de la pile d'exécution et migration vers la cible ;
- redémarrage de la pile d'exécution sur la cible.

Toutes les variables locales sont envoyées vers la cible avec la méthode et l'index du bytecode Java en cours d'exécution. Chez la cible, les méthodes sont redémarrées au même point en compilant la méthode à redémarrer de manière spécifique : la mise à jour des variables locales est effectuée au début de la méthode et un saut vers le bytecode en cours d'exécution est généré. Une boucle permet de recommencer la même opération sur chaque méthode de la pile d'exécution en partant du sommet de la pile. Cette partie de la MVLet n'est pas achevée faute de temps. Les deux mécanismes restant à implanter sont la compilation spécifique pour redémarrer les méthodes et l'introspection des types des variables locales pour différencier les valeurs primitives des références d'objets⁹. De plus, la migration du fil d'exécution ne peut pas prendre en charge les appels aux fonctions natives : il est laissé à la charge de la MVLet de migration spécifique de ne pas avoir d'appels à des fonctions natives sur la pile d'exécution.

La MVLet de migration générique ne fait aucune assertion sur l'instant de démarrage de la migration : c'est en effet un aspect spécifique de la migration. Une fois les tâches migrées, elles ne sont pas automatiquement redémarrées, en effet, l'instant de redémarrage est spécifique (voir la section suivante).

⁹Malgré l'utilisation de l'allocateur présenté dans la section 3.1.2 du chapitre 3, on n'est pas à l'abri d'une collision entre une valeur et une référence.

5.3 Migration d'un client Jabber

La migration d'un client de messagerie Jabber illustre la MVLet générique de migration et donne un exemple de migration spécifique. Le problème est de déplacer un client de messagerie d'une machine X vers une machine Y sans interruption de service. Cet exemple a été développé dans l'optique de faire passer un fil de discussion d'un ordinateur de bureau vers un pda.

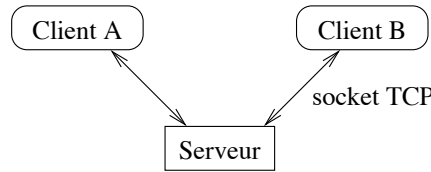


FIG. 6.15 – Architecture du protocole Jabber

L'architecture du protocole repose sur un serveur centralisé et des clients connectés au serveur. Lorsqu'un client *A* arrive sur le réseau, il se connecte au serveur qui crée un thread de communication avec lui. Ensuite, le client *A* peut contacter le client *B* en envoyant des messages au serveur par cette socket. Le serveur s'occupe alors de renvoyer les messages au client *B* en passant par le thread de communication de *B*.

De son côté, le client possède deux threads : un thread de lecture sur le clavier qui envoie les messages au serveur et un thread de réception qui s'occupe d'afficher sur l'écran les messages adressés au client.

Le serveur, écrit en C, et le client, écrit en Java, ont été développés par C. Mescam pendant son stage. Aucune réflexion n'a été menée à ce niveau pour faciliter la migration. En revanche, la connaissance de l'architecture de l'application est nécessaire pour pouvoir écrire la MVLet de migration spécifique.

Le serveur doit être adapté pour permettre la migration : il n'existe pas de solution pour reconnecter une socket TCP après la migration et les messages envoyés au client pendant la migration ne peuvent pas être traités. Le code du serveur n'est pas modifié, en revanche l'édition finale de lien du serveur redirige les appels aux fonctions réseaux vers une bibliothèque qui sait prendre en charge la migration chez le serveur. Cette redirection est effectuée avec des appels au programme standard de copie de fichiers exécutables (`objcopy`) qui permet de renommer des symboles de fichiers un binaire. L'ensemble des symboles redéfinis forme la coupe de l'aspect réseau du serveur Jabber : un nouvel aspect est tissé statiquement sur le binaire du serveur.

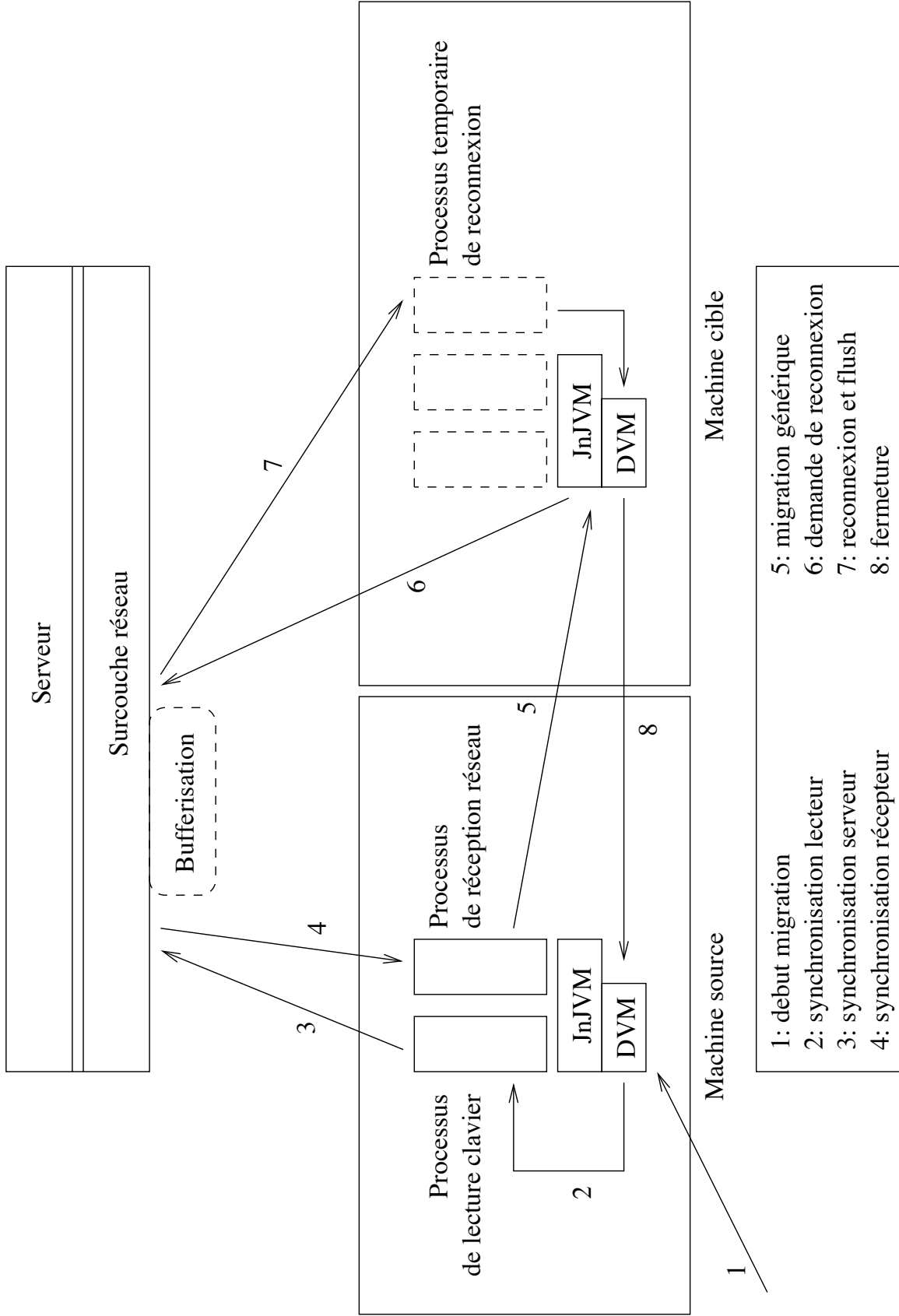


FIG. 6.16 – Protocole de migration

La figure 6.16 présente le protocole. L'utilisateur qui choisit de migrer son application envoie un ordre via la DVVM au client pour démarrer le protocole (1). Il faut noter que les MVLet de migration peuvent être chargés dynamiquement à ce moment dans la JnJVM. Le client commence par figer son thread de lecture (2) : l'entrée standard est redirigée vers un tube et un message d'arrêt de lecture est envoyé au thread de lecture. Cette manipulation permet de mettre le client dans un état cohérent pour la migration puisque plus rien n'est exécuté dans une fonction native.

