



Université des Sciences et Technologies de Lille

Numéro d'ordre : 3839

Spécialisation tardive de systèmes Java embarqués pour petits objets portables et sécurisés

Thèse de doctorat (spécialité Informatique) présentée le 20 septembre 2006

par Alexandre Courbot

Composition du jury

Président :	Jean-Marc Geib, LIFL, Université des Sciences et Technologies de Lille.
Rapporteurs :	Valérie Issarny, INRIA Rocquencourt. Michel Riveill, École Polytechnique de l'Université de Nice - Sophia Antipolis.
Examinateur :	Jean-Jacques Vandewalle, GemAlto Systems Research Labs.
Directeur :	David Simplot-Ryl, LIFL, Université des Sciences et Technologies de Lille.
Co-Encadrant :	Gilles Grimaud, LIFL, Université des Sciences et Technologies de Lille.

Révision 503 compilée le 25 septembre 2006 à 12:16.

À mon père,
Michel COURBOT,
24 décembre 1945 – 1er août 2006.

Table des matières

Avant-propos et remerciements

« Mes parents m'ont donné un corps et une éducation. Mon maître a fait de moi un homme. »

Le Bushido,
Code d'honneur et de morale du samouraï.

Ce document couvre le travail que j'ai effectué au cours de mes trois années de thèse. Bien entendu, en pratique ces années ne se limitent pas qu'au travail rapporté dans le mémoire. Pourtant, celui-ci n'accorde guère d'intérêt aux sentiments que ressent le futur docteur au cours de cette aventure. Fort heureusement, les avant-propos, un peu moins formels que le reste, permettent d'y remédier.

De mon passage de l'univers étudiant à celui de la recherche, j'ai ainsi retenu deux moments forts.

Le premier grand moment a été lorsque j'ai pris conscience que maintenant, il va falloir apprendre à se débrouiller un peu tout seul. Je me souviens encore très clairement du premier « demerden Sie dich » que m'a addressé Gilles avec un grand sourire, et de la panique qui s'en est suivie, proche de celle que l'on ressent lorsqu'on se fait addresser un « RTFM » après avoir posé une question un peu maladroite sur un forum.

L'autre grand moment étant bien évidemment la rédaction, mais détailler ce sujet serait gâcher le plaisir des prochains futurs docteurs qui parcourent les remerciements de leurs aînés dans l'espoir de savoir comment ça s'est passé pour eux.

Tout cela pour dire que la thèse est une aventure qui, bien que sanctionnant un travail personnel, fait intervenir de nombreuses personnes qu'il est maintenant temps de remercier.

Merci en premier lieu à Valérie Issarny *et à Michel* Riveill *de m'avoir fait l'honneur de rapporter ce travail. Merci également à Jean-Marc* Geib *d'avoir accepté de présider mon jury.*

Merci à David Simplot-Ryl *pour m'avoir accepté dans son équipe ainsi que pour sa manière toute à lui de rendre le travail agréable au quotidien. Sa rigueur couplée à sa bienveillance font que le travail sous sa direction a été à la fois plaisant et enrichissant.*

Je remercie ensuite bien évidemment Gilles Grimaud *pour son orientation, ses grandes idées, sa gentillesse et sa disponibilité. J'ai eu l'occasion avec lui d'avoir de nombreuses discussions mémorables qui démarraient de mon sujet de thèse pour rapidement sortir de leur cadre initial, comme lorsqu'il a tenté de m'expliquer la théorie des cordes (que je finirais bien par comprendre un jour) ou lorsque l'on cherchait à savoir si Dieu avait romizé l'univers actuel (théorie alternative au* Big-Bang *injustement dénigrée par la communauté des physiciens).*

Merci également à Jean-Jacques VANDEWALLE *pour avoir donné un avis industriel et réaliste sur mon travail qui m'a permis de le peaufiner.*

Merci à tous les membres, actuels et anciens, de l'équipe POPS que j'ai pu cotoyer : Nadia BEL-HADJ-AISSA, *Kévin* MARQUET, *Julien* GRAZIANO, *Jean* CARLE, *Farid* NAÏT-ABDESSELAM, *Vincent* CORDONNIER, *Vincent* BENONY, *François* INGELREST, *Damien* DEVILLE, *Fadila* KHADAR, *Michaël* HAUSPIE, *Antoine* GALLAIS, *Hervé* MEUNIER, *Mamhoud* TAÏFOUR, *Nathalie* MITTON, *Christophe* DEMAREY, *Julien* CARTIGNY, *Caroline* FONTAINE, *Rémy* OBEIN. *Merci en particulier à tous ceux qui ont relu le présent document.*

Merci aux personnes extérieures à l'équipe que j'ai eu le bonheur de fréquenter, en particulier Ouassila LABBANI, *Yann* HODIQUE, *Dorina* GHINDICI, *Isabelle* SIMPLOT-RYL, *Damien* DEVIGNE, *Antoine* HONORÉ, *Joël* VENNIN, *Romain* ROUVOY *et j'en oublie sans doute plein...*

Je ne peux pas non plus oublier l'équipe administrative du LIFL à qui j'ai parfois demandé des miracles qui ont souvent été accomplis.

Les applications de cette thèse ont été effectuées sur le projet JITS. Merci donc à tous les ingénieurs, stagiaires, et autres personnes ayant travaillé dessus.

Ce document a été rédigé sous Kubuntu Linux, à l'aide de LaTeX, en utilisant l'éditeur Kile. Les figures ont été réalisées en utilisant TikZ. Que leurs auteurs et contributeurs en soient remerciés.

Bien évidemment, je ne puis oublier ma famille qui se demandait parfois avec inquiétude pourquoi je restais au labo aussi tard : ma mère, Marie-Jeanne, pour toutes les bonnes raisons que j'ai de la remercier, ma sœur Isabelle et mon beau-frère Jean-Jacques, à qui je souhaite beaucoup de bonheur.

Enfin, ce mémoire *ne mériterait pas ce titre si je ne rappelais pas celle de celui sans qui rien de tout cela n'aurait été possible. Je remercie mon père, Michel* COURBOT, *Médaille Militaire, Chevalier dans l'Ordre National du Mérite, Médaille de la Défense Nationale, Médaille de Reconnaissance de la Nation, Médaille d'Honneur pour Actes de Courage et de Dévouement, Médaille des Opérations Extérieures en Ex-Yougoslavie, Médaille des Nations Unies, mais avant tout une personne dévouée jusqu'au bout à sa famille et à ses responsabilités. Merci de m'avoir donné les moyens de réussir là où je souhaitais aller. Merci de m'avoir enseigné, par les actes plus que par les paroles, ces choses qui ne s'apprennent pas à l'école. La vie a parfois une façon cruelle de nous dire à son tour «* demerden Sie dich *» alors que l'on a encore beaucoup à apprendre. J'imagine que c'est, en quelque sorte, une nouvelle thèse qui commence...*

Premier Chapitre

Introduction

« It will be an Internet of things. Your stuff will be connected so you don't have to [be]. Your clocks, cars, pets, thermostats and luggage will be connected. Your car will run an auction with gas stations in range, negotiating best price to refill your tank [and] will also check to see what is low in your fridge and direct you to [the] nearest gas station with convenience store. »

Scott McNealy,
PDG de Sun Microsystems.

1.1 Petits objets portables et sécurisés

Les évolutions du domaine informatique de ces dernières années annoncent une nouvelle répartition de la puissance de calcul, un véritable « new deal » de l'informatique domestique. Jusqu'alors, les moyens de calcul étaient concentrés au sein d'une seule et unique machine sédentaire capable d'accomplir un grand nombre de tâches, ce qui la rend complexe à utiliser pour ses utilisateurs et leur impose un apprentissage particulier.

Cette vision centralisée de l'informatique est progressivement en train de disparaître au profit d'une informatique enfouie, omniprésente, spécialisée et transparente [Weis 93]. L'avènement des téléphones portables, la banalisation des cartes à puce, la généralisation des ordinateurs de bord et des récepteurs GPS dans les automobiles, l'émergence de la domotique, ... sont les prémices de cette informatique ubiquitaire qui n'en est encore qu'à ses balbutiements.

Selon les principes de l'informatique ubiquitaire, une bonne technologie assiste la réalisation d'une tâche et ne doit surtout pas prendre l'ascendant sur celle-ci, par exemple en se matérialisant dans une machine dont le seul but est d'effectuer des calculs. Aussi, contrairement aux ordinateurs personnels, les cartes à puce ou les téléphones portables sont des objets du quotidien *améliorés* avec des capacités informatiques, que l'utilisateur perçoit selon leur fonction première, sans avoir conscience de tenir un ordinateur dans sa main : l'informatique est alors invisible et *enfouie* dans les objets familiers.

1.2 Orientation de la recherche

Cette nouvelle vision de l'informatique apporte avec elle ses défis, de relation avec l'utilisateur d'abord : l'interaction avec celui-ci doit rester celle d'un objet du quotidien et non pas devenir celle d'un ordinateur ; notamment, la maintenance du système doit être quasi nulle. Ensuite, ces nombreux petits systèmes doivent être capable de communiquer entre eux (en supportant des interfaces et protocoles de communication standardisés, comme par exemple les technologies Bluetooth [Bluetoot 01]) et de coopérer pour remplir au mieux les besoins de l'utilisateur. Enfin, d'un point de vue industriel, les défis sont également importants : outre les problèmes liés à la consommation énergétique [Simu 99], le coût de production de ces nouveaux équipements, généralement produits par millions, doit être aussi bas que possible pour que ces derniers soient commercialement viables et compétitifs. Réduire le coût de production d'un équipement informatique signifie principalement réduire sa complexité, ainsi que la puissance de calcul et la quantité de mémoire qu'il embarque.

Parallèlement à cela, le domaine du génie logiciel a également évolué, avec des moyens permettant aux programmeurs de produire des applications plus sûres, plus facilement. L'expertise des développeurs s'oriente naturellement vers ces nouvelles technologies, qui nécessitent des environnements d'exécution plus complexes et qui par conséquent sont rarement utilisées dans les équipements embarqués. À cet égard, Java constitue un excellent exemple : Conçu pour pouvoir être utilisé partout, l'environnement Java s'est alourdi de nombreuses fonctionalités et n'est finalement pas utilisable dans les équipements embarqués, à moins d'utiliser des versions dégradées et incompatibles comme J2ME [J2ME 00] ou Java Card [Java 03]. Mais dans ce dernier cas, la règle d'or de Java « *Compile once, run everywhere* » n'est plus respectée.

C'est dans ce contexte que s'est développée notre recherche visant à proposer le Java original aux équipements embarqués et contraints. Nous partons du principe que l'environnement Java initial ne peut certes pas être utilisé tel quel sur ces équipements, et qu'une spécialisation est par conséquent nécessaire pour d'obtenir une version embarquable. Cependant, nous ne soutenons pas l'idée reprise par J2ME et Java Card d'imposer un environnement dégradé auquel les applications doivent se conformer. En plus d'être incompatible avec l'édition standard de Java, la spécialisation qu'il subit est arbitraire et limite son champ applicatif ainsi que la gamme de matériel qu'il peut piloter.

Nous pensons au contraire que la spécialisation des environnements Java embarqués doit se faire au cas par cas. Pour permettre cela, nous proposons d'effectuer la spécialisation du système après son déploiement complet. De cette manière, les applications embarquées restent compatibles avec le Java standard. De plus, la spécialisation effectuée sur le système déployé peut alors tenir compte des applications qu'il contient, de leur contexte d'exécution, et de l'équipement sur lequel elles s'exécutent. Pour permettre le déploiement complet du système avant sa spécialisation, nous utilisons la notion de *romization*, que nous définissons comme l'acte de déployer un système en dehors de son environnement d'exécution réel, puis d'en capturer une image mémoire afin de le migrer vers celui-ci, de telle sorte qu'il puisse y continuer son exécution.

Les contributions de cette thèse sont donc les suivantes :

- Une formalisation de la notion de romization ainsi que sa localisation dans le cycle de vie du logiciel destiné à des équipements embarqués ;
- Une proposition d'architecture de romization pour systèmes Java capable de prendre

en charge l'intégralité du déploiement du système ;
- Un spécialiseur d'environnements Java déployés hors-ligne par notre architecture de romization, tirant profit de la richesse d'informations disponible à propos du système.

1.3 Organisation du document

Ce document présente une démarche relatée de manière chronologique — ainsi, il est préférable de le lire du début à la fin, les idées s'enchaînant entre les chapitres. Toutefois, chaque chapitre est écrit de façon à être abordable de manière indépendante. À chaque fois qu'un concept d'un chapitre précédant s'avère nécessaire, une référence y est faite. L'organisation générale du document est la suivante :

Le chapitre 2 établit un état de l'art sur les systèmes embarqués, leur déploiement, leur spécialisation, et sur les systèmes Java destinés au monde de l'embarqué. Nous insistons plus particulièrement sur les caractéristiques de Java qui le rendent difficile à embarquer, comme le mécanisme de chargement de classes, et énumérons les alternatives possibles qui comprennent notamment la romization. Cette notion n'étant pas clairement définie dans la littérature, nous observons comment les dérivés embarquables de Java l'emploient, ce qui nous amène à conclure que ces solutions ne sont pas optimales car elles obligent le système à inclure des fonctionalités lui permettant de terminer son déploiement après sa migration.