Le thread de lecture envoie alors un message au serveur (3) pour qu'il traite le client de manière spécifique : les messages à destination du client sont mis dans un tampon pendant la phase de migration et seront émis au client migré après la migration. Le serveur envoie alors un message au client (4) pour qu'il démarre sa migration. Ces deux messages passent par la socket de communication applicative entre le client et le serveur : le protocole est enrichi par la MVLet de migration spécifique chez le client et par la surcouche logicielle chez le serveur pour gérer de manière spécifique ces messages. La MVLet aspect est utilisé pour modifier les méthodes de réception. De la même façon, la méthode de lecture clavier chez le client est déroutée pour prendre en compte le message de synchronisation du thread de lecture (2).

La mémoire et les threads de réception et d'émission du client sont dans un état cohérent (pas de fonctions natives) et peuvent migrer (5) vers la cible en utilisant la migration générique. Il faut noter que la cible doit ne posséder au minimum qu'une DVVM : la JnJVM ainsi que la MVLet de migration générique peuvent être envoyées dynamiquement à la cible en même temps.

Un thread temporaire de reconnexion est créé chez la cible. Un message (6) est envoyé au serveur pour qu'il se reconnecte au client via ce thread. La migration de la socket se fait en créant de manière temporaire une socket d'écoute dans le thread de reconnexion. Celle-ci est connectée par le serveur (7) pour régénérer une nouvelle socket de communication. Dès cette connexion effectuée, les données du tampon du serveur sont envoyées au client. Le client ne possède pas de fichiers ouverts. Seules les entrées/sorties standards ont besoin d'être migrées, elles sont recrées chez la cible normalement.

Un fois la reconnexion réussie, le serveur ferme l'ancienne socket et la nouvelle est dupliquée (appel à `dup`) à la place de l'ancienne pour rendre ces changements transparents au serveur. Le processus temporaire de reconnexion débloque les deux autres processus sur la cible, envoie un message à la source (8) via la DVVM pour qu'elle quitte et le thread de reconnexion se termine. A partir de ce point, le protocole reprend de manière normale.

La partie migration générique étant toujours en cours de développement, aucune mesure de performances ne peut être présentée.

5.4 Conclusion

Cet exemple de migration montre comment utiliser les applications actives pour offrir des mécanismes système spécialisés en fonction de l'application (migration spécialisée) et en fonction des besoins système (migration générique). Ces travaux ne sont pas tout à fait achevés mais illustrent les avantages qu'une application peut tirer de l'approche Machine Virtuelle Virtuelle. L'application peut à la fois implanter des mécanismes génériques dans la JnJVM et spécialiser ceux-ci en fonction de ses besoins spécifiques.

Ces travaux ouvrent une perspective intéressante : l'adaptation de l'application, pendant la migration, en fonction de la plate-forme cible. En reprenant l'idée de départ d'un

client qui se déplace d'un ordinateur de bureau vers un pda, il semble intéressant d'adapter l'application aux nouvelles ressources matérielles mises à sa disposition.

Cet exemple de migration montre aussi que l'application n'est pas seule en cause pour réussir à réaliser de tels protocoles : le mécanisme de migration mis en place fait intervenir une deuxième entité, le serveur, qui doit être capable de prendre en compte la migration. L'intégration des applications actives avec l'existant demande donc la modification des clients, mais aussi des serveurs.

6 Récapitulatif

Dans ce chapitre quatre séries de travaux illustrant les possibilités des applications actives dans la μvm et dans la JnJVM ont été présentées :

- la modification du lexeur et du parseur de la μvm pour exécuter du code à partir de serveurs distants et à partir de fichiers XML ;
- la modification de la JnJVM pour optimiser la gestion mémoire et la gestion des synchronisations à partir d'informations d'analyse d'échappement ;
- une étude du tissage d'aspects applicatif dans la JnJVM ;
- une étude de la migration dans la JnJVM.

Cette évaluation qualitative prouve le haut degré de flexibilité de l'approche : que ce soit la μvm , les MVlets chargées ou les parties passives des applications actives, chaque partie peut être adaptée par l'application pour que l'environnement corresponde mieux aux attentes de l'application.

Les différentes MVlets présentées dans ce chapitre montrent que des mécanismes dédiés ne sont pas en contradiction avec un environnement générique en utilisant une architecture application active. Les applications actives permettent de répondre à de réels problèmes, tissage d'aspects, migration, analyse d'échappement, tout en profitant d'une homogénéité logicielle.

Le point faible que nous ont montré ces expériences est la nécessité de maîtriser parfaitement le code de la JnJVM et de la μvm . Ce point est assez critique : seuls des spécialistes de la JnJVM peuvent conceptualiser et mettre en œuvre des MVlets performantes pour la JnJVM. Toutefois, l'étude présentée dans ce chapitre montre qu'une fois qu'une MVlet de spécialisation est développée, elle peut être réutilisée par toute application active. De la même façon, une rustine (un patch) pour une application est pratiquement toujours développée par l'équipe qui a développé l'application. Une fois cette rustine déployée, les utilisateurs de l'application n'ont pas à se soucier du contenu de la rustine, mais uniquement des nouvelles fonctionnalités qu'elle apporte.

La figure 6.17 récapitule toutes les MVlets présentées dans cette thèse. Les MVlets entourées de pointillés ont juste été évoquées et ne sont pas réalisées (Proof Carrying Code, Détection de Fautes). Chaque MVlet adapte la MVV construite par le chargement de la MVlet précédente. Il n'y a aucune contradiction à utiliser simultanément plusieurs MVlets orthogonales (ce qui est le cas de toutes les MVlets présentées dans cette thèse). Toutefois, il peut exister des points de conflit entre MVlets : actuellement, seuls les concepteurs de MVlets savent si leurs modifications peuvent avoir des effets de bord sur d'autres spécialisations. Un travail doit maintenant être mené pour formaliser et résoudre ce problème.

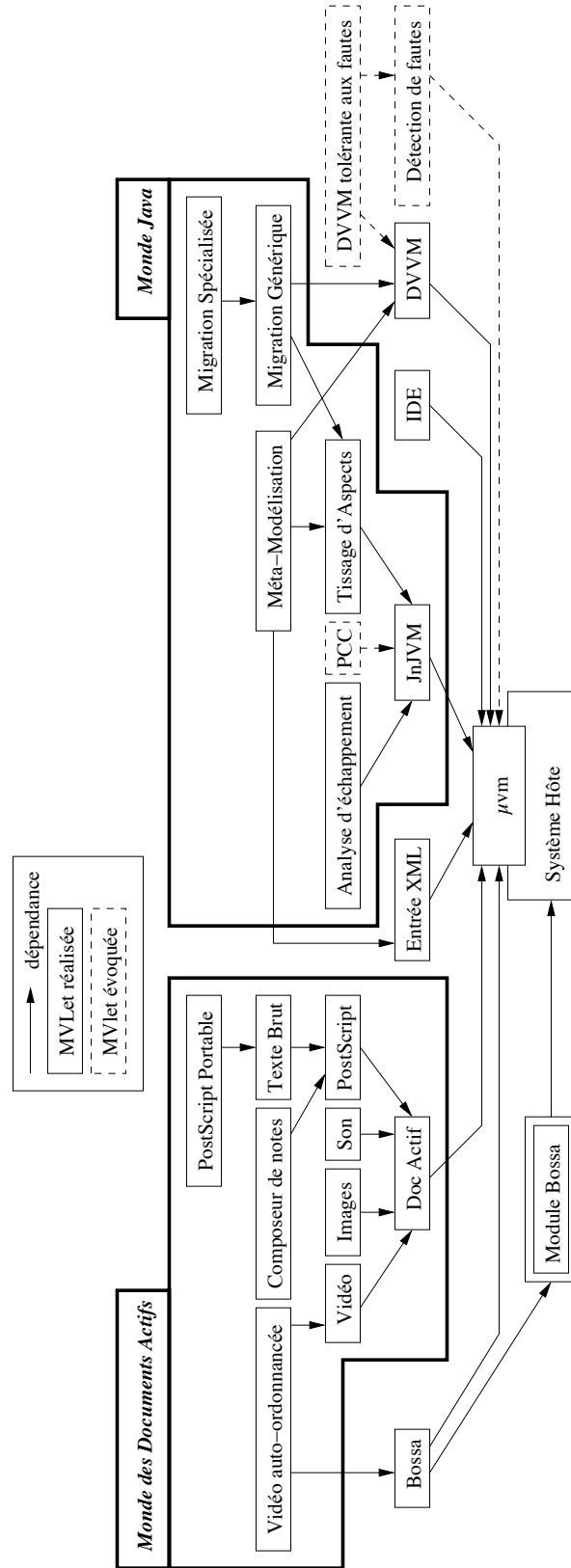


FIG. 6.17 – Récapitulatif de nos travaux sur les applications actives

CHAPITRE 7

Conclusions et perspectives

1 Les applications actives, la μvm , les documents actifs et la JnJVM

Dans cette thèse, nous avons introduit la notion d'applications actives, un cadre général permettant à une application de construire, modifier et spécialiser dynamiquement son environnement d'exécution pour qu'il corresponde à ses besoins. Les applications actives apportent une réponse au foisonnement d'environnement dédiés :

- Elles accélèrent l'intégration de nouveaux mécanismes dans les environnements d'exécution standards ;
- Elles diminuent l'hétérogénéité logicielle en offrant des applications portables ;
- Elles permettent une grande réutilisabilité du code en factorisant le développement de mécanismes dédiés ;
- Elles offrent les mêmes performances et les mêmes possibilités que les environnements dédiés équivalents.

La μvm est le support d'exécution des application actives. C'est un compilateur dynamique réflexif et minimal. Il offre à la fois les mécanismes permettant de construire un environnement d'exécution, mais aussi les outils pour adapter dynamiquement cet environnement pendant l'exécution de l'application active. Le haut degré d'adaptation de la μvm repose sur un protocole méta-objet (MOP) permettant aux applications actives de modifier la μvm et de construire de nouveaux environnements d'exécution adaptables. Le MOP n'ajoute pas d'indirection dans le code et offre les mêmes performances que les appels virtuels en C++. En revanche, la présence de méta-données maintenues par le MOP augmente la mémoire utilisée par la μvm , ce qui exclue pour l'instant une utilisation de la μvm dans des environnements trop limités en terme de mémoire.

L'architecture des applications actives est validée par un prototype de documents actifs et un prototype de machine virtuelle Java.

L'étude des documents actifs montre que les applications actives sont bien adaptées pour résoudre les problèmes de qualité de service, de composition de documents multimédia, de droits d'auteur et de cohérence dans les documents. En ajoutant du code actif à un docu-

ment, le document devient un programme exécutable, ce qui lève les limites traditionnelles des documents passifs (qui ne contiennent pas de code).

La JnJVM , le prototype de machine virtuelle Java, est un environnement d'exécution Java flexible et spécialisable par une application active. Celui-ci offre des performances équivalentes à la première machine virtuelle Java à compilateur en ligne (JIT) de Sun Microsystems (HotSpot 1.0) datant de 1995. Ce résultat est dû à un manque d'optimisation du prototype et non à l'architecture adoptée. En effet, avec un meilleur compilateur dynamique, une optimisation du ramasse-miettes et une gestion efficace de la synchronisation et des exceptions, la JnJVM peut théoriquement atteindre les mêmes performances que les machines virtuelles Java de Sun Microsystems ou d'IBM. Les mesures effectuées sur la JnJVM montrent donc qu'il n'y a pas de contradiction entre de bonnes performances et l'approche adoptée.

La grande originalité de la JnJVM par rapport à d'autres machines virtuelles Java est l'utilisation du protocole méta-objet (MOP) hérité de la μvm . Ce protocole permet en effet à une application active de spécialiser dynamiquement la JnJVM en fonction de ses besoins. L'application active reste alors portable et contient le code de spécialisation de la JnJVM. Les applications actives ne diminuent pas pour autant le travail de développement : la JnJVM fait 15000 lignes de code, ce qui est équivalent à toute autre machine virtuelle Java.

2 Les applications d'évaluation

L'évaluation qualitative de la JnJVM montre différentes possibilités d'adaptation de l'environnement d'exécution par l'application. Les MVLets présentées adaptent l'application, mais aussi la μvm et la JnJVM. La vision uniforme de la mémoire objet permet à ces MVLets d'utiliser un seul paradigme : il n'y a pas de rupture entre l'application et son environnement d'exécution.

Les études présentées dans le chapitre 6 répondent à des problèmes récurrents en recherche informatique, à savoir optimiser la gestion mémoire et la gestion des synchronisations avec la MVLet à analyse d'échappement, offrir un cadre pour l'adaptation de l'application avec la MVLet aspects, et apporter une réponse aux problèmes de la migration de fil d'exécution avec la MVLet de migration. Toutes ces MVLets transforment l'environnement en fonction des besoins de l'application, sans engendrer de surcoût à l'exécution. Ces travaux mettent en avant le haut degré d'adaptabilité des composants réflexifs de la μvm et montrent qu'une application active peut s'exécuter dans un environnement générique et le spécialiser tout en restant portable.

Le langage d'entrée de la μvm est modifié par deux MVLets : les MVLet XML et DVVM. La MVLet XML permet à la μvm de recevoir des programmes sous la forme de fichiers XML et la MVLet DVVM de recevoir des programmes sous une forme compact adaptée à la migration de code sur un réseau. Ces deux MVLets montrent que la chaîne de compilation de la μvm peut être réutilisée par une application active pour étendre ou modifier le langage d'entrée.

La MVLet à analyse d'échappement optimise la gestion mémoire et la gestion de la synchronisation. Les mesures de performances de la MVLet à analyse d'échappement

montrent que l'exploitation de l'algorithme d'analyse d'échappement n'est pas adéquat pour toute application : le bilan des tests effectués est positif lorsque l'application utilise de nombreux objets à vie courte alloués dans de nombreuses méthodes, et nul lorsque l'application alloue de nombreux objets dans peu de méthodes. Ces tests confirment donc que les mécanismes mis en place au niveau système ne sont pas forcément adéquats pour toutes les applications.

Les gains observés avec l'analyse d'échappement augmentent lorsque la fréquence de collection diminue, ce qui montre que plus la quantité de mémoire disponible est faible, plus l'exploitation des informations d'échappement devient intéressante. En particulier, utiliser l'analyse d'échappement dans du matériel embarqué, type carte à puce ou PDA, semble être une voie prometteuse.

La MVLet à analyse d'échappement utilise la réflexion des composants héritée de la μvm pour tisser un aspect sur la gestion mémoire et la gestion de la synchronisation. À l'aide du MOP défini dans la μvm , des aspects peuvent être tissés en tout point de la JnJVM, ce qui permet à une application active de modifier ou d'étendre le comportement de tout l'environnement d'exécution.

La MVLet aspect présente un cadre pour tisser des aspects dans la JnJVM. Cette MVLet montre qu'il est possible de concilier les aspects avec de bonnes performances et une haute dynamique en intégrant le tissage d'aspects au cœur de l'environnement d'exécution, comme avec Steamloom [BHMO04], et qu'il est possible d'utiliser ce mécanisme dédié sans avoir à construire un nouvel environnement incompatible avec l'existant. La MVLet à migration de fil d'exécution utilise la MVLet aspect et montre que le tissage d'aspects dynamique apporte des solutions efficaces aux besoins en adaptation de l'application. Le couplage de la JnJVM et de la MVLet aspects avec l'outil de modélisation Objecteering [obja] permet de répercuter à l'exécution, sans interruption de service, les modifications apportées au modèle de l'application. Ce couplage entre un outil de conception et l'exécution d'une application permet l'automatisation d'une grande partie de la mise à jour dynamique d'une application.