Le chapitre 3 extrait les caractéristiques principales des solutions de romization existantes afin de formaliser ce processus et d'en donner une définition globale. À la lumière de cette définition, il propose une nouvelle architecture pour effectuer la romization, prenant en charge tous les aspects du déploiement et de l'exécution du système et dans laquelle les différentes préoccupations de la romization sont clairement séparées. Les détails de l'implémentation que nous en avons réalisée sont ensuite donnés.

Le chapitre 4 s'intéresse aux possibilités fortes de spécialisation offertes par l'architecture de romization que nous avons présentée. Il présente et formalise l'analyse qui est effectuée sur le système déployé, puis présente différentes spécialisations applicables sur le système à partir de son résultat.

Dans le chapitre 5, nous présentons des résultats expérimentaux obtenus en utilisant notre architecture de romization et une implémentation de nos procédés de spécialisation. Nous validons notre assertion de départ qui est que la spécialisation devient plus efficace avec l'avancement du déploiement du logiciel ; puis nous évaluons la précision obtenue sur les analyses ainsi que le gain en termes d'empreinte mémoire obtenu sur le système.

Enfin, le chapitre 6 conclue sur nos travaux et synthétise les réponses qu'ils ont apporté. Il présente également les limitations de notre approche, notamment concernant l'extensibilité du système spécialisé. Nous fermons alors notre étude par une énumération non-exhaustive des possibilités de recherches liées à la romization que nous n'avons pas étudiées dans le présent travail.

Deuxième Chapitre

Systèmes Java embarqués

> « Dans quelques années, les avions seront pilotés par un commandant de bord et un chien. Le travail du chien sera de mordre le pilote s'il essaie d'appuyer sur les boutons. »
>
> **Scott Adams**.

Les systèmes embarqués constituent un domaine à part de l'informatique, de par leurs contraintes matérielles, qui conditionnent la manière dont le logiciel et développé, et également de par leur mode de fonctionnement particulier. Dans ce chapitre, nous allons tout d'abord caractériser les systèmes embarqués et leurs spécificités, ainsi que les approches employées pour leur construction. Nous allons ensuite expliquer la popularité de Java dans ces systèmes, décrire les variantes de Java destinées aux systèmes embarqués, et mettre l'accent sur les problèmes du déploiement d'applications Java embarquées. Ceci nous amènera à considérer les utilisations qui sont faites d'un concept encore mal défini dans la littérature : la romization.

2.1 Systèmes embarqués

Seuls 2% des microprocesseurs produits au cours de l'année 2000 ont été installés dans des équipements interactifs, le reste étant employé dans des systèmes embarqués autonomes [Tenn 00]. Rien ne laisse à penser que la tendance ait changé aujourd'hui, bien au contraire : l'informatique enfouie prend de plus en plus d'importance dans la vie de tous les jours, avec notamment l'utilisation de plus en plus intensive des cartes à puce, des étiquettes RFID [Fink 03] et des objets de la vie quotidienne augmentés par des capacités informatiques [Fuji 05]. Afin de situer le contexte global de nos travaux, nous allons commencer par essayer de caractériser ces systèmes embarqués.

2.1.1 Caractérisation

Il n'existe pas à proprement parler de définition standard pour désigner un système embarqué. Cependant, la littérature [Gajs 94, Gans 03, Edwa 03] nous renseigne sur la compréhension que l'on peut avoir de ce terme. Un système embarqué affiche ainsi communément les caractéristiques suivantes :

Composé de matériel et de logiciel conçus conjointement Les systèmes embarqués forment un ensemble *hard/soft* cohérent et souvent figé. Logiciel et matériel sont ainsi

assemblés conjointement, contrairement à d'autres domaines où le développement logiciel est conditionné à l'avance par la plate-forme matérielle sur laquelle il doit s'exécuter. Le co-design matériel/logiciel [De M 97] est ainsi un thème de recherche encore particulièrement actif.

Enfoui dans l'équipement qu'il contrôle Le système embarqué n'est pas auto-suffisant comme un ordinateur personnel. Son existence dépend de l'équipement qu'il contrôle, et dont il fait partie intégrante.

Dédié à une ou plusieurs tâches précises Contrairement aux autres systèmes qui restent généralement aussi universels que possible, un système embarqué est exclusivement dédié à sa fonction de contrôle de l'équipement. Ce point se ressent fortement dans sa conception.

Fonctionnant avec peu ou pas d'intervention humaine Bien souvent, l'utilisateur n'a même pas conscience du système informatique qui contrôle son équipement.

Limité en ressources Les contraintes de coût, de consommation énergétique et d'occupation physique limitent grandement le matériel embarquable. La puissance de calcul ainsi que la quantité de mémoire disponible sont ainsi très limités.

Ayant de hautes exigences de sûreté Les systèmes embarqués sont conçus pour être émis dans un milieu où l'expertise informatique n'est pas disponible. Parfois, ils ne sont même pas atteignables physiquement. Ils doivent par conséquent être en mesure de fonctionner dans toutes les conditions qu'ils peuvent être amenés à rencontrer.

Les applications de ces systèmes couvrent des domaines extrêmement variés, allant des *pacemakers* aux missiles de croisière, en passant par des objets plus familiers et pacifiques comme les téléphones cellulaires ou les cartes à puce.

Les équipements qui nous intéressent sont les POPS : *Petits Objets Portables et Sécurisés*, équipements produits en masse et dont le coût par unité doit être le plus bas possible pour le fabricant. Afin de réduire ce coût, le matériel qu'ils embarquent est réduit au strict minimum nécessaire pour accomplir leur tâche.

2.1.2 Matériel pour systèmes embarqués

Un système embarqué se distingue en premier lieu par sa configuration matérielle très hétéroclite. Celle-ci est particulière à un équipement donné : contrairement à l'informatique grand public où des « gammes » peuvent facilement être déclinées, la mise au point d'un système embarqué est toujours une opération particulière et unique, dans laquelle des besoins différents donneront lieu à des choix différents de matériel. Cependant, comme tout système informatique, un système embarqué comprend au minimum une unité de calcul ainsi que de la mémoire, auxquels peuvent s'ajouter des interfaces de communication, voire des liaisons réseau.

Microprocesseur L'unité de calcul d'un équipement embarqué est choisie en fonction des besoins en terme de puissance de calcul de l'équipement, de son coût, ainsi que de sa consommation énergétique, de telle sorte à minimiser ces deux derniers paramètres. Les applications ne nécessitant pas beaucoup de puissance de calcul se rabattent sur des microcontrôleurs, très bon marché et fonctionnant avec peu d'énergie. Ces circuits tout-en-un intègrent également dans la même puce la mémoire nécessaire et des interfaces d'entrée/sortie. Ils permettent

ainsi d'embarquer l'intelligence d'un équipement dans un espace très réduit et pour un coût minimal, pour peu que les besoins en terme de puissance de calcul soient très modestes. Les applications plus lourdes, telles que les téléphones portables, embarquent généralement un microprocesseur étudié pour le monde de l'embarqué, comme ceux de la famille ARM. Celui-ci est alors couplé aux autres matériaux formant un système informatique par un bus.

2.1.2.1 Mémoire

Concernant la mémoire, différents types sont présents dans un système embarqué, chacune ayant ses propres caractéristiques en termes de performance, de point mémoire[1], de coût, de capacité d'écriture et de persistance. Pour fournir un service, un système embarqué nécessite un minimum de mémoire de travail ou RAM. Cependant, le point important de cette mémoire et son coût élevé tendent à limiter son utilisation. De plus, cette mémoire n'étant pas persistante, elle n'est pas adaptée à tous les cas d'utilisation : toutes les données qu'elle contient sont perdues en cas de coupure de l'alimentation du système. Une mémoire adaptée (EEPROM ou Flash) est donc nécessaire pour les systèmes ayant besoin de données persistantes. Elles ne constituent pas pour autant un substitut à la RAM, car leurs performances, notamment en ce qui concerne l'écriture pour l'EEPROM, les rendent impraticables comme mémoire de travail. Leur point et leur coût sont également loin d'être négligeables. La ROM est la mémoire qui offre le point ainsi que le coût les plus faibles, tout en gardant de bonnes performances en lecture et en étant persistante. Ces bonnes propriétés encouragent à maximiser son utilisation dans un équipement embarqué. Cependant, elle n'est pas réinscriptible par l'équipement qui l'embarque : la ROM est initialisée une fois pour toute en dehors de son équipement cible. Elle est donc utilisée pour stocker des données statiques, comme le code des programmes embarquées ou des données purement constantes.

2.1.2.2 Interfaces d'entrée/sortie

La présence de périphériques permettant d'interagir avec l'utilisateur n'est pas systématique dans un équipement embarqué. Beaucoup d'entre eux fonctionnent en effet de manière 100% autonome : on pourra citer les étiquettes électroniques ou les capteurs. La carte à puce ne dispose pas d'interface directe avec l'utilisateur : en revanche, elle peut se servir du lecteur dans lequel elle est insérée pour, par exemple, demander son code secret. Les appareils plus gros comme les téléphones portables ou les appareils photos numériques incluent des périphériques d'entrée/sortie assez familiers pour l'utilisateur d'ordinateurs, comme un écran, une série de boutons de contrôle, voire un périphérique de pointage.

2.1.2.3 Liaison réseau

La présence d'une liaison réseau dans un équipement embarqué n'est également pas systématique. Sa présence et ses caractéristiques se justifient par les besoins en communication de l'équipement. Ainsi, un système de contrôle d'ABS fonctionne de manière totalement déconnectée, une carte à puce communique avec son lecteur par le biais d'une liaison série à faible débit, et un téléphone portable nouvelle génération peut se connecter à l'internet via TCP/IP grâce à sa connexion sans fil.

[1] Le *point mémoire* désigne l'occupation physique de la cellule de base de la mémoire, utilisée pour coder un *bit*.

Le tableau 2.1 résume les caractéristiques de quelques équipements de différentes gammes.

TAB. 2.1 – Les systèmes embarqués couvrent une gamme très variée de matériel.

Équipement	Type	Processeur	Mémoire	Périphériques
Carte Platinium	Carte à puce	μcontrôleur Atmel AT90SC6464C câblé crypto	64 Ko EEPROM, 64 Ko Flash, 3 Ko RAM	Ligne série 625 Kbps
MICAz	Capteur	μcontrôleur Atmel ATmega128L	128 Ko Flash, 4 Ko EEPROM, 4 Ko SRAM	Émetteur radio à 250 Kbps
Zaurus SL-5500	PDA	StrongARM 206 Mhz	64 Mo RAM, 16 Mo Flash ROM	Clavier, écran, audio, liaison réseau USB ou 802.11b

Cette très grande variété dans le matériel utilisable, ainsi que les limitations des équipements les plus contraints, conditionnent fortement la manière dont le logiciel embarqué est écrit.

2.1.3 Systèmes d'exploitation pour matériel embarqué

Le logiciel embarqué en général, et les systèmes d'exploitation embarqués en particulier, sont un cas particulier d'ingénierie logicielle. En effet, les fortes contraintes de coût à l'unité, de consommation énergétique et d'encombrement physique font qu'ils doivent plus que n'importe quel autre logiciel fonctionner de manière optimale avec le matériel qu'ils utilisent, afin de réaliser au mieux la tâche demandée tout en limitant les ressources utilisées.

Par conséquent, les logiciels embarqués sont très souvent écrits au cas par cas en fonction de la cible matérielle. Cette pratique augmente de façon considérable le coût du logiciel, puisque celui-ci n'est alors que très peu réutilisable. La réécriture constante du code pour diverses architecture ou divers usages particuliers a également un autre effet pervers : elle augmente le risque d'y insérer des erreurs.

Ces problèmes peuvent être adressés par le biais de la personnalisation des systèmes à embarquer [Kicz 93, Deny 02], qui consiste à produire un système adapté à une tâche spécifique à partir d'un système plus générique.

2.1.3.1 Un besoin de personnalisation

Nous pouvons recenser deux approches principales à la personnalisation de systèmes embarqués [Ripp 04b]. Une première approche consiste à partir d'un noyau de système minimaliste et à y ajouter les composants permettant de remplir les tâches désirées. Cette approche est désignée sous le nom d'*approche constructive*. La seconde approche est tout l'inverse : un système existant et complet est réduit en une variante spécialisée et embarquable. Nous désignons cette approche sous le nom d'*approche destructive* ou *approche spécialisante*.

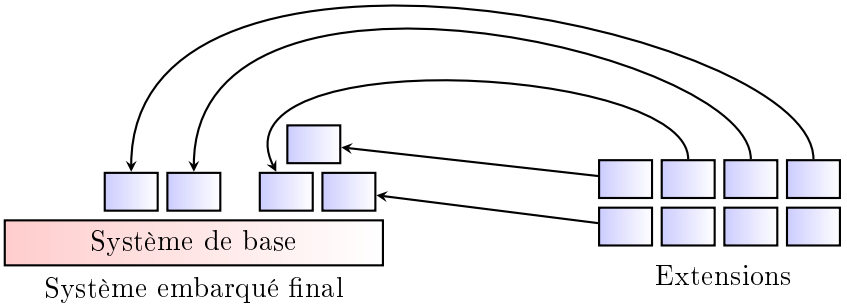


FIG. 2.1 – L'approche constructive à la construction de systèmes embarqués permet de concevoir un système par l'addition de « briques » fournissant chacune l'implémentation d'un service.

2.1.3.2 L'approche constructive

L'approche constructive est largement répandue dans la conception de systèmes d'exploitation, que ceux-ci soient embarqués ou pas. Nous pouvons parmis de nombreux exemples citer OSKit [Ford 97], Fractal [Brun 02] et Think [Fass 01]. Elle consiste à fournir un système minimal qui est étendu par insertion d'extensions logicielles. Le système désiré est ainsi construit par l'ajout de différentes « briques » qui forment un ensemble cohérent, le concepteur du système choisissant celles dont il a besoin (Figure 2.1).

La construction du système peut se faire soit de manière statique (le système est construit avant son déploiement effectif, par exemple pendant sa compilation), soit de manière dynamique (le système charge lui-même des éléments additionnels au cours de son exécution). Le noyau Linux illustre bien ces deux temps dans la construction du système. Un module noyau étend les fonctionnalités du noyau Linux en fournissant un service tel que le support d'un matériel particulier, d'un protocole réseau ou d'un système de fichiers. Ces modules peuvent être directement inclus dans le noyau pendant la compilation de ce dernier, ou bien compilés en tant qu'entités séparées chargeables (et déchargeables) dynamiquement dans un noyau en cours d'exécution. Ainsi, le noyau Linux est déclinable sur une gamme extrêmement variée de matériel : le chargement dynamique de modules permet de gérer facilement les changements de configuration matérielle ou de besoins, fréquents sur un ordinateur personnel. La liaison statique permet quant à elle de minimiser l'empreinte mémoire du noyau dans les cas où les ressources sont limitées et la configuration matérielle bien fixée.

Il existe de nombreuses manières d'écrire un système extensible [Selt 97]. Les paragraphes suivants listent quelques-unes des approches concernant les systèmes embarqués, selon qu'elles se basent sur une construction statique ou dynamique du système.

Construction statique L'approche constructive statique consiste à définir l'ensemble des composants désirés dans un système, et à les lier entre eux statiquement lors d'une phase de construction qui se déroule avant le déploiement du système sur son équipement cible (par exemple, la compilation). Le système ainsi créé est généralement figé, mais n'embarque pas

de mécanismes d'extension qui ont un coût sur son empreinte mémoire. De plus, les parties constantes du système ainsi construit (code, méta-données, ...) peuvent être placées dans une mémoire à lecture seule.

Bien que le principe reste globalement le même, à savoir un assemblage de divers modules sur un support logiciel de base, la construction statique peut se faire selon différentes méthodologies.

OSKit [Ford 97] est l'exemple typique de construction statique de système d'exploitation. Chaque composant du système est présenté sous la forme d'une bibliothèque qui peut être connectée à d'autres bibliothèques pour former un système complet et cohérent. L'originalité d'OSKit réside dans le fait que ces bibliothèques sont en fait des composants de systèmes d'exploitations déjà existants, tels que Linux ou FreeBSD. Il est ainsi possible d'obtenir des compositions originales, telles qu'un système utilisant la pile TCP/IP de FreeBSD et les pilotes de cartes réseau de Linux.

Think [Fass 01, Fass 02], propose de construire les noyaux de systèmes d'exploitation à partir de composants. Son originalité est de laisser libres non seulement le choix des composants à utiliser, mais également la manière dont ces derniers sont reliés entre eux. Le modèle de liaison peut ainsi être basé sur de simples pointeurs, ou au contraire passer par des schémas complexes de communication avec sérialisation/désérialisation et passage de messages au travers d'une liaison réseau. TinyOS [Levi] est un autre système d'exploitation pour réseaux de capteurs, très utilisé, basé lui aussi sur une architecture à composants.

Nous pouvons également citer les systèmes d'exploitation orientés objet, parmis lesquels SOS [Shap 89], Apertos [Yoko 92], Choices [Camp 93, Camp 96] et Taligent [Ande 94, Myer 95], dont chaque composant du système est un objet interchangeable.

Construction dynamique La construction dynamique de système consiste à déployer les modules du système non plus pendant la phase de construction de celui-ci, mais au cours de son exécution dans son environnement final. Dans ce cas, le système doit disposer de suffisamment de mémoire réinscriptible pour charger le module, et inclure les mécanismes lui permettant de l'installer et de le désinstaller. Les noyaux Linux embarqués [Beck 96] permettent cette opération, de même que le système d'exploitation pour réseaux de capteur SOS [Han 05][2], mais cette approche pose des problèmes de sécurité : en effet, le code chargé dynamiquement dans le noyau fonctionne avec les privilèges de ce dernier et peut ainsi interférer sans restriction avec tout le système, ce qui pose problème si le code chargé n'est pas de confiance. On peut alors soit s'assurer de la sûreté du code, par exemple en utilisant le principe du code portant sa preuve [Necu 97], soit chercher à isoler les extensions dans des zones protégées.

Les micro-noyaux permettent cela en rompant avec la tradition monolithique des noyaux. Seule une partie minime du noyau est située en espace privilégié, les autres services s'exécutant en espace utilisateur et communiquant avec le micro-noyau par envoi de messages. Les modules chargés dynamiquement voient leur espace d'adressage limité à la zone non-privilégiée, et leur capacité à nuire au système est ainsi amoindrie. Les micro-noyaux sont aujourd'hui employés avec succès, et les implémentations sont légion. Nous pouvons citer l'historique MINIX [Tane 87], ainsi que Mach [Rash 89], et SPIN [Bers 94].

[2] Bien qu'homonyme avec le système orienté objet SOS [Shap 89] dont nous venons de parler, SOS [Han 05] n'a aucun lien avec celui-ci.

Les exo-noyaux [Engl 95b, Engl 95a, Engl 98] poussent le concept des micro-noyaux encore plus loin, puisqu'ils considèrent en plus les abstractions proposées par le système comme des extensions, la partie privilégiée de celui-ci se contentant de démultiplexer les accès aux ressources matérielles. Ce domaine de recherche est encore très actif de nos jours. Nous pouvons ainsi citer les projets exOS [Engl 98] et CAMILLE [Grim 00].

Il est à noter que ces deux axes de constructions ne sont pas exclusifs : un système peut très bien être construit statiquement, puis être étendu au cours de son exécution et ainsi supporter les constructions statiques et dynamiques. C'est le cas notamment de Think.

Discussion sur l'approche constructive De cet état des lieux de l'approche constructive à la conception de systèmes d'exploitation embarqués, nous pouvons retenir plusieurs points. Tout d'abord, le grand succès de systèmes comme TinyOS témoigne du bien fondé de cette approche pour des cas d'utilisation très particuliers. Ensuite, l'approche constructive permet l'extensibilité du système pendant son exécution, voire même le remplacement de certains de ses composants à la volée, et ce pour un coût relativement faible. Ce point est particulièrement intéressant dans le cadre des systèmes embarqués pour lesquels une maintenance est à prévoir. Les deux axes possibles de construction, statique et dynamique, laissent également la possibilité pour le fabricant de déterminer quelles fonctionalités sont à inclure dans le système de base, et quelles autres peuvent être chargées à la demande.

Toutefois, les limites de cette approche de ressentent sur plusieurs points :

- De par son concept même, la granularité de ce type de construction ne peut pas être plus fine que les briques servant à le construire, qui sont des entités indivisibles. Le besoin d'un composant système plus spécialisé nécessite la réécriture de celui-ci ;
- Chacun de ces systèmes implémente ses propres abstractions (ou incite à le faire, dans le cas des exo-noyaux), ce qui rend la portabilité au niveau applicatif quasiment nulle ;
- Par conséquent, les systèmes embarquables sur des équipements contraints se limitent à une problématique particulière (par exemple, les réseaux de capteurs pour TinyOS).

L'approche spécialisante propose une manière alternative pour personnaliser un système qui se situe à l'opposé de celle de l'approche constructive.

2.1.3.3 L'approche spécialisante

L'approche spécialisante à la construction de systèmes embarqués consiste à extraire d'un système générique non-embarquable un sous-ensemble embarquable qui soit adapté à l'accomplissement exclusif d'une ou de plusieurs tâches supportées par le système d'origine.Elle permet de s'affranchir partiellement ou complètement des problèmes de granularité de spécialisation, de portabilité des applications existantes et du nombre de couches d'abstractions à franchir pour accéder au matériel.

Le système spécialisé est créé à partir du système complet grâce à la connaissance partielle de son contexte d'exécution, qui peut comprendre entre autres des informations sur l'architecture cible ou sur les applications à exécuter. Cette connaissance permet d'extraire du système générique le sous-ensemble nécessaire pour satisfaire ce contexte d'exécution, sans tenir compte des autres (Figure 2.2). La spécialisation peut alors être faite soit de façon manuelle, soit à l'aide d'un outil dédié.

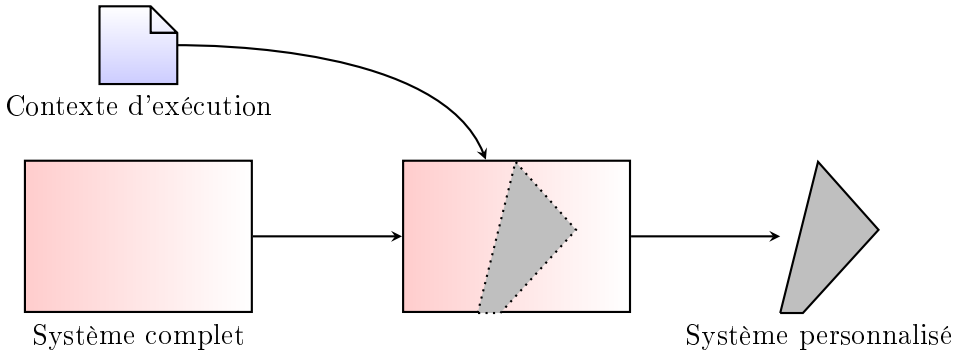


FIG. 2.2 – L'approche spécialisante à la construction de systèmes embarqués consiste à extraire le sous-ensemble utile d'un système générique existant afin d'en obtenir une variante spécialisée et embarquable.

Spécialisation manuelle La spécialisation peut être effectuée de manière manuelle, pour par exemple produire une version embarquable d'un système qui ne l'est initialement pas. Ainsi, le projet μcLinux [μCL] est un dérivé manuel de Linux pour microcontrôleurs. Nous pouvons également citer les variantes embarquées de Java, comme J2ME [J2ME 00] et Java Card [Java 03], sur lesquelles nous allons revenir plus loin dans ce chapitre.

Ce moyen de spécialisation est difficilement maintenable et nécessite un investissement considérable en travail humain. De plus, il n'a que peu d'intérêt scientifique. Fort heureusement, il est également possible de produire une variante spécialisée d'un programme par des moyens automatiques.

Spécialisation automatique Ceux-ci se basent typiquement sur une phase d'analyse du système et de son contexte d'exécution, dont le résultat est utilisé afin d'extraire le sous-ensemble utile à son exécution dans ce contexte donné. Cette analyse prend généralement la forme d'une interprétation abstraite [Jone 94, Cous 96], qui consiste à exécuter partiellement un programme dans le but d'obtenir des informations sur sa sémantique sans pour autant effectuer tous les calculs requis par son exécution complète. Cette technique est communément utilisée pour obtenir une approximation du graphe de flot de contrôle d'un programme [Alle 70], procéder à une inférence de type [Wand 87, Pals 91], résoudre certains appels de méthodes virtuelles [Sund 00] ou faire une vérification sur les propriétés d'un programme [Lero 01, Comi 03].

Des techniques particulières permettent alors de spécialiser un programme, en exploitant le résultat d'une interprétation abstraite ou de manière autonome. Ainsi, le découpage de programme [Weis 81, Tip 95, Hong 05] extrait la partie d'un programme affectant un ensemble de ses variables à un moment donné de son exécution. Cette pratique permet d'obtenir un programme spécialisé ne s'intéressant qu'à un sous-ensemble des fonctionnalités du programme original.

L'évaluation partielle [Jone 93, Cons 93], quant à elle, produit une variante spécialisée d'un programme à l'aide d'invariants spécifiés par l'utilisateur [Le M 04] ou inférés (par exemple) au travers d'une interprétation abstraite. Ceux-ci sont alors propagés dans tout le programme qui est ensuite simplifié.