La MVLet à migration montre comment utiliser les applications actives pour implanter un moteur de migration au niveau système, sans être confronté au défaut de cette approche : la gestion de la migration des ressources locales utilisées par l'application. En scindant notre MVLet en deux parties, générique et spécialisée, nous implantons les mécanismes de migration de fil d'exécution au cœur de l'environnement tout en offrant la possibilité de gérer les liens entre l'application et ses ressources. De plus, le prototype développé montre qu'il est possible d'injecter les mécanismes de migration après le déploiement de l'application : il n'y a pas de rupture non plus entre configuration et reconfiguration.

3 Perspectives

Les perspectives à très court terme sont d'achever la MVLet à migration de fil d'exécution et de mesurer ses performances. Outre ces perspectives à très court terme, quatre axes principaux sont envisagés :

- sécuriser la μvm au niveau langage et plate-forme ;
- optimiser la μvm en vitesse et en taille mémoire ;

- décrire avec un plus haut niveau les points d'adaptation d'une MVLet pour accélérer le développement de MVLets dédiés ;
- adapter automatiquement la JnJVM en fonction des performances de l'application.

Sécurité Les applications actives laissent le code actif construire l'environnement. Cette approche est performante et portable, mais pose de nombreux problèmes de sécurité : comme l'application contient du code système, il faut que l'environnement d'exécution puisse non seulement se protéger du code applicatif (comme dans tout environnement d'exécution), mais aussi du code actif.

Le code actif, constitué de MVLets, est exécuté par la μvm : c'est donc à elle d'offrir les mécanismes suffisants pour vérifier que le code effectue des actions régulières. Une première ébauche du travail à réaliser provient d'une étude des mécanismes de sécurité dans les machines virtuelles Java. Les différents points recensés sont :

- Renforcer le typage dans la μvm pour vérifier, pendant la compilation, la validité des affectations de champs et des appels de méthodes des composants.
- Introduire un système de protection donnant des droits sur les différents éléments d'un composant (accès public, privé, en modification). Ces droits peuvent aussi dépendre du degré de confiance accordé au code (par exemple, en interrogeant un tiers de confiance).
- Isoler les méthodes dangereuses avec des gestionnaires de sécurité donnant des droits plus ou moins grands en fonction du degré de confiance. De la même façon, les mots clés du langage peuvent aussi être isolés.

Ces trois premières esquisses de solution permettraient d'assurer au chargement de MVLet le même niveau de sûreté que celui assuré par le chargement d'une application Java dans une JVM. Il faut toutefois noter que pour augmenter la sécurité dans la μvm , il faut obligatoirement diminuer le degré d'adaptabilité. Ce point ne pose pas de problème si les MVLets possèdent des certificats de confiance : plus le degré de confiance dans une MVLet est grand, plus la sécurité peut diminuer et plus elle peut adapter une MVV. De la même façon, les mises à jour automatiques comme le propose Windows XP dépendent du degré de confiance attribué à Microsoft.

Travaux autour de l'optimisation Les optimisations en vitesse de la μvm et de la JnJVM permettraient à celles-ci d'offrir des performances équivalentes aux machines virtuelles Java d'IBM et de Sun Microsystems et montrerait clairement que l'architecture des applications actives n'entraîne pas de perte de performances. Les différentes optimisations envisagées à ce niveau sont :

- implanter les flottants (32 et 64 bits) et entiers sur 64 bits dans le compilateur en ligne de la μvm (la VPU) ;
- optimiser l'allocation de registre de manière à optimiser les accès aux variables locales les plus fréquemment utilisées dans la VPU ;
- mettre en ligne les méthodes courtes dans l'appelant pour éviter des appels ;
- ajouter des instructions de synchronisation fondamentales dans la VPU (comme un test-and-set) pour implanter, sans indirection, des fonctions de synchronisation ;
- ajouter des instructions pour préserver le contexte d'exécution (les registres) dans la VPU de manière à avoir une gestion efficace et portable des exceptions ;
- implanter l'algorithme de vieillissement des objets décrit dans la section 3.1.4 du chapitre 3 directement dans la VPU pour optimiser le ramasse-miettes générationnel.

Ces optimisations de la compilation demanderaient un grand travail d'ingénierie et n'apporteraient pas de nouveauté sur le plan scientifique, mais seraient utiles pour une utilisation à plus grande échelle de la μvm et de la JnJVM. Outre ces optimisations de bas niveau, un travail de recherche peut être réalisé pour générer du code spécifique en fonction du matériel pour profiter des spécificités de la plate-forme.

L'optimisation de la taille de la mémoire de la μvm et de la taille du code généré par la μvm ouvrirait quand à lui de nombreuses perspectives, en particulier pour du matériel à forte contrainte mémoire comme les cartes à puces. L'architecture des applications actives peut, par exemple, être utilisée pour mettre à jour à distance les cartes à puces de décodeurs de chaînes câblées en fonction de l'évolution des demandes des utilisateurs.

Description de points d'adaptation Les points d'adaptation de la μvm et des MVLets sont directement liés au code : modifier un comportement revient à modifier une entrée dans un vecteur de fonctions et il faut donc connaître le nom de la méthode pour réaliser ce travail. Connaître le nom d'une méthode réalisant un travail demande de connaître le code de la MVLet qui doit être adaptée. Par exemple, pour réaliser la MVLet d'analyse d'échappement, il faut savoir que la méthode qui s'occupe de générer l'entête à la fin d'une méthode s'appelle

```
:jnjvm.compile.in-java-compiler.
```

A moins de lire le code (ou d'interroger le développeur), il n'y a aucun moyen de retrouver ce nom.

En offrant un langage de plus haut niveau, du type courtier (trader) dans les intergiciels, permettant de retrouver un nom à partir d'une description de haut niveau, il deviendrait envisageable de séparer la conception de MVLets de spécialisation du code spécialisé.

Auto-adaptation Les travaux sur l'analyse d'échappement montrent que l'utilisation des informations d'analyse d'échappement ne sont pas forcément adéquats pour toutes applications. Malgré l'élimination de boucles, l'analyse d'échappement pourrait tout de même diminuer les performances de l'application. Par exemple, une fonction récursive allouant des objets capturés finirait par saturer la pile d'allocation et dégraderait les performances de l'application (voir la section 3.4 du chapitre 6).

Il semble donc intéressant d'étudier l'adéquation du mécanisme système mis en œuvre avec les résultats attendus : si les performances de l'application sont dégradées avec la MVLet d'analyse d'échappement, alors il est plus efficace d'allouer les objets capturés qui saturer la pile avec l'allocateur du ramasse-miettes ou de supprimer la MVLet d'analyse d'échappement pour remettre la JnJVM dans son état initial. Pour résoudre ce problème, il faut placer la JnJVM à l'analyse d'échappement dans un état d'observation à chaque fois que la pile d'allocation est saturée, trouver quelles sont les allocations qui saturer la pile et recompiler les fonctions qui effectuent ces allocations en éliminant l'allocation en pile.

De manière plus générale, un mécanisme système peut ne pas être adéquat. En intégrant de l'observation, il devient possible de quantifier l'impact d'un mécanisme système dans la JnJVM et de pouvoir choisir un autre mécanisme. Cet axe de recherche ouvre de nombreuses perspectives et pose de nouveaux challenges : il faut pouvoir quantifier la dégradation, observer sans ralentir les performances et choisir le bon algorithme en fonction de l'application ou de son contexte d'utilisation. L'architecture des applications actives apporterait alors une solution pour adapter automatiquement les applications dans le cadre de l'informatique

autonome (autonomous computing [ibm]).

Bibliographie

- [ABE00] A. Andersen, G. Blair, and F. Eliassen. Oopp : A reflective component-based middleware. In *Norsk Informatikk Konferance*, 2000.
- [AFD⁺00] N. Achir, M. Fonseca, Y. Doudane, N. Agoulmine, and A. Mehaoua. Active networking system evaluation : A practical experience. In *The 7th International Workshop on Mobile Multimedia Communications*, Tokyo, JAPAN, 2000.
- [AHK⁺98] D. Alexander, M. Hicks, P. Kakkar, A. Keromytis, M. Shaw, J. Moore, C. Gunter, S. Nettles, and J. Smith. The switchware active network implementation. In *ACM SIGPLAN Workshop on ML*, 1998. cite-seer.ist.psu.edu/article/alexander98switchware.html.
- [AL91] A.W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 26, pages 96–107, New York, NY, 1991. ACM Press.
- [aMS99] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *ACM Symposium on Operating Systems Principles*, pages 48–63, 1999.
- [AP03] H. Azatchi and E. Petrank. Integrating generations with advanced reference counting garbage collectors. In *12th International Conference on Compiler Construction (CC 2003)*, Warsaw, Poland, April 2003.
- [BBC02] L. Baduel, F. Baude, and D. Caromel. Efficient, flexible, and typed group communications in java. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 28–36, Seattle, 2002. ACM Press. ISBN 1-58113-559-8.
- [BBD⁺00] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull, R. Belliardi, D. Locke, S. Robbins, P. Solanki, and D. de Niz. *The Real-Time Specification for Java*. Addison-Wesley, 2000. www.rtsj.org/rtsj-V1.0.pdf.
- [BCC⁺99] G.S. Blair, F.M. Costa, G. Coulson, H.A. Duran, N. Parlavantzas, F. Delpiano, B. Dumant, F. Horn, and J-B. Stefani. The design of a resource-aware reflective middleware architecture. In *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection, Reflection'99*, pages 115–134, Saint Malo, France, July 1999.
- [BCS02] E. Bruneton, T. Coupaye, and J-B. Stefani. Recursive and dynamic software composition with sharing. In *7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga, Spain, 06 2002.
- [BDS91] H-J. Boehm, A.J. Demers, and S. Shenker. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, volume 26, pages 157–164, 1991.

- [BH02] S. Bouchenak and D. Hagimont. Zero overhead java thread migration. Rapport Technique 0261, Inria, May 2002.
- [BHMO04] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *third International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 83–92, Lancaster, UK, March 2004.
- [Bla03] B. Blanchet. Escape analysis for java : Theory and practice. 25(6) :713–775, 2003.
- [BLBF02] R. Braden, B. Lindell, S. Berson, and T. Faber. The asp ee : An active network execution environment. In *2002 DARPA Active Networks Conference and Exposition (DANCE'02)*, page 238, San Francisco, May 2002.
- [Blo00] J. Blosser. Explore the dynamic proxy api. Technical report, Sun Microsystems, Nov. 2000. <http://java.sun.com/developer/technicalArticles/DataTypes/proxy/>.
- [BM00] L. Bölöni and D.C. Marinescu. *Intelligent Systems and Interfaces*, chapter An Object-Oriented Framework for Building Collaborative Network Agents, pages 31–64. Kluwer Publishing House, 2000.
- [Boe] H-J. Boehm. Two-level tree structure for fast pointer lookup. http://www.hpl.hp.com/personal/Hans_Boehm/gc/tree.html.
- [Bon04] J. Bonér. Aspectwerkz – dynamic aop for java. http://codehaus.org/jboner/papers/aosd2004_aspectwerkz.pdf, 2004.
- [BR00] E. Bruneton and M. Riveill. Javapod : une plate-forme à composants adaptable et extensibles. Rapport technique 3850, Inria Rhone-Alpes, 2000.
- [Cal99] K.L. Calvert. Architectural framework for active networks. draft version 1.0, Active Network Working Group, jully 27 1999. <http://www.dcs.uky.edu/calvert/arch-latest.ps>.
- [CF89] A. Cox and R. Fowler. The implementation of a coherent memory abstraction on a numa multiprocessor : experiences with. In *ACM Symposium on Operating Systems Principles*, pages 32–44, 1989.
- [CFKL96] P. Cao, E.W. Felten, A.R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4) :311–343, 1996.
- [CFL94] P. Cao, D.W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Operating Systems Design and Implementation*, pages 165–177, 1994.
- [CGS⁺99] J-D. Choi, M. Gupta, M.J. Serrano, V.C. Sreedhar, and S.P. Midkiff. Escape analysis for java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.
- [Chi98] S. Chiba. Javassist — a reflection-based programming wizard for java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Oct. 1998.
- [Cle04] C. Clement. Ordonnancement adaptable : Mvv-bossa. Master's thesis, Université Pierre et Marie Curie – Dea Systèmes Informatiques Répartis, Sept. 2004.

- [CN03] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proceedings of the second international conference on Generative programming and component engineering table (GPCE '03)*, pages 364 – 376, Erfurt, Germany, 2003.
- [CWHT] M. Campione, K. Walrath, A. Huml, and Tutuorial Team. *The Java Tutorial Continued : The Rest of the JDK*. Sun Microsystems.
- [Dan00] J. Daniel. *Au coeur de Corba*. 2000.
- [DGK⁺02] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, and E. Petrank. Thread-local heaps for java. In *Proceedings of the third international symposium on Memory management*, pages 76–87, Berlin, Germany, 2002.
- [DH98] S. Deering and R. Hinden. Internet protocol, version 6 (ipv6) specification. <ftp://ftp.isi.edu/in-notes/rfc2460.txt>, december 1998. Network Working Group.
- [DLM⁺78] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection : an exercise in cooperation. *Communication of the ACM*, 21(11) :966–975, sept 1978.
- [DM95] F. Demers and J. Malenfant. Reflection in logic, functional and object-oriented programming : a short comparative study. In *Workshop of IJCAI'95 on Reflection and Meta-Level Architectures and their Application in A.I.*, pages 29–38, Aug. 1995.
- [DPP⁺99] D.S. Decasper, B. Plattner, G.M. Parulkar, S. Choi, J. DeHart, T. Wolf, and B. Plattner. A scalable high-performance active network node. *IEEE Network*, 13(1) :8–19, Jan/Feb 1999.
- [DvZH] M. Dahn, J. van Zyl, and E. Haase. *BCEL – The ByteCode Engineneering Library – Manual*. <http://jakarta.apache.org/bcel/manual.html>.
- [EKO95] D.R. Engler, M.F. Kaashoek, and J. O'Toole. Exokernel : An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.
- [Ell99] C.S. Ellis. The case for higher-level power management. In *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, page 162, 1999.
- [EPP03] Edinburgh Parallel Computing Center EPPC. Java grande forum benchmark suite – version 2.0. <http://www.epcc.ed.ac.uk/javagrande/>, 2003.
- [EvOS96] A. Eliëns, J. van Ossenbruggen, and B. Schönhage. Animating the web – an sgml-based approach. In *Proceedings of the International Conference on 3D and Multimedia on the Internet, WWW and networks*, april 1996.
- [FBM⁺96] N. Freed, N. Borenstein, K. Moore, J. Klensin, and J. Postel. Multipurpose internet mail extensions mime. <http://www.mhonarc.org/ehood/MIME/>, 1996. rfc2045, rfc2046, rfc2047, rfc2048, rfc2049.
- [FPR97] B. Folliot, I. Piumarta, and F. Riccardi. Virtual Virtual Machines. *Cabernet Radical Workshop*, 1997.
- [FPR98] B. Folliot, I. Piumarta, and F. Riccardi. A dynamically configurable, multi-language execution platform. In *8th ACM SIGOPS European Workshop*, 1998.