Les extracteurs de bibliothèque [Rays 02, Tip 03] permettent d'extraire le sous-ensemble utile des fonctions d'une bibliothèque à partir de la connaissance de programmes les utilisant.

Nous pouvons également citer la restructuration de programme, qui consiste à optimiser les structures de données d'un programme. Ainsi, la déclassification [Powe 05] est une spécialisation dédiée aux langages objets permettant de supprimer certaines classes intermédiaires inutilisées dans un programme, en les fusionnant avec les classes utiles.

Toutes ces approches sont complémentaires car elles peuvent s'effectuer au même niveau. Ainsi, étant donné un code source de programme, il est possible d'y effectuer un découpage pour isoler certaines fonctionnalités, puis de passer le code source produit à un évaluateur partiel avant de supprimer les structures de données devenues inutilisées après ces premières spécialisations.

Discussion sur l'approche spécialisante L'approche spécialisante est l'exact inverse de l'approche constructive : le point de départ est un programme ou un système générique qui est transformé en une version « dégradée », apte à accomplir un sous-ensemble des tâches du système initial. La granularité de ce genre de personnalisation est très fine, car elle s'effectue sans élément de système atomique. De plus, la variété des techniques utilisables permet d'intervenir tant sur le code du programme que sur ses structures de données. L'approche destructive permet donc d'obtenir un système spécialisé plus finement que l'approche constructive. Sa nature lui permet également de travailler sur des environnements standardisés et donc d'assurer une certaine portabilité des programmes *avant* spécialisation.

Cependant, cette forme de personnalisation rend difficile toute extensibilité du système spécialisé produit. Celui-ci est en effet le résultat d'une opération d'analyse et d'adaptation effectuée de manière statique. Cette opération est irréversible une fois effectuée, et l'ajout de nouvelles fonctionalités à l'exécution ne peut se faire que si celles-ci sont en conformité avec les spécialisations déjà effectuées, ce qui limite grandement les possibilités d'évolution du système[3].

Ces problèmes d'évolution du système déployé vont en réalité au-delà des systèmes spécialisés. Le cas général du déploiement d'applications embarqué constitue en effet une problématique constante.

2.1.4 Problèmes liés au déploiement des applications embarquées

Le chargement de nouvelles applications dans un système embarqué déjà déployé n'est pas toujours faisable : elle est notamment techniquement impossible si l'équipement fonctionne en circuit complètement clos, sans connexion avec l'extérieur qui permettrait de transférer les applications à charger. Si une telle connexion est présente, et que le système embarqué est capable de charger du code dynamiquement, plusieurs problèmes sont alors à prendre en compte :

[3]Pour faire un parallèle, nous pourrions comparer l'approche constructive à la personnalisation de systèmes à la *construction* de bâtiments, puisqu'à partir des mêmes briques on peut réaliser de nombreuses constructions... dont on verra toujours les briques. Une réorganisation de ces briques reste toutefois possible, même après construction. L'approche spécialisante s'apparente quant à elle à de la *sculpture* : on extrait une forme désirée et détaillée d'un gros bloc, duquel on aurait pu extraire n'importe quelle autre forme... mais la forme obtenue n'est pas remodelable, sauf peut-être pour produire une forme plus petite.

1. Le code chargé dynamiquement occupe une mémoire réinscriptible, contrairement au code chargé initialement qui peut être placé en ROM ;
2. Le chargement de code venant d'un agent extérieur pose de sérieux problèmes de sécurité. Il n'est en effet pas prudent d'exécuter un binaire venant de l'extérieur sans vérification, ni raisonnable d'embarquer des mécanismes complexes de vérification à l'exécution. Necula décrit comment il est possible de s'assurer de la sûreté d'un binaire chargé dynamiquement sans passer par la vérification à l'exécution grâce au mécanisme de code portant sa preuve (*Proof-Carying Code*) [Necu 96, Necu 97].
3. Un autre problème concerne la taille occupée par le code des programmes. D'une manière générale, il est souhaitable de minimiser la taille du code, que celui-ci soit chargé dynamiquement ou pas. À cet égard, la taille du code natif est généralement imposante.

Les solutions à base de code interprété, comme Java ou .Net, proposent une solution à ces deux problèmes, les programmes sous forme de bytecode ayant la réputation d'être compacts[4] et aisément vérifiables [Lero 01, Devi 02].

Pour ces raisons, Java s'impose progressivement comme solution embarquée extensible et sûre. Nous allons maintenant étudier les moyens permettant d'exploiter Java dans des équipements embarqués, puis les problèmes introduits par son mode particulier de déploiement et les solutions permettant de les contourner.

2.2 Variantes embarquées de Java

À l'origine, Java est le produit de recherches visant à développer une plate-forme adaptée à la gestion des périphériques réseau et des systèmes embarqués [Gosl 96]. Pourtant, le succès de Java est surtout venu de son utilisation dans les applications web et les serveurs d'entreprise. Ce succès s'explique par la promesse qu'un programme compilé pour la machine virtuelle Java soit en mesure de s'exécuter sur n'importe quelle plate-forme supportant celle-ci (« *Compile once, run everywhere* »).

Cependant, et malgré la portabilité accrue apportée par cette plate-forme, Java n'a pas pu s'imposer comme une solution convenant à toutes les échelles. En effet, même si la philosophie sous-jacente laisse supposer que la plate-forme Java soit utilisable sur tout type d'équipement, les détails pratiques introduisent des contraintes à sa mise à l'échelle. Un environnement d'exécution Java est en effet composé des éléments suivants :

- Une machine virtuelle, chargée d'exécuter les programmes Java,
- Un ensemble d'APIs et de bibliothèques,
- Des outils pour le déploiement des applications et la configuration du système.

Les outils et formats de déploiement de l'édition standard de Java ne sont notamment pas adaptés à toutes les situations. Comme nous allons le voir dans ce chapitre, le format `class` utilisé pour charger des classes Java est trop demandeur en ressources pour des équipements tels que la carte à puce, et des alternatives ont donc vu le jour.

Une machine virtuelle Java, en tant que telle, n'est pas grosse consommatrice de ressources. Cependant, ses performances sont grandement améliorées par l'adjonction d'outils tels qu'un compilateur *Just In Time* (JIT) [Cram 97] ou d'un ramasse-miettes perfectionné.

[4] Bien qu'il ait été démontré que le bytecode n'est pas toujours plus compact qu'une forme native (particulièrement comparé à l'ARM Thumb [Miki 01, Cost 05, Cour 06]).

De telles implémentations de la machine virtuelles améliorent grandement ses performances, mais réclament des ressources supplémentaires. Afin de garantir les meilleures performances aux équipements capables de supporter une machine virtuelle perfectionnée et de laisser la possibilité aux machines plus limitées d'exécuter des programmes Java, Sun a mis à disposition plusieurs implémentations de sa machine virtuelle :

HotSpot implémentation complète et performante des spécifications de la machine virtuelle Java ;

La *Compact Virtual Machine* (CVM) implémentation toujours complète, mais plus légère (de l'ordre du méga-octet) ;

La *Kilobyte Virtual Machine* (KVM) qui propose une plus faible empreinte mémoire (dizaines de kilo-octets) et dont certaines fonctionalités peuvent être désactivées à la compilation ;

La machine virtuelle *Java Card* qui implémente la spécification homonyme, version très dégradée de la spécification Java originale ;

Squawk [Shay 03] projet de recherche pour une machine virtuelle supportant les mêmes fonctionalités que la KVM, mais conçue pour fonctionner sur la nouvelle génération de cartes à puce.

Concernant les APIs, l'édition standard de Java propose un très large ensemble d'interfaces aux fonctionnalités du matériel sous-jacent. Celles-ci sont suffisamment génériques pour couvrir la variété des cibles supportées par Java, et peuvent être complétées par d'autres APIs couvrant des problèmes spécifiques, comme la gestion du déploiement d'applications serveurs.

Cependant, cette richesse n'est pas adaptée au monde de l'embarqué. L'étendue et la généricité des APIs de base de Java les rendent bien trop lourdes pour être utilisées sur un équipement limité. D'une part, beaucoup des fonctionnalités d'une machine de bureau ne sont pas présentes sur les équipements embarqués, ce qui rend une grande part des APIs inutiles — à l'inverse, certaines fonctionnalités spécifiques à des équipements embarqués ne sont pas gérées[5]. D'autre part, le champ d'action des équipements embarquées est en général plus restreint que celui des stations de travail : certaines fonctionnalités pourraient donc devenir plus légères si elles étaient spécialisées.

Au vu de ces différences de besoins, l'offre Java de Sun se décline en quatre versions principales :

1. Java 2, Enterprise Edition (J2EE), s'adressant aux solutions serveurs d'entreprise,
2. Java 2, Standard Edition (J2SE), l'édition de référence pour stations de travail,
3. Java 2, Micro Edition (J2ME), destiné au marché de l'embarqué, notamment de la téléphonie mobile,
4. Java Card, pour les cartes à puce.

La figure 2.3 reprend ces différentes éditions, avec les cibles qu'elles visent, leur jeu d'APIs et les implémentations de la machine virtuelle utilisées.

Dans la suite de cette section, nous détaillons certaines des solutions permettant d'embarquer la technologie Java, et nous intéresserons plus particulièrement à leur adhérence à l'édition de référence.

[5] Par exemple, aucune API de J2SE n'est capable de couvrir les besoins des capteurs.

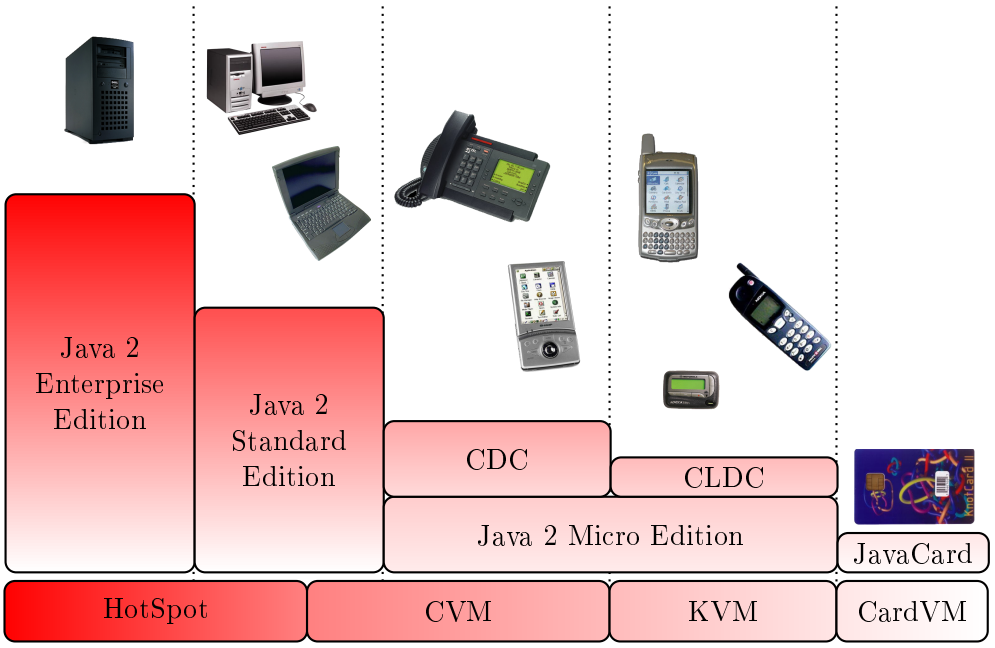


FIG. 2.3 – Les différentes déclinaisons de Java proposées par Sun.

2.2.1 Java 2, Micro Edition

Java 2, Micro Edition [J2ME] (ou J2ME) est un dérivé de Java qui cible les équipements dotés d'au moins 128 Ko de mémoire. Il est né du constat de Sun que « *One size does not fit all* » (*Une seule taille ne peut pas convenir à tout*) [J2ME 00] et se veut être une version dégradée mais embarquable de Java.

J2ME est disponible sous la forme de plusieurs *configurations*, chacune adaptée à un type d'équipement particulier. À l'heure actuelle, deux configurations sont disponibles sur le marché :

la *Connected Device Configuration* (CDC) vise les équipements connectés et sédentaires, tels que les téléphones fixes ou les stations multimédias. Elle se base généralement sur la CVM.

la *Connected Limited Device Configuration* (CLDC) est quant à elle destinée à des équipements mobiles, plus restreints et ayant une connexion réseau intermittente, comme les téléphones portables. La machine virtuelle de choix pour cette configuration est la KVM.

La configuration J2ME choisie forme le « bloc » de base sur lequel viennent se greffer des *profils*. Un profil enrichit l'API de J2ME par des fonctionnalités spécifiques à un certain type d'équipement ou une certaine famille d'applications. Par exemple, le profil MIDP (*Mobile Information Device Profile*) pour CLDC fournit des interfaces adaptées aux écrans basse résolution et aux fonctionnalités réseau limitées des téléphones portables.