- [FSLM02] J-P. Fassino, J-B. Stefani, J. Lawall, and G. Muller. Think : A software framework for component-based operating system kernels. In *USENIX Annual Technical Conference*, Monterey, CA, USA, June 2002.
- [FT01] B. Folliot and G. Thomas. Protocole de membership hautement extensible : conception est expérimentations. In *Actes de la 2ème Conférence Française sur les Systèmes d'Exploitation (CFSE'02), Chapitre français de l'ACM-SIGOPS, GDR ARP*, pages 25–36, April 2001.
- [Fun98] S. Funrocken. Transparent migration of java-based mobile agents. In *Mobile Agents*, pages 26–37, 1998.
- [Gal05] A. Galland. *Contrôle de ressources dans les cartes à microprocesseur*. PhD thesis, Université Pierre et Marie Curie, Paris, France, 2005.
- [Geo04] N. Geoffray. Conception et réalisation d'une interface xml pour la machine virtuelle virtuelle. Technical report, Université Pierre et Marie Curie – Stage de deuxième année de Magistère d'Informatique Appliqué d'Ile de France, Sept. 2004. [http ://www-src.lip6.fr/homepages/Gael.Thomas/](http://www-src.lip6.fr/homepages/Gael.Thomas/).
- [GFS98] T. Givens, O. Farmer, and Q. Smith. Active networks, redefining the nature of computer networks. [http ://www.cc.gatech.edu/classes/AY2000/cs6250_fall/l44.ppt](http://www.cc.gatech.edu/classes/AY2000/cs6250_fall/l44.ppt), December 1998. Cours du Georgia Institute of Technology.
- [GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification (2nd Edition)*. 2000.
- [GL03] J.D. Gradecki and N. Lesiecki. *Mastering AspectJ : Aspect-Oriented Programming in Java*. 2003.
- [gnu] The GNU Classpath Project. www.gnu.org/software/classpath/classpath.html.
- [Gnu00] The gnutella protocol specification v0.4 – document revision 1.2, 2000. [http ://www9.limewire.com/developer/gnutella_protocol_0.4.pdf](http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf).
- [Gol97] M. Golm. Design and implementation of a meta architecture for java. Master's thesis, University of Erlangen, Jan 1997.
- [Gol98] M. Golm. metaxa and the future of reflection. In *OOPSLA Workshop on Reflective Programming in C++ and Java*, Vancouver, British Columbia, Oct. 1998.
- [Gri00] G. Grimaud. *Camille : un Système d'exploitation ouvert pour carte à microprocesseurs*. PhD thesis, Laboratoire Fondamentale de Lille (LIFL), 2000.
- [Har01] T. Harpin. Using java.lang.reflect.proxy to interpose on java class methods, class methods. Technical report, Sun Microsystems, July 2001. [http ://java.sun.com/developer/technicalArticles/JavaLP/Interposing/](http://java.sun.com/developer/technicalArticles/JavaLP/Interposing/).
- [HB78] Jr. H.G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4) :280–294, 1978.
- [HBG⁺98] F.J. Hauck, U. Becker, M. Geier, E. Meier, U. Rasthofer, and M. Steckermeier. AspectIX : An aspect-oriented and CORBA-compliant ORB architecture. Technical Report TR-I4-98-08, Univ. of Erlangen-Nuernberg, IMMD IV, 1998.

- [HC92] K. Harty and D.R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27, pages 187–197, New York, NY, 1992. ACM Press.
- [HKM⁺98] M.W. Hicks, P. Kakkar, J.T. Moore, C.A. Gunter, and S. Nettles. PLAN : A packet language for active networks. In *International Conference on Functional Programming*, pages 86–93, 1998.
- [HMT⁺04] A. Hachichi, C. Martin, G. Thomas, S. Patarin, and B. Folliot. Reconfigurations dynamiques de services dans un intergiciel à composants corba ccm. In *1ère Conférence Francophone sur le Déploiement et la (Re) Configuration de Logiciels (DECOR'04)*, Grenoble, France, october 2004. To appear.
- [HSM98] I. Hadzic, J.M Smith, and W.S. Marcus. On-the-fly programmable hardware for networks. In *Globecom'98*, 1998.
- [ibm] Autonomic computing. <http://www.research.ibm.com/autonomic/index.html>. by IBM research.
- [IW99] R.J. Stroud I. Welch. From dalang to kava — the evolution of a reflective java extension. In *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, Saint-Malo, France, July 1999.
- [JA99] M. Jordan and M. Atkinson. Orthogonal persistence for the java platform – draft specification. Technical report, Sun Microsystems Laboratories, Oct. 1999. <http://research.sun.com/research/forest/COM.Sun.Labs.Forest.doc.opjspec.abs.html>.
- [jaca] <http://jac.aopsys.com/>. <http://jac.aopsys.com/>.
- [jacb] Jacorb web site. <http://www.jacorb.org>.
- [jav] Evolving performance of the java virtual machine. <http://www-106.ibm.com/developerworks/java/library/j-berry/sidebar.html>. in the IBM virtual Machine.
- [jbo] Jboss aop home page. <http://www.jboss.org/products/aop>.
- [JBY00] K. Jun, L. Boloni, and D. Yau. Intelligent qos support for an adaptive video server. In *proceedings of the Information ResourcesManagement Association International Conference (IRMA 2000)*, pages 1096–1098, Anchorage Alaska, May 2000.
- [jik] Jikes – research virtual machine home (rvm) page. <http://www-124.ibm.com/developerworks/oss/jikesrvm>.
- [jit02] Performance comparison of jits, Jan. 2002. <http://www.shudo.net/jit/perf/>.
- [KCM⁺00] F. Kon, R.H. Campbell, M.D. Mickunas, K. Nahrstedt, and F.J. Ballesteros. 2k : A distributed operating system for dynamic heterogeneous environments. In *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, page 201, Pittsburgh, Pennsylvania, August 2000.
- [KLM⁺97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

- [KLVA93] K. Krueger, D. Loftesness, A. Vahdat, and T. Anderson. Tools for the development of application-specific virtual memory management. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 48–64, 1993.
- [KRL⁺00] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L.C. Magalhães, and R.H. Campbell. Monitoring, security, and dynamic configuration with the dynamic-tao reflective orb. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.
- [LCC97] P.C.H. Lee, R.-C. Chang, and M.C. Chen. Hipec : A system for application-customized virtual-memory caching management. *Software : Practice and Experience*, 27(5) :547–571, 1997.
- [LH83] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6) :419–429, 1983.
- [Lia99] S. Liang. *The Java Native Interface*. June 1999.
- [LMB⁺96] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7) :1280–1297, 1996.
- [LMB02] J. Lawall, G. Muller, and L.P. Barreto. Capturing os expertise in an event type system : the bossa experience. In *Tenth ACM SIGOPS European Workshop 2002 (EW2002)*, pages 54–61, St. Emilion, France, sep 2002.
- [LY] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Paperback.
- [Mae87a] P. Maes. Computational reflection. Master's thesis, Vrije Universiteit, Artificial Intelligence Laboratory, Brussels, Belgium, 1987.
- [Mae87b] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA '87*, Orlando, Floride, USA, 1987.
- [Mar03] C. Martin. Outils et méthodes pour la construction de systèmes adaptables. Master's thesis, Université Pierre et Marie Curie – DEA Systèmes Informatiques Répartis, Sept. 2003. <http://www-src.lip6.fr/homepages/Gael.Thomas/>.
- [MBZC00] S. Merugu, S. Bhattacharjee, E.W. Zegura, and K.L. Calvert. Bowman : A node OS for active networks. In *INFOCOM (3)*, pages 1127–1136, 2000.
- [Mes04] C. Mescam. La migration avec la jnvm. Master's thesis, Université Pierre et Marie Curie – Dea Systèmes Informatiques Répartis, Sept. 2004.
- [MGNS94] M. Makpangou, Y. Gourhant, J-P. Le Narzul, and M. Shapiro. *Fragmented objects for distributed abstractions*, pages 170–186. IEEE Computer Society Press, 1994.
- [MNR⁺98] S. Michel, K. Nguyen, A. Rosenstein, H. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching : towards a new global caching architecture. *Computer Networks and ISDN Systems*, 30(22-23) :2169–2177, November 1998.