Adhérence à la spécification Java Les limitations de J2ME au regard de la spécification Java standard dépendent de la configuration. CDC fournit un sous-ensemble des APIs de J2SE, ainsi que certaines APIs spécifiques. Ces dernières sont implémentées dans le paquet `javax.microedition`.

CLDC est lui-même un sous-ensemble de CDC. Les APIs sont largement plus réduites, et beaucoup de fonctionalités dépendent de la présence de différents profils. Les paquets de J2SE considérés comme coûteux sont réimplémentés par d'autres paquets incompatibles : par exemple, le paquet `javax.microedition.io` fournit un système de connexions qui remplace de manière plus légère la gestion réseau de `java.net`. La machine virtuelle de CLDC est également bridée : dans sa version 1.1, CLDC ne supporte pas les nombres flottants, la finalisation des instances, la réflexion ou la gestion de certaines erreurs d'exécution.

2.2.2 Java Card

Java Card [Java 03, Chen 00] a été conçu pour permettre de faire profiter des caractéristiques de Java (sûreté d'exécution, faible empreinte mémoire du code applicatif, etc.) à des équipements aussi restreints que la carte à puce. Les applications Java Card (ou *cardlets*) sont programmées en utilisant les outils standards de développement Java (compilateur, etc.) et sont ensuite converties vers le format de chargement `cap` pour être transférées sur la carte.

Adhérence à la spécification Java Java Card ne reproduit qu'un sous-ensemble très restreint des fonctionnalités de la machine virtuelle Java. La version 2.2 de la spécification Java Card ne supporte ainsi pas les nombres longs, les flottants, les chaînes de caractères, le multitâche ou la collection de la mémoire allouée. Elle redéfinit également la taille de certains types primitifs (les entiers sont ainsi codés sur 16 bits au lieu de 32). Enfin, elle implémente certaines fonctions spécifiques non-présentes dans la machine virtuelle originale, comme le mécanisme de pare-feu permettant d'isoler les applications.

L'API de base de Java Card est également très limitée. Seule une partie très réduite des paquets `java.*` est implémentée, la plupart des fonctionnalités étant fournies par des paquets dédiés.

2.2.3 LeJOS et TinyVM

LeJOS [LeJO] et TinyVM [Tiny] sont deux implémentations de l'environnement d'exécution Java pour la plateforme *Lego Minstorm*. Devant fonctionner dans un milieu extrêmement limité (32 Kilobits de mémoire) et spécifique (contrôle du mouvement, télécommande, caméra, senseurs, ...), ceux-ci ne proposent qu'un minuscule sous ensemble de la machine virtuelle et de l'API standard de Java (une cinquantaine de classes et interfaces de `java.*`), et complètent leurs besoins par des paquets dédiés.

2.2.4 VM★

VM★ [Kosh 05b] est un canevas dédié à la construction de machines virtuelles Java spécialisées pour réseaux de capteurs, prenant en compte les contraintes de puissance, de mémoire disponible et de consommation énergétique propres à ces équipements. Il prend en

charge la compaction des classes à déployer ainsi que la synthèse des machines virtuelles et du système d'exploitation sous-jacent.

Les machines virtuelles sont construites au cas par cas par assemblage de composants, en fonction des besoins de leurs applications et de l'équipement sur lequel elles sont déployées. Elles peuvent être mises à jour en même temps que leurs applications par édition des liens incrémentale [Kosh 05a]. Toutefois, ce type de mise à jour ne protège pas le système contre les injections de code malicieux.

2.2.5 JEPES

JEPES [Schu 03b] est une plate-forme Java embarquée visant les équipements extrêmement contraints, capable de fonctionner avec 512 octets de RAM et 4 Ko de ROM. Elle ne supporte qu'un sous-ensemble du langage Java, prend une liberté totale vis-à-vis de l'API standard qu'elle redéfinit complètement, et change comme Java Card la taille des types de base. JEPES comprend un compilateur qui prend en entrée un ensemble de classes Java et produit un binaire monolithique incluant les classes compilées et optimisées ainsi que le support d'exécution du système. Ce binaire ainsi produit est totalement fermé.

JEPES se pose comme une alternative au développement en C des applications pour équipements extrêmement contraints, et promet de combiner la sûreté de programmation de Java avec les performances et la faible empreinte mémoire d'un système natif, les classes étant en effet compilées vers du langage natif pour éviter d'embarquer une machine virtuelle dans le système final. Sur des équipements dotés de si peu de mémoire, le coût en terme d'empreinte mémoire de la machine virtuelle est en effet très difficile à amortir par le gain obtenu grâce à la forme bytecode des applications.

2.2.6 JDiet

Ce projet n'est pas à proprement parler un environnement d'exécution Java embarqué, mais permet en revanche d'embarquer des applications qui ne sont initialement pas prévues pour cela. JDiet [JDie] est un sous-projet du projet Spoon, un préprocesseur pour Java proposant un méta-modèle du langage qui permet de modifier un programme source écrit en Java. JDiet est capable de transformer un code source écrit pour J2SE en une variante adaptée à J2ME CLDC. L'approche adoptée consiste donc en une adaptation du programme à un environnement contraint au niveau du source.

Les transformations effectuées par JDiet restent cependant limitées : elles consistent, pour la plupart, à changer des conteneurs non disponibles dans CLDC par ceux qui y sont présents (par exemples, les déclarations de `Set` et de `List` sont transformées en `Vector`). Ainsi, un programme invoquant une interface de J2SE pour laquelle JDiet ne propose pas de transformation ne pourra pas être totalement converti. Un utilitaire nommé *ASMJDietVerifier* permet de vérifier qu'un code source ne référence pas d'entité non-présente dans l'API de CLDC.

2.2.7 Discussion

Dans cette section, nous avons fait un tour d'horizon des différentes solutions permettant d'utiliser Java pour le monde de l'embarqué. Nous pouvons constater que l'utilisation de

Java dans des équipements contraints et spécifiques oblige à prendre une certaine liberté par rapport à la spécification originale de Java. Les solutions Java embarquées ne sont en effet pas entièrement compatibles avec cette dernière : elles manquent d'implémenter certaines fonctionnalités, ou les redéfinissent d'une manière plus légère ou adaptée au contexte de l'embarqué. Nous pouvons ainsi constater que les implémentations telles que J2ME, Java Card ou LeJOS sont des spécialisations manuelles et arbitraires de l'implémentation J2SE originale.

Parmis les éléments les plus altérés de ces environnements embarqués figurent l'API, mais également le mécanisme de chargement des classes. Le chargement de classes Java est en effet une opération particulièrement coûteuse, et qui plus est pas très adaptée aux environnements contraints. Nous allons maintenant étudier ce mécanisme et passer en revue les solutions de déploiement alternatives proposées pour les milieux contraints.

2.3 Déploiement d'applications en Java

L'installation d'un logiciel au sein d'un environnement d'exécution constitue une opération de déploiement. En Java, le déploiement d'applications passe par le chargement de classes au sein de la machine virtuelle. Nous allons maintenant expliciter le déploiement logiciel et le faire correspondre avec le chargement de classes Java. Puis nous verrons en quoi ce modèle de déploiement n'est pas directement applicable aux systèmes embarqués et contraints, pour ensuite étudier les alternatives proposées.

2.3.1 Principes du déploiement d'applications

Le déploiement désigne l'ensemble des opérations faisant passer un composant logiciel d'un état où celui-ci est prêt à être chargé sur un système opérationnel (*ready-to-load*) à un état où le composant est prêt à s'exécuter sur le dit système (*ready-to-run*). Carzaniga *et al* définissent l'ensemble des opérations pouvant être réalisées sur un composant dans le cadre du déploiement [Carz 98] :

La *mise à disposition* (R) est décidée par le producteur du composant, et marque la naissance de ce dernier sur le marché.

La *mise en retraite* ($\overline{\mathbf{R}}$) est également une opération du producteur du composant. Elle marque la fin de sa maintenance, et sa mort du point de vue du producteur.

L'*installation* (I) consiste en l'insertion du composant dans le système cible. Cette activité peut elle-même être décomposée en deux sous activités : Le *transfert* (**T**) du composant vers le système cible, et sa *configuration* (**C**) au sein de ce dernier. L'installation d'un composant peut déclencher l'installation d'autres composants dont il dépend.

La *désinstallation* ($\overline{\mathbf{I}}$) concerne la suppression du composant du système sur lequel il a été installé.

La *mise à jour* (U) permet de remplacer un composant installé par une version plus récente. La mise à jour est une forme particulière de l'installation, et reprend ses sous-étapes de transfert et de configuration. Mais contrairement à l'installation, elle est initiée par le producteur.

L'*adaptation* (D) consiste en la modification locale du composant installé afin d'améliorer son comportement par rapport à l'environnement dans lequel il est installé.

L'*activation* (A) est la mise en service proprement dite du composant dans son environnement de fonctionnement. Le composant commence à accomplir les tâches pour lesquelles il a été écrit à partir de cette phase. Nous définissons l'état à partir duquel le composant rentre dans sa phase active comme étant son **état initial utile**.

> **Définition :** L'**état initial utile** d'un composant correspond à l'état à partir duquel il est complètement déployé sur sa cible et commence à remplir les tâches pour lesquelles il a été déployé.

la *désactivation* ($\overline{\mathbf{A}}$) replace le composant dans un état de non-fonctionnement. Il reste présent sur le système, mais n'accomplit plus ses tâches.

La figure 2.4 montre les états possibles d'un composant et les transitions autorisées entre ces états.

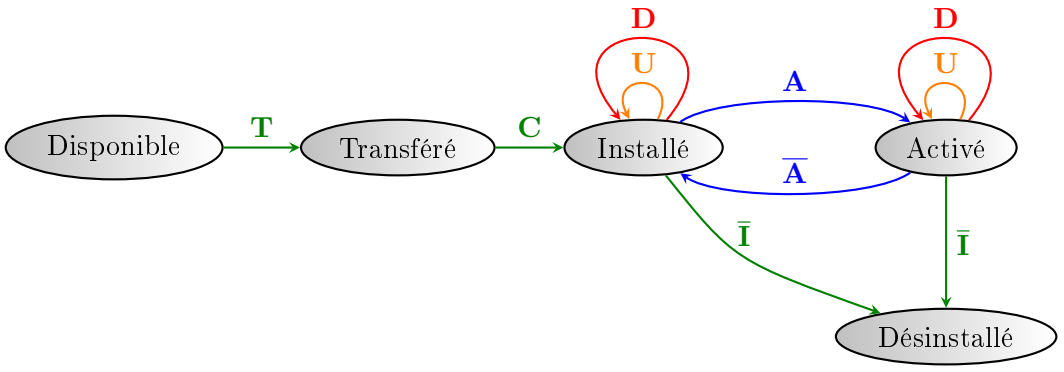


FIG. 2.4 – États possibles de déploiement d'un composant logiciel et transitions autorisés entre ces états, après mise à disposition par le producteur.

Les transitions présentées se veulent exhaustives : un gestionnaire de composants ne gère pas obligatoirement toutes les transitions montrées. Notamment, certains gestionnaires ne sont pas capables de mettre à jour ou d'adapter un composant activé. Ainsi, OSGi [OSGi 03] requiert qu'un composant actif soit désactivé avant de pouvoir être mis à jour. De même, le chargement de classes Java instancie ce modèle de manière particulière.

2.3.2 Chargement de classes Java

Le chargement de classes est au coeur de la philosophie Java. Il s'agit d'un mécanisme de déploiement au travers duquel une classe Java (le composant) est installée au sein d'une machine virtuelle (le système opérationnel). Cette opération est gérée par une hiérarchie de chargeurs de classes qui offre des garanties de contrôle et de sécurité, et permet un chargement dynamique de classes particulièrement flexible [Hall 02, Lian 98].

On distingue deux types de classes :

1. Les classes primitives ou de type tableau qui sont créées à la demande et sont utilisables immédiatement après leur création ;
2. Les classes qui ne sont ni primitives, ni de type tableau et nécessitent d'être chargées à partir d'un flux binaire (typiquement, un fichier) au format `class`.

Le chargement s'applique seulement à ce second type de classes.

La spécification de la machine virtuelle Java [Lind 99] nous apprend qu'une classe ni primitive, ni tableau, installée par un chargeur de classes passe par différentes étapes avant d'être utilisable. Elle doit être *chargée* depuis son flux binaire, *liée* aux autres classes de la machine virtuelle, et fin *initialisée*.

2.3.2.1 Chargement

Lorsqu'il est demandé à un chargeur de classes d'installer une classe donnée dans la machine virtuelle, celui-ci tente dans un premier lieu de déléguer cette opération à son parent. Si ce dernier ne peut créer la classe demandée, alors le chargeur tente de le faire lui-même, en cherchant un flux binaire correspondant au nom de la classe. Si ce flux est trouvé, une représentation spécifique à la machine virtuelle est créée à partir de ce dernier, et la nouvelle classe se voit attribuer l'état `LOADED` (chargée).

Cette opération correspond au transfert (**T**) de la classe dans la machine virtuelle.

2.3.2.2 Édition de liens

L'édition de liens correspond à la configuration de la classe nouvellement chargée afin qu'elle puisse être utilisée avec les autres classes de la machine virtuelle. Cette étape comprend trois phases :

- La vérification de la validité du typage du bytecode ;
- La préparation de l'espace nécessaire pour stocker les champs statiques de la classe et leur initialisation aux valeurs par défaut ;
- La résolution des références symboliques externes de la classe. Cette phase implique de charger les classes référencées qui ne sont pas déjà présentes dans le système.

Cette dernière phase n'est pas conditionnée par la spécification officielle, et son temps d'occurrence est laissé à l'appréciation du concepteur de machine virtuelle. On observe deux tendances quant à son implémentation. La première est de résoudre les références symboliques au cas par cas et le plus tardivement possible, c'est à dire quand l'interpréteur de bytecode rencontre une référence non-résolue. Celle-ci est alors résolue, et la classe référencée est chargée si besoin est. Cette approche implique que des classes peuvent être chargées à n'importe quel moment de l'exécution du système, et que les flux de classes chargeables soient donc disponibles pendant tout ce temps. En revanche, les références non-rencontrées par l'interpréteur ne sont jamais résolues, ce qui évite de charger des classes inutilement. Cette approche, appelée *résolution de références tardive*, est la plus populaire notamment pour les équipements dont l'espace de stockage permet de garder toutes les classes disponibles dans leur forme non-chargée.

Une alternative consiste à résoudre d'une traite toutes les références d'une classes dès que son chargement est effectué. Elle implique donc de charger toutes les classes référencées, y compris celles qui ne sont pas utilisées lors de l'exécution du code. Cette pratique peut se justifier si les chargeurs de classes n'ont accès aux flux des classes à charger que pendant un temps limité : dans ce cas, il est judicieux de charger toutes les classes atteignables tant que le flux est disponible, même s'il n'est pas certain qu'elles soient utilisées, pour assurer que le système puisse s'exécuter sans trouble. On parle ici de *résolution de références précoce*.

Le temps de déclenchement de l'édition des liens est ainsi laissé libre au programmeur. Quel que soit le moyen employé, une classe dont l'édition de liens a été effectuée atteint l'état `LINKED` (liée), cette étape étant assimilable à la configuration (**C**) d'un composant dans son système opérationnel.

2.3.2.3 Initialisation

Cette opération est la dernière avant que la classe ne devienne véritablement utilisable par la machine virtuelle et n'atteigne son état initial utile. Elle consiste en l'exécution du code des initialiseurs statiques, qui sont notamment chargés de donner aux champs statiques leurs valeurs initiales. Avant cela, la classe parente est elle-même initialisée si elle ne l'était pas déjà. Contrairement aux précédentes étapes, le temps d'occurrence de l'initialisation d'une classe est précisément spécifié : celle-ci doit être effectuée juste avant le premier *usage actif* de la classe. Un « usage actif » désigne la création d'une instance de la classe, l'invocation d'une de ses méthodes statiques ou encore l'accès à l'un de ses champs statiques non-constants. Cette exigence se justifie par le fait que les initialiseurs statiques peuvent contenir du code dont le comportement dépend de l'ordre d'initialisation des classes, et assure donc une cohérence dans le cas où la classe est utilisée sur différentes implémentations de machine virtuelle Java.

L'initialisation d'une classe correspond à son activation (**A**) du point de vue du modèle à composants. Une classe initialisée obtient l'état `READY` (prête).

2.3.2.4 Discussion du modèle de déploiement de la machine virtuelle Java

La figure 2.5 résume les différentes étapes du chargement de classes et leur correspondance avec le modèle à composants.

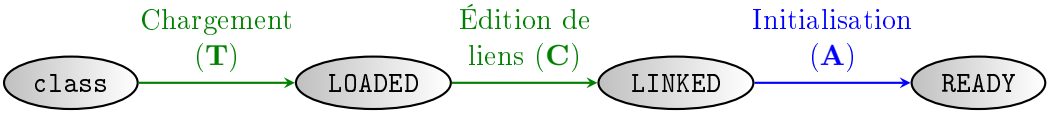


FIG. 2.5 – Le chargement de classes Java.

On notera que les opérations de désinstallation, de désactivation et de mise à jour sont absentes. L'adaptation est possible, mais du ressort de la machine virtuelle — la spécification suggère le remplacement des bytecodes pouvant entraîner une résolution de référence ou l'initialisation d'une classe par des variantes n'effectuant pas ces opérations, une fois celles-ci effectuées. L'état initial utile d'une classe correspond à son état `READY`.

On voit également que les contraintes en terme de temps de déclenchement des différentes opérations de déploiement sont quasi-nulles. Seule l'activation est obligatoirement déclenchée par le premier usage actif de la classe. Le transfert et la configuration d'une classe peuvent par contre survenir à des moments très différents selon l'implémentation de la machine virtuelle, car ces opérations n'ont pas d'effet de bord sur le comportement visible de celle-ci. Il est ainsi possible soit d'attendre qu'une classe doive être initialisée pour l'installer, soit au contraire de charger et de lier préventivement un grand nombre de classes au démarrage de la machine virtuelle.

Ce modèle de déploiement est donc suffisamment flexible pour gérer les cas où l'on souhaite limiter le nombre de classes présentes en mémoire, ou prévenir le chargement de classes pendant l'exécution du système, soit pour éviter d'impacter sur les performances, soit parce que la forme non-chargée des classes n'est pas disponible pendant l'exécution.

Cependant, le processus de chargement en lui-même réclame la résolution de références symboliques, ainsi que la transformation du format `class` vers le format interne de la machine virtuelle. Ces opérations sont coûteuses en temps d'exécution et surtout requièrent une quantité de mémoire de travail relativement importante. Pour ces raisons, le chargement complet de classes Java est très difficilement applicable aux équipements contraints en mémoire, tels que la carte à puce. Java Card propose donc un schéma de chargement de classes alternatif et plus léger pour la carte.

2.3.3 Formats pré-chargés

Afin de réduire les ressources nécessaires au déploiement d'applications, Java Card utilise un format préchargé nommé `cap`. Un fichier `cap` est créé en dehors de la carte à partir des classes d'un paquet Java par un outil nommé *convertisseur*. Celui-ci vérifie que les opérations effectuées par les classes restent dans le sous-ensemble autorisé par Java Card, crée une représentation des classes chargées (opération de transfert **T**), regroupe les *constant pools*[6] de toutes les classes en un constant pool global, résout les références internes et alloue l'espace destiné aux champs statiques des classes (opérations de configuration **C**) et enfin l'initialise avec leurs valeurs de départ (opération d'activation **A**). Un fichier au format `cap` est alors créé, qui contient cette forme pré-chargée des classes dans un état très proche de leur état initial utile.

Ce fichier prêt-à-exécuter est alors donné à la carte qui n'a plus que quelques opérations triviales à réaliser avant de pouvoir l'utiliser : copie verbatim du flux reçu, ajustement des références mémoires et résolution des références externes. Le déploiement d'applications Java Card est donc un processus de déploiement distribué, dans lequel un hôte réalise la partie la plus coûteuse du processus de déploiement avant de donner la forme pré-mâchée des classes à la carte qui n'a plus qu'à finaliser le travail (figure 2.6).

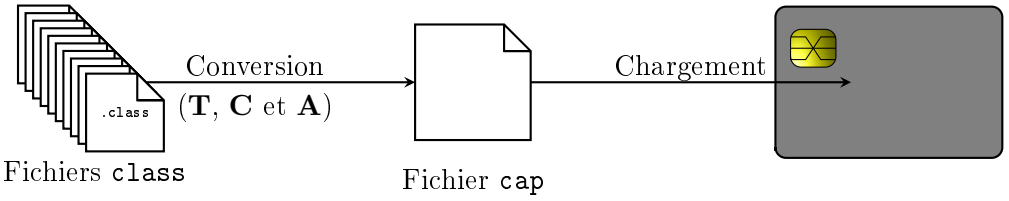


FIG. 2.6 – Le processus de chargement de Java Card.

Le format `cap` impose cependant certaines limitations. Notamment, du fait que le convertisseur ne peut exécuter de bytecode, les champs statiques ne peuvent être initialisés que par des valeurs constantes connues à la compilation. Il est impossible, par exemple, d'affecter un objet ou une valeur calculée dans les initialiseurs statiques.

[6] Le « constant pool » d'une classe est un tableau comprenant toutes les valeurs statiques (numériques, chaînes de caractères, ou référence vers des classes, des méthodes ou des champs) utilisées par cette classe.

Une autre limitation est que l'ajout dans la carte de nouvelles classes référençant des classes chargées précédemment devient plus difficile. En effet, pour permettre cela le convertisseur doit connaître le plan mémoire effectué lors du premier chargement pour ajuster les références des nouvelles classes.

L'approche de Java Card a inspiré le format JEFF [J Co 02], qui est l'adaptation du pré-chargement de classes aux autres plates-formes Java. JEFF reprend l'idée d'un format contenant des classes pré-chargées, contenant un constant pool global et fournit en plus des références symboliques des indexes de substitution. Il laisse cependant l'activation des classes à la charge de la machine virtuelle, pour ne pas avoir à imposer des initialiseurs statiques constants comme le fait le format `cap`. JEFF permet ainsi de fournir des paquets applicatifs environ deux fois plus compacts qu'un fichier `jar` et bien plus simples à charger. Cette approche est particulièrement intéressante dans le cadre des petits équipements tournant sous J2ME CLDC, ces derniers ne disposant pas toujours de la quantité de mémoire nécessaire au chargement de plusieurs classes Java.

Les formats pré-chargés permettent de pallier au manque de ressources des équipements embarqués. Cependant, quel que soit le moyen de chargement employé, l'opération finale d'installation des classes est toujours effectuée par la machine virtuelle, et les classes chargées le sont donc dans une mémoire que celle-ci peut écrire. Or, comme nous l'avons vu dans la section précédente, la mémoire réinscriptible est une denrée précieuse pour les équipements embarqués. Traditionnellement, le code applicatif est placé en ROM, ce qui n'est pas possible avec ces moyens de chargement de classes.

Il est cependant possible de placer la forme pré-chargée des classes en ROM, à condition que celle-ci soit liée à la machine virtuelle avant que cette dernière ne soit transférée vers son équipement cible. Cette pratique est connue sous le nom de *romization*.

2.3.4 Le processus de romization

La romization est un processus consistant à créer, au sein d'un environnement hôte, une image mémoire d'un système directement exécutable par un équipement cible une fois copié bit-à-bit dans la mémoire de ce dernier.

Dans le cas des programmes Java, la romization est l'acte d'effectuer toutes les opérations de déploiement des classes en dehors de la machine virtuelle, à la manière des formats pré-chargés, puis de lier statiquement la forme pré-chargée des classes avec le binaire de la machine virtuelle [Gans 03].

Cette opération présente deux avantages :

- Premièrement, comme pour les formats pré-chargés, elle permet de libérer la cible des opérations de chargement des classes Java qui sont, comme nous l'avons vu, très coûteuses et souvent irréalisables sur un équipement contraint ;
- Deuxièmement, certains éléments des classes ainsi chargées (en particulier les méta-données et le bytecode) atteignent leur état final en dehors de la cible. Étant constantes pendant toute l'exécution du système, elles peuvent donc être placées en ROM.

De par ces propriétés avantageuses, la romization est une opération très courante, pour ne pas dire nécessaire, dans le déploiement d'un système Java embarqué. Les variantes embarquées de Java offrent généralement toutes un moyen de romizer des classes.

Dans J2ME J2ME est fourni avec un outil nommé *JavaCodeCompact* (JCC) qui permet de créer une forme pré-chargée des classes qui lui sont fournies. L'ensemble de ces classes doit obligatoirement former une closure transitive, c'est-à-dire inclure toutes les classes qu'il référence.

JCC effectue les opérations de chargement et d'édition des liens sur les classes, mais ne peut pas les initialiser car cette opération nécessite que la machine virtuelle soit démarrée. Il est également capable de purger les classes pré-chargées des champs et méthodes jugées inutiles à partir d'une liste fournie[7].

La forme pré-chargée des classes traitées par JCC est produite sous la forme d'un fichier C reprenant la structure interne des classes chargées dans la KVM. Ce fichier est alors compilé en même temps que la KVM et lié avec elle de telle sorte que les classes romizées apparaissent comme chargées dès le démarrage de celle-ci (figure 2.7).