- [mpi] The message passing interface (mpi) standard. <http://www-unix.mcs.anl.gov/mpi>.
- [MRC⁺00] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil : An idl for hardware programming. In *Proceedings of the USENIX 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, Oct 2000.
- [MVB01] C. Marchetti, L. Verde, and R. Baldoni. Corba request portable interceptors : a performance analysis. In *3rd International Symposium on Distributed Objects and Applications*, Rome, Italy, September 2001.
- [Nec97] G.C. Necula. Proof-carrying code. *Conference Record of POPL'97 : The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, jan 1997. Paris, France.
- [OB99] A. Oliva and L.E. Buzato. The design and implementation of guaraná. In *5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)*, San Diego, California, USA, May 1999.
- [obja] Objecteering home page. <http://www.objecteering.com/>.
- [Objb] ObjectWeb, http://jac.objectweb.org/docs/programmer_guide.html. *Aspect-Oriented Programming with JAC*.
- [Obj01] Object Managment Group. *Interceptors Published Draft with Corba 2.4+ Core Chapters*, 2001. Document Number ptc/2001-03-04.
- [Obj02] Object Managment Group. *Corba Components, 3.0 Specification*, June 2002. formal/02-06-65.
- [Obj04] Object Managment group. *Common Object Request Broker Architecture : Core Specification, 3.0.3 Specification*, March 2004. formal/04-03-01.
- [OFT04] F. Ogel, B. Folliot, and G. Thomas. A step toward ubiquitous computing : An efficient flexible micro-orb. In *proceedings of the 11th ACM SIGOPS European Workshop*, pages 176–181, Leuven, Belgium, sept. 2004.
- [OGB98a] A. Oliva, I.C. Garcia, and L.E Buzato. Guaraná : a tutorial. Technical report ic-98-31, Univerisidade Estadual de Campinas, Sept. 1998.
- [OGB98b] A. Oliva, I.C. Garcia, and L.E Buzato. The implementation of guaraná on java. Technical report ic-98-32, Univerisidade Estadual de Campinas, Sept. 1998.
- [OGB98c] A. Oliva, I.C. Garcia, and L.E Buzato. The reflective architecture of guaraná. Technical report ic-98-14, Univerisidade Estadual de Campinas, Sept. 1998.
- [Oge04] F. Ogel. *Environnements d'exécution dynamiquement adaptables*. PhD thesis, Université Pierre et Marie Curie, Paris, France, 2004.
- [ope04] Opencm user's guide. http://opencm.objectweb.org/doc/0.8.1/user_guide.html, 04 2004.
- [OPPF03] F. Ogel, S. Patarin, I. Piumarta, and B. Folliot. C/span : A self-adapting web proxy cache. In *Autonomic Computing Workshop Fifth Annual International Workshop on Active Middleware Services*, page 178, June 2003. http://www-sor.inria.fr/patarin/pub/cspan_ams2003.pdf.
- [ORBa] Orbacus web site. http://www.orbacus.com/support/new_site/index.jsp.

- [orbb] Orbix web site. <http://www.iona.com/products/orbix.htm>.
- [Org72] E.I. Organick. *The Multics System : An Examination of Its Structures*. Cambridge, MA, 1972.
- [OTF05] F. Ogel, G. Thomas, and B. Folliot. Support efficient dynamic aspects through reflection and dynamic compilation. In *20th Annual ACM Symposium on Applied Computing (SAC'05, PSC)*, Santa Fe, New Mexico, USA, March 2005. To appear.
- [OTFP03] F. Ogel, G. Thomas, B. Folliot, and I. Piumarta. Application-level concurrency management. In *proceedings of the Concurrent Information Processing and Computing NATO Advanced Research Workshop (CIPC 2003)*, Sinaia, Romania, July 2003.
- [OTGF05] F. Ogel, G. Thomas, A. Galland, and B. Folliot. Mvv : une plate-forme à composants dynamiquement reconfigurables – la machine virtuelle virtuelle. In Jean-Louis Giavitto, editor, *Technique et Science Informatiques (TSI)*. Hermes Science, 2005. To appear.
- [OTP⁺03] F. Ogel, G. Thomas, I. Piumarta, A. Galland, B. Folliot, and C. Baillarguet. Towards active applications : the virtual virtual machine approach. In Mitica Craus, Dan Gâlea, and Alexandru Valachi, editors, *New Trends in Computer Science and Engineering*. A92 Publishing House, polirom press edition, 2003. ISBN 973-9476-40-6.
- [PBY03] F. Peschanski, J-P. Briot, and A. Yonezawa. Fine-grained dynamic adaptation of distributed components. In *ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, Rio de Janeiro, Brazil, 06 2003.
- [Piu04] I. Piumarta. The virtual processor : Fast, architecture-neutral dynamic code generation. In *3rd Virtual Machine Research and Technology Symposium (VM'04)*, pages 97–110, San Jose, CA, USA, 2004.
- [PM98] G. Pierre and M. Makpangou. Saperlipopette! : a distributed Web caching systems evaluation tool. In *Proceedings of the 1998 Middleware conference*, pages 389–405, 1998.
- [pro04] *ProActive Installation and User Guide*, April 2004. <http://www-sop.inria.fr/oasis/ProActive/doc/index.html>.
- [PRS04] R. Pawlak, J.P. Retaillé, and L. Seinturier. *Programmation Orientée Aspect pour Java/J2EE*. 2004.
- [PvS01] G. Pierre and M. van Steen. Globule : a platform for self-replicating web documents. In *proceedings of the 6th International Conference on Protocols for Multimedia Systems*, Oct 2001.
- [RBH00] N. Rouhana, S. Boustany, and E. Horlait. Yaap : Yet another active platform. In Springer, editor, *Mobile Agents for Telecommunication Applications, Second International Workshop, MATA 2000*, pages 185–194, 2000.
- [RF97] D. Reed and R. Fairbairns. *The NEMESIS kernel overview*, May 1997. <http://www.cl.cam.ac.uk/Research/SRG/netos/old-projects/pegasus/publications/overview.ps.gz>.

- [RMC⁺00] L. Réveillère, F. Mérillon, C. Consel, R. Marlet, and G. Muller. The devil language. Technique report RT-0244, IRISA/INRIA-Rennes, Equipe : COMPOSE, Oct. 2000.
- [RR98] E. Rose and K.H. Rose. Lightweight Bytecode Verification. *Formal Underpinnings of Java (an OOPSLA workshop)*, ACM, 1998. Vancouver, BC, Canada.
- [Sab04] N. Sabri. Le modèle de composants fractal. http://etna.int-evry.fr/equipe_sysrep/seminaire/Presentations/2004/presentationFractal.pdf, 2004. Seminaire de l'Institut National des Telecommunications.
- [SBL03] M.M. Swift, B.N. Bershad, and H.M. Levy. Improving the reliability of commodity operating systems. In *ACM Symposium on Operating Systems Principles*, Bolton Landing, New-York (USA), 2003.
- [SC00] D. Schmidt and C. Cleeland. Applying a pattern language to develop extensible orb middleware. *Design patterns in communications software*, pages 393–438, 2000.
- [SCS02] A. Senart, O. Charra, and J-B. Stefani. Developing dynamically reconfigurable operating system kernels with the think component architecture. In *Workshop on Engineering Context-aware Object-Oriented Systems and Environments (ECOOSE'2002)*, in association with OOPSLA '2002, Seattle, USA, November 2002.
- [Sel96] M. Seltzer. The world wide web : issues and challenges. presented at IBM Almaden, 7 1996. <http://www.eecs.harvard.edu/margo/slides/www.html>.
- [SGGB99] E.G Sirer, R. Grimm, A.J Gregory, and B.N Bershad. Design and implementation of a distributed virtual machine for networked computer. In *ACM Symposium on Operating Systems Principles*, pages 202–216, 1999.
- [Sha96] T. Shanley. *Protected Mode Software Architecture*. PC System Architecture Series. 1996.
- [SHT99] M. Van Steen, P. Homburg, and A. Tannenbaum. Globe : A wide-area distributed system. *IEEE Concurrency*, 7(1) :70–78, Jan-March 1999.
- [SKB⁺01] M. Sanders, M. Keaton, S. Bhattacharjee, K. Calvert, S. Zabele, and E. Zegura. Active reliable multicast on CANEs : A case study. In *Proc. of IEEE OpenArch*, 2001.
- [SM01] K. Shudo and Y. Muraoka. Asynchronous migration of execution context in java virtual machines. *Future Generation Computer Systems*, 18(2) :225–233, Oct. 2001.
- [smi] Synchronized multimedia integration language (smil 2.0). <http://www.w3.org/TR/smil20/>.
- [Smi82] B.C. Smith. *Procedural Reflexion in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, Feb 1982.
- [Smi84] B.C. Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 23–35, Salt Lake City, Utah, United States, 1984.
- [Smi90] B. C. Smith. What do you mean, meta ? *Workshop on reflection and Metalevel Architectures in OO Programming, ECOOP/OOPSLA '90, Ottawa, Ontario, Canada*, 1990.

- [SMY99] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A simple extension of java language for controllable transparent migration and its portable implementation. In *Coordination Models and Languages*, pages 211–226, 1999.
- [SSY00] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode transformation for portable thread migration in java. In *ASA/MA*, pages 16–28, 2000.
- [Sto81] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7) :412–418, 1981.
- [Sun03] Sun Microsystems. *Connected Limited Device Configuration – Specification v1.1*, March 2003.
- [Tat99] M. Tatsubori. An extension mechanism for the java language. Master’s thesis, Tokyo Institute of Technology, University of Tsukuba, 1999. Master of Engineering Dissertation, Graduate School of Engineering.
- [TCIK00] M. Tatsubori, S. Chiba, K. Itano, and M-O. Killijian. Openjava : A class-based macro system for java. In *Reflection and Software Engineering*, pages 119–135. Lecture Notes in Computer Science 1826, Springer-Verlag, July 2000.
- [TCM98] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *Symposium on Reliable Distributed Systems*, pages 135–143, 1998.
- [TFO03] G. Thomas, B. Folliot, and F. Ogel. Jnvm : Une plateforme java adaptable pour applications actives. In *Actes de la 3ème Conférence Française sur les Systèmes d’Exploitation (CFSE’03), Chapitre français de l’ACM-SIGOPS, GDR ARP*, La Colle sur Loup, France, Oct. 2003. ISBN 2-7261-1264-1.
- [TFP02] G. Thomas, B. Folliot, and I. Piumarta. Les documents actifs basés sur une machine virtuelle. In *Actes de la conf. Journées des Jeunes Chercheurs en Systèmes, Chapitre français de l’ACM-SIGOPS*, pages 441–447, Hammamet, Tunisie, April 2002.
- [THL02] P. Tullmann, M. Hibler, and J. Lepreau. Janos : A java-oriented os for active networks. In *DARPA Active Networks Conference and Exposition*, pages 117–129. IEEE, 2002.
- [TL94] C.A. Thekkath and H.M. Levy. Hardware and software support for efficient exception handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, San Jose, California, 1994.
- [TMC97] S. Thibault, R. Marlet, and C. Consel. A domain specific language for video device drivers : from design to implementation. In *Conference on Domain-Specific Languages*, Santa-Barbara, Oct. 1997.
- [TOG⁺05] G. Thomas, F. Ogel, A. Galland, B. Folliot, and I. Piumarta. Building a flexible java runtime upon a flexible compiler. In Jean-Jacques Vandewalle David Simplot-Ryl and Gilles Grimaud, editors, *Special Issue on ‘System & Networking for Smart Objects’ of IASTED International Journal on Computers and Applications*, volume 27, pages 28–47. ACTA Press, 2005.
- [TRV⁺00] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable support for transparent thread migration in java. In *ASA/MA*, pages 29–43, 2000.

- [TW96] D.L. Tennenhouse and D.J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), 1996.
- [VB94] G. Voelker and B. Bershad. Mobisaic : An information system for a mobile wireless computing environment. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 185–190, dec. 1994.
- [vDKV00] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages : An annotated bibliography. *ACM SIGPLAN Notices*, 35(6) :26–36, jun 2000.
- [vECGS92] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages : A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.
- [VM01] M. Vadet and P. Merle. Les containers ouverts dans les plates-formes à composants. In *Journées composants : flexibilité du système au langage*, 10 2001.
- [vOES96] J. van Ossenbruggen, A. Eliëns, and B. Schönhage. Web applications and sgml. In *proceedings of Conference on electronic publishing and document manipulation (EP96)*, sept. 1996.
- [Wei93] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7) :75–84, July 1993.
- [Wes97] D. Wessels. The squid internet object cache. National Laboratory for Applied Network Research/UCSD, software, 1997. [http ://www.squid-cache.org](http://www.squid-cache.org).
- [WGT98] D.J. Wetherall, J. Guttag, and D.L. Tennenhouse. Ants : A toolkit for building and dynamically deploying network protocols. In *Open Architectures and Network Programming*, San Francisco, april 1998. IEEE.
- [WLG98] D. Wetherall, U. Legedza, and J. Guttag. Introducing new internet services : Why and how. *IEEE Network Magazine, special issue on Active and Programmable Networks*, 1998.
- [WS97] Z. Wu and S. Schwiderski. Reflexive java : Making java even more reflexive. Ansa phase 3, ANSA – APM Limited, Cambridge, UK, Feb 1997.
- [WS99] I. Welch and R.J. Stroud. Dalang — a reflective extension for java. Technical Report CS-TR-672, University of Newcastle upon Tyne, September 1999.
- [WS01] I.S. Welch and R.J. Stroud. Kava - using bytecode rewriting to add behavioural reflection to java. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS 2001)*, pages 119–130, San Antonio, Texas, USA, January 2001. USENIX Association 2001.
- [xml] Xml home page. www.w3c.org/XML/.

Résumé

L'émergence de nouveaux domaines informatiques entraîne de nouveaux besoins en terme de mécanismes systèmes que les environnements traditionnels ne couvrent pas. Actuellement, il n'existe pas de solution pour introduire ces mécanismes sans introduire d'hétérogénéité entre les plate-formes d'exécution. Pour résoudre ce problème, nous proposons de placer le code spécialisé dans l'application et d'exécuter l'application, qui devient active, dans un environnement générique et standard.

Cette architecture repose sur une plate-forme hautement adaptable développée pendant ces travaux, la micro machine virtuelle. Elle a été testée avec une machine virtuelle Java réflexive et adaptable appelée la JnJVM. Pour valider notre approche, trois spécialisations de la JnJVM ont été implantées. Elles construisent des JVM dédiées au tissage d'aspects, à la migration d'un fil d'exécution et à de l'analyse d'échappement.

Abstract

The emergence of new computer fields involves new requirements in term of system mechanisms that traditional environments do not cover. Today, there is no solution to introduce these mechanisms without introducing heterogeneity between runtime environments. To solve this problem, we propose to put the specialized code in the application and to execute the application, which becomes active, in a generic and standard environment.

This architecture rely on a highly adaptable platform developed during this work, the micro virtual machine. It was tested with a reflexive and adaptable Java virtual machine called the JnJVM. To validate our approach, three specializations of the JnJVM were implemented. They build JVM dedicated to aspect weaving, to flow migration and to escape anlysis.