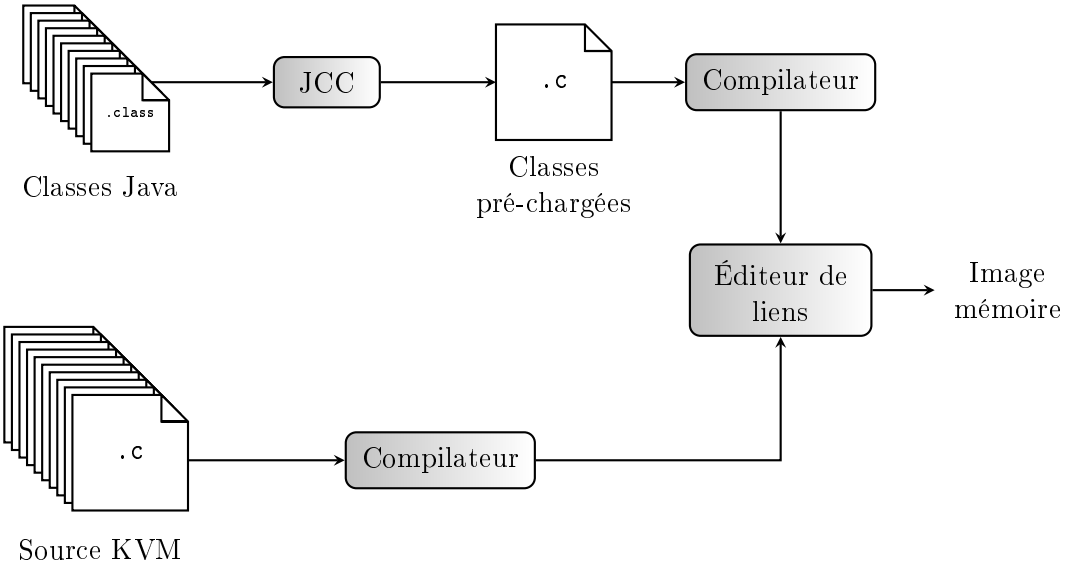


FIG. 2.7 – Le processus de romization de J2ME.

Dans Java Card Le format `cap` est déjà un format prêt à l'exécution : il est donc possible de lier un ou plusieurs fichiers `cap` avec la machine virtuelle Java Card avant l'activation de celle-ci. Dans ce cas, les classes ainsi liées peuvent, tout comme les classes romizées de J2ME, être placées en ROM.

Dans VM★ et JEPES VM★ permet également de placer des classes pré-chargées au sein d'une machine virtuelle avant son installation dans l'équipement cible. En fait, c'est généralement ce qui se passe étant donné que les applications à exécuter sont typiquement envoyées au capteur avec leur support d'exécution.

JEPES, en produisant un binaire monolithique comprenant support d'exécution et classes

[7] Un outil permet de calculer une liste de tels éléments à partir d'un point d'entrée, cependant nous n'avons pu en trouver aucune trace.

chargées, effectue ainsi également une opération de romization. La forme compilée des classes correspond en effet à leur forme chargée pour cette plate-forme.

2.4 Limites dans le déploiement de systèmes Java embarqués

Nous avons, dans ce chapitre, dressé un état de l'art de l'emploi de Java dans les systèmes embarqués. Nous avons tout d'abord caractérisé les systèmes embarqués et leurs contraintes, et défini la cible privilégiée de notre travail, à savoir les petits objets destinés à l'informatique enfouie, produits en masse et fortement contraints. Ceux-ci nécessitent des systèmes parfaitement adaptés à l'accomplissement de leurs tâches, et donc produits au cas par cas.

Cette production sur-mesure peut se faire manuellement, mais cette approche est très lourde et sujette aux erreurs de conception. Ceci nous a poussé à étudier les différents moyens d'ingénierie logicielle permettant d'obtenir un système embarqué tout en laissant l'opportunité de réutiliser le code produit. Nous avons relevé deux types d'approches : l'approche constructive, qui consiste à construire un système adapté à partir de briques de base, laisse une grande liberté dans la spécialisation du système, y compris après son déploiement. Mais elle impose des abstractions propres à chaque système qui rendent difficile toute compatibilité au niveau applicatif. L'approche spécialisante, au contraire, extrait une variante embarquable et dédiée à l'exécution exclusive d'un ensemble limité de tâches à partir d'un système plus générique. Elle permet de décliner le même système en une multitude de variantes tout en gardant la même abstraction au niveau applicatif. En revanche, les spécialisations effectuées ne sont pas réversibles et les systèmes ainsi produits sont très difficiles à étendre après leur activation.

C'est ce dernier type de spécialisation qui est effectué sur les variantes embarquées de Java, telles que J2SE et Java Card. En effet, elles sont conçues à partir d'une version révisée de la spécification J2SE. Cette divergence précoce avec l'édition standard de Java les rend incompatibles avec celle-ci, brisant ainsi la promesse initiale de Java, « *compile once, run everywhere* ».

Les variantes embarquées de Java ont également à faire face à d'autres problèmes liés au chargement des classes : d'une part, le chargement du format `class` est une opération trop lourde pour la plupart des équipements contraints. D'autre part, les classes chargées le sont par le système Java lui-même, ce qui interdit de placer les applications dans une mémoire à lecture seule. Le premier problème est résolu en effectuant les opérations les plus coûteuses du chargement de classes hors du système cible : ainsi, le système reçoit une forme pré-chargée des classes qu'il peut utiliser pratiquement sans autre manipulation. Le second problème est contourné en liant cette forme pré-chargée avec la machine virtuelle avant son démarrage, ce qui permet de la placer en ROM. Cette dernière opération porte le nom de *romization*.

Les solutions de romization que nous avons passées en revue sont ainsi correctement capables de pré-charger des classes. Cependant, le déploiement d'applications Java ne se limite pas à cette seule opération. Par exemple, une application Java Card n'atteint son état initial utile qu'après invocation de sa méthode `install()`. Le déploiement des applications est donc effectué de façon incomplète, ce qui se ressent sur l'empreinte de l'image mémoire produite puisque celle-ci doit inclure les mécanismes permettant au système de compléter son déploiement.

Dans le prochain chapitre, nous présentons une nouvelle architecture de romization permettant de s'affranchir de cette dernière limitation. Puis, dans le chapitre 4, nous nous appuyons sur cette architecture pour élaborer un procédé de spécialisation tardive permettant d'obtenir des systèmes Java embarqués fortement spécialisés, mais acceptant des applications conformes à la spécification standard de Java.

Troisième Chapitre

Une nouvelle architecture de romization

« Il faut inventer en même temps que l'on apprend. »

Plutarque.

Dans le chapitre précédent, nous avons établi un état de l'art des systèmes Java embarqués ainsi que des solutions de romization, procédé permettant de pré-charger des classes dans la machine virtuelle Java avant le démarrage de celle-ci. Au cours de cette étude, nous avons observé deux points préjudiciables à l'utilisation de Java dans les systèmes embarqués et contraints :

1. Les solutions Java embarquées sont des versions dégradées de l'environnement Java standard incompatibles avec celui-ci ;
2. La romization ne permet pas d'effectuer le déploiement complet du système hors-ligne[1].

Dans le présent chapitre, nous nous attardons sur ce dernier problème, ce qui nous permettra d'adresser le premier au chapitre suivant. La romization est un processus qui a été développé de façon empirique et de manière à résoudre des problèmes technologiques, sans réelle approche scientifique. Nous allons donc commencer par définir clairement la notion de romization, puis à la lumière de cette définition nous mettrons l'accent sur le pourquoi des lacunes des solutions existantes. Ceci nous amènera à évaluer le véritable potentiel du processus de romization. La deuxième section du chapitre propose une nouvelle architecture de romization, permettant d'effectuer le déploiement complet du système hors-ligne et de capturer une image mémoire de celui-ci à n'importe quel moment de son déploiement ou de son exécution. Enfin, la dernière section donne les détails de notre implémentation de cette architecture.

3.1 La romization : un déploiement de système *in-vitro*

Commençons par définir la notion de romization de manière générale. Cette section montre les convergences qui existent entre le processus de déploiement d'un composant logiciel et celui de romization. Ceci nous permet de déterminer que la romization est un cas particulier du déploiement logiciel.

[1] Nous employons les termes *en ligne* et *hors-ligne* pour désigner le cadre dans lequel se déroule une action sur le système : respectivement, au sein de son environnement normal d'exécution et hors de celui-ci.

3.1.1 La romization, une opération de déploiement logiciel

Comme nous l'avons vu dans le chapitre précédent, la romization des classes Java a été introduite afin de pallier deux problèmes :

1. La nécessité de pouvoir placer la forme chargée des classes Java en ROM,
2. Les fortes contraintes des équipements embarqués : ceux-ci ne disposent généralement ni de la mémoire ni de la puissance de calcul nécessaires pour effectuer eux-mêmes le chargement de classes.

La romization telle qu'effectuée par J2ME consiste donc à pré-charger les classes que l'on veut voir figurer dans le système à l'aide d'un outil externe, qui génère une image mémoire de leur forme chargée pour la machine virtuelle embarquée. Cette image mémoire est ensuite liée statiquement à la machine virtuelle, et peut ainsi être placée en ROM. Les classes pré-chargées sont alors utilisables par la machine virtuelle embarquée dès le démarrage de celle-ci.

Or, comme nous l'avons également vu en 2.3.2, le chargement de classes Java est aussi et surtout une opération de déploiement d'un composant logiciel (la classe) au sein d'un système (la machine virtuelle). À ce titre, nous pouvons affirmer, de manière plus générale, que la romization est une opération de déploiement des classes dans la mémoire morte d'un support informatique qui n'est pas encore fonctionnel. Nous qualifions cette activité d'opération de « pré-déploiement » car elle est préalable au démarrage du système auquel elle est destinée.

> **Définition :** La romization est le processus de pré-déploiement par lequel un composant logiciel est déployé à l'aide d'un outil spécifique (le *romizer*), dans un *environnement hôte*, avant d'être capturé et transféré vers son *environnement cible* sur lequel il reprend son exécution.

Ce pré-déploiement peut être équivalent au pré-chargement de classes, mais n'est en aucun cas limité à cette seule opération. Dans le cas particulier du pré-chargement de classes, le romizer effectue hors-ligne une opération que réalise normalement le système, à savoir le chargement des classes.

3.1.2 Limites du pré-chargement de classes

Les schémas de romization étudiés lors du chapitre précédent se contentent de pré-charger des classes : cette opération est effectuée alors que le système n'est pas encore démarré. Cela est valide par rapport à la spécification Java car celle-ci n'impose aucune contrainte temporelle sur le déclenchement des opérations de chargement et d'édition des liens des classes. Comme nous l'avons vu en 2.3.2.4, une classe peut être chargée bien avant son utilisation ou au contraire au dernier moment — et donc, pourquoi pas, avant même que le système ne soit démarré (figure 3.1).

Il n'en va cependant pas de même avec l'activation de la classe. Cette étape, qui marque la fin du déploiement et l'arrivée de la classe dans son état initial utile, est strictement spécifiée et son déclenchement ne peut pas se produire avant que le système ne soit lui-même démarré[2]. Le chargement effectué pendant la romization de J2ME n'est donc qu'un déploiement partiel.

[2] Java Card fait exception à cette règle car l'activation des classes se fait à partir de données strictement constantes.

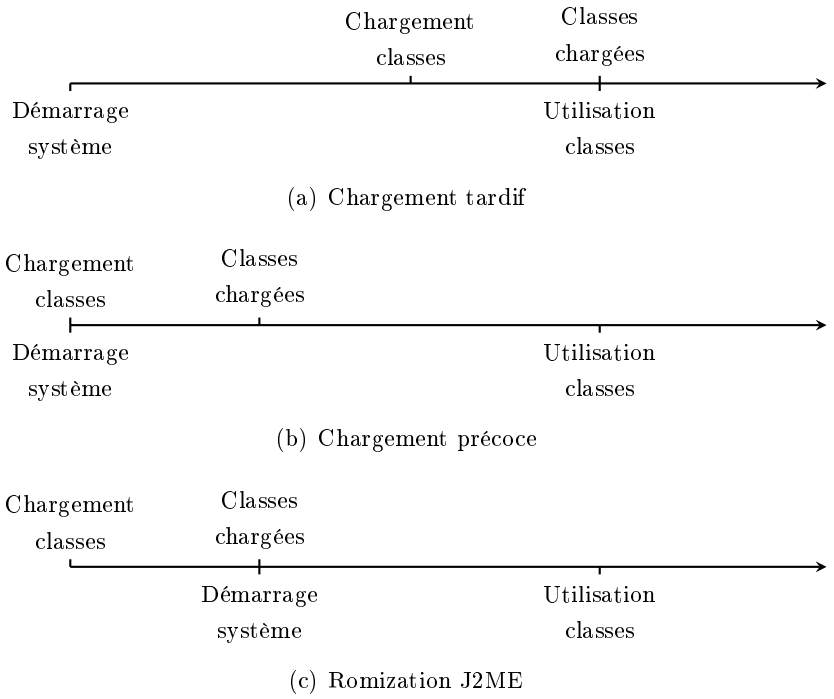


FIG. 3.1 – Chronologie du démarrage de la machine virtuelle Java dans les cas du chargement tardif, du chargement précoce et de la romization de J2ME.

Cette contrainte n'est pas très forte du point de vue des classes seules. L'exécution des initialiseurs statiques n'est pas une opération particulièrement coûteuse pour l'équipement cible. En revanche, le fait que le système ne soit pas démarré ne lui permet pas d'atteindre son état initial utile.

Une application pré-chargée peut ainsi avoir à effectuer certaines opérations d'initialisation avant d'atteindre son état initial utile, comme allouer de la mémoire ou pré-calculer certaines valeurs. Ces opérations, qui requièrent que le système soit démarré, ne peuvent pas être effectuées par un simple pré-chargeur de classes. Pourtant, elles correspondent à des opérations d'initialisation qui gagneraient à être effectuées hors-ligne. Parfois même, la mémoire requise pour les effectuer est trop importante pour qu'elles le soient sur l'équipement cible, ce qui les rend non-embarquables.

Voyons deux exemples d'applications embarquées pour lesquelles le pré-chargement de classes seul ne constitue pas un déploiement optimal.

3.1.2.1 Exemple 1 : `TransitApplet`

`TransitApplet` est une application carte fournie dans les exemples du kit de développement Java Card[3]. Elle est prévue pour être embarquée sur une carte à puce et permet de gérer des transactions sur un compte présent dans la carte, de manière similaire aux cartes Monéo. Chaque carte porte un numéro d'identifiant unique, et les accès au compte sont

[3] http ://java.sun.com/products/javacard/

conditionnés par une authentification à clé partagée entre la carte et l'hôte.

Le listing 3.1 montre le code d'installation de l'applet. Conformément à la spécification de Java Card [Java 03], la méthode `install()` est appelée par la carte pour installer l'applet. Celle-ci construit une instance de `TransitApplet` et passe le tableau des données d'initialisation au constructeur, qui y lit l'identifiant de la carte ainsi que les données servant à construire la clé statique. Cette opération est réalisée par le fabricant, avant remise de la carte au client.

Les membres `uid` et `staticKey` restent strictement constants après cette installation : ces données sont initialisées une fois pour toutes pendant le déploiement. Mais la méthode `install()` étant exécutée sur la carte, elles doivent être placées dans une mémoire inscriptible par cette dernière, quand bien même elles sont strictement constantes une fois la phase d'initialisation de l'applet terminée. Si tout le déploiement de l'applet, incluant l'exécution de la méthode `install`, s'était effectué hors-ligne, les données référencées par `uid` et `staticKey` auraient pu être placées en ROM.

Cette applet est un exemple typique du gain obtenu à déployer une application entièrement hors-ligne. Mais il existe d'autres cas d'applications où le résultat optimal n'est obtenu que si l'application est également démarrée et utilisée partiellement hors-ligne.

3.1.2.2 Exemple 2 : `PhotoCard` en mémoire à lecture seule

Cette application carte est également fournie avec le kit de développement Java Card. Elle offre un service de gestion de photos : l'utilisateur peut consulter les photos présentes sur la carte, en ajouter ou en effacer.

Une variante d'utilisation de cette application serait de fournir une carte remplie de photos en lecture seule, à la manière d'un Photo-CD. Les photos sont alors placées en ROM, ce qui permet d'une part de réduire grandement le coût de production des cartes, et d'autre part d'augmenter leur capacité mémoire du fait que le point mémoire[4] de la ROM est très inférieur à celui d'une mémoire réinscriptible.

Seulement, il est impossible de fournir un ensemble initial de photos à charger à l'application avant son démarrage. En effet, la procédure d'installation ne permet pas de charger un ensemble initial de photos. Il est donc nécessaire d'activer l'applet et de charger des photos en utilisant les méthodes prévues à cet effet pour obtenir un premier ensemble utilisable (listing 3.2).

L'utilisation de `PhotoCard` pour produire une carte dans laquelle les photos seraient stockées dans une ROM implique donc non seulement d'être capable de déployer l'application entièrement hors-ligne, mais également de commencer son exécution hors-ligne afin de charger l'ensemble de photos en utilisant les méthodes prévues à cet effet. Les photos ainsi chargées pourraient alors être logées en ROM lors du transfert du système dans sa mémoire physique.

Les exemples que nous avons étudiés proviennent de Java Card, mais le problème reste actuel pour tous les gestionnaires de composants, y compris les plus complexes.

[4] Le *point mémoire* désigne l'occupation physique de la cellule de base de la mémoire, utilisée pour coder un *bit*. Voir 2.1.2

Listing 3.1 – Code d'installation de l'applet `TransitApplet`.

```
1  /**
2   * Unique ID
3   */
4  private byte[] uid;
5
6  /**
7   * DES static key, shared b/w host and card
8   */
9  private DESKey staticKey;
10
11 public static void install(byte[] bArray, short bOffset, byte bLength) {
12   // Create a Transit applet instance
13   new TransitApplet(bArray, bOffset, bLength);
14 }
15
16 /**
17  * Creates a new Transit applet instance.
18  *
19  * @param bArray
20  *             The array containing installation parameters
21  * @param bOffset
22  *             The starting offset in bArray
23  * @param bLength
24  *             The length in bytes of the parameter data in bArray
25  */
26 protected TransitApplet(byte[] bArray, short bOffset, byte bLength) {
27   // Create static DES key
28   staticKey = (DESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_DES,
29     KeyBuilder.LENGTH_DES, false);
30
31   // Retrieve the UID
32   uid = new byte[UID_LENGTH];
33   Util.arrayCopy(bArray, bOffset, uid, (short) 0, UID_LENGTH);
34   bOffset += UID_LENGTH;
35
36   // Retrieve the static key data
37   staticKey.setKey(fixParity(bArray, bOffset, LENGTH_DES_BYTE),bOffset);
38   bOffset += LENGTH_DES_BYTE;
39 }
```

Listing 3.2 – La phase d'installation de `PhotoCard` ne permet pas de pré-charger des photos destinées à être placées en ROM.

```
1  public static void install(byte[] aid, short s, byte b) {
2    new PhotoCardApplet();
3  }
4  ...
5  public void loadPhoto(short photoID, byte[] data,
6    short size, short offset, boolean more)
7  ...
8  public void deletePhoto(short photoID)
9  ...
10 public byte[] getPhoto(short photoID, short offset, short size)
```

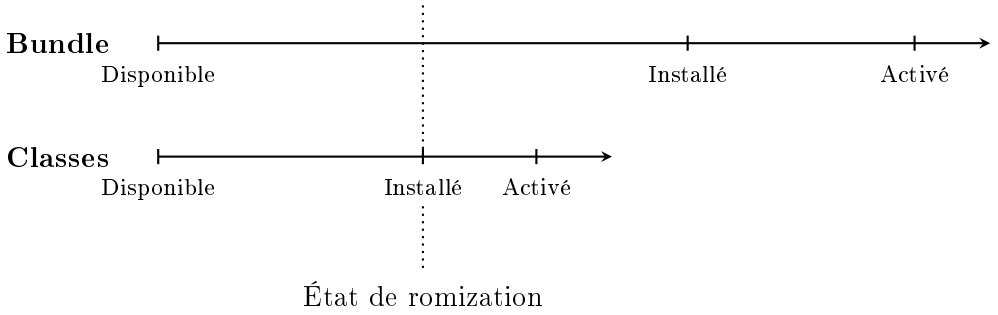


FIG. 3.2 – L'état de déploiement d'un composant de haut niveau (ici, un bundle OSGi) lors de la romization est moins avancé que l'état des sous-composants dont il dépend.

3.1.2.3 Un problème commun à tous les gestionnaires de composants

Dans le même esprit, l'utilisation de OSGi dans les équipements contraints est fortement compromise par l'impossibilité de démarrer le système hors-ligne. Tout comme Java Card, OSGi est basé sur le chargement de classes Java. Les classes d'un *bundle*[5] peuvent donc être pré-chargées, cependant son installation requiert également l'exécution d'une méthode d'installation. Là encore, cette opération se révèle impossible tant que le système n'est pas démarré. Un bundle OSGi ne peut ainsi être déployé que de manière très partielle. Il s'avère que plus l'on se base sur des mécanismes de déploiement haut niveau, qui dépendent de l'activation du système, moins on est susceptible d'atteindre l'état initial utile du composant par simple pré-chargement de classes (figure 3.2).

Pourtant, nous savons que la romization est un acte de pré-déploiement effectué hors-ligne. Le pré-chargement de classes fait partie de cet acte, mais n'en constitue pas la totalité. Il n'y a donc pas de raison pour que le romizer ne considère pas le système dans son ensemble lors de la romization, et ne permette pas son exécution hors-ligne.

[5] « *Bundle* » est le terme employé par OSGi pour désigner un composant déployable.

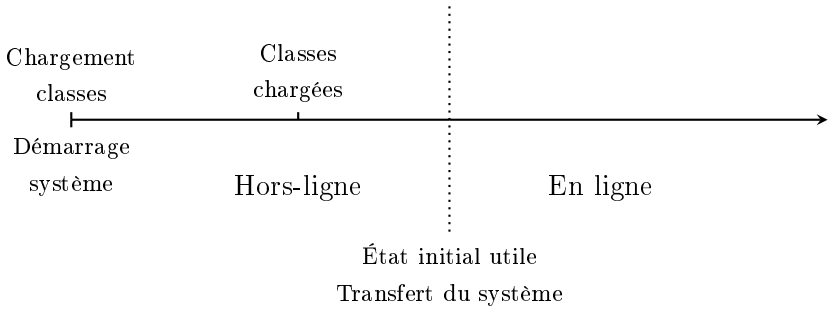


FIG. 3.3 – La romization, un déploiement *in-vitro* de système embarqué.

3.1.3 Démarrage hors-ligne du système

Le pré-chargement de classes seul ne permet de réaliser hors-ligne qu'une partie très limitée du déploiement du système : nous avons vu dans la section précédente que l'état initial utile du système ne pouvait pas être atteint dans ces conditions. Il en est ainsi car les schémas de romization classiques ne sont pas en mesure de réaliser l'ensemble des opérations qui peuvent être demandées au système final.

Afin de pouvoir compléter le déploiement, il est nécessaire de se baser sur un schéma permettant de démarrer et de commencer l'exécution du système. Un tel schéma implique l'utilisation d'un environnement d'exécution virtuel, de telle sorte que le système puisse être démarré hors-ligne. Il est alors en mesure d'effectuer lui-même les opérations de déploiement de composants en son sein, au lieu de se reposer sur des mécanismes de pré-chargement. C'est alors une fois que le système a atteint l'état désiré comme état initial dans l'équipement embarqué qu'une image mémoire de celui-ci est capturée puis transférée.

Selon ce modèle, l'exécution du système se fait donc en deux phases : une première phase hors-ligne, dans l'environnement d'exécution virtuel, puis une reprise dans l'environnement d'exécution réel, avec entre les deux phases une capture du système et son transfert d'un environnement à l'autre. Du point de vue du système, il ne doit y avoir aucune différence entre ces deux environnements, ni de discontinuité ressentie dans l'exécution.

Ainsi, nous pouvons comparer le processus de romization au développement *in-vitro* d'un organisme vivant, qui est ensuite transféré vers son milieu naturel pour continuer son évolution. Avec le processus de romization, le système commence son exécution dans un environnement « de laboratoire » qui n'est pas son environnement réel. Ce n'est qu'après avoir atteint la maturité désirée qu'il est transféré vers son environnement d'exécution réel (figure 3.3).

Nous proposons, dans les sections suivantes, une architecture de romization pour systèmes Java embarqués basée sur ce modèle.

3.2 Proposition d'architecture de romization

À la lumière de notre définition du processus de romization, nous pouvons extraire certaines des propriétés que doit posséder un environnement de romization capable d'assumer le déploiement d'applications dans son ensemble :

Propriété 1 : L'exécution du système dans un environnement d'exécution virtuel, extérieur à l'environnement d'exécution réel ;

Propriété 2 : La capacité pour l'environnement d'exécution virtuel à produire une image mémoire du système à n'importe quel moment de l'exécution de ce dernier ;

Propriété 3 : L'affectation d'une mémoire de destination aux à chaque entité du système présente lors du déploiement hors-ligne ;

Propriété 4 : L'adaptation éventuelle du système à son exécution future.

La propriété 1 découle directement de la section 3.1.2. Nous y avons montré que le déploiement d'un logiciel ou d'un simple composant peut consister en l'exécution arbitraire de n'importe quelle partie du logiciel déployé, ce qui requiert la capacité à exécuter le système. La propriété 2 est une conséquence du schéma de fonctionnement d'un romizer (produire une image mémoire d'un système déployé) et de la propriété 1, qui peut mettre le système dans n'importe lequel de ses états atteignables. Les propriétés 3 et 4 sont la reprise des propriétés premières d'un romizer, décrites en 2.3.4.

Commençons par voir quelles sont les conséquences de l'inclusion d'un environnement d'exécution virtuel dans le processus de romization.

3.2.1 Un environnement d'exécution virtuel dans le romizer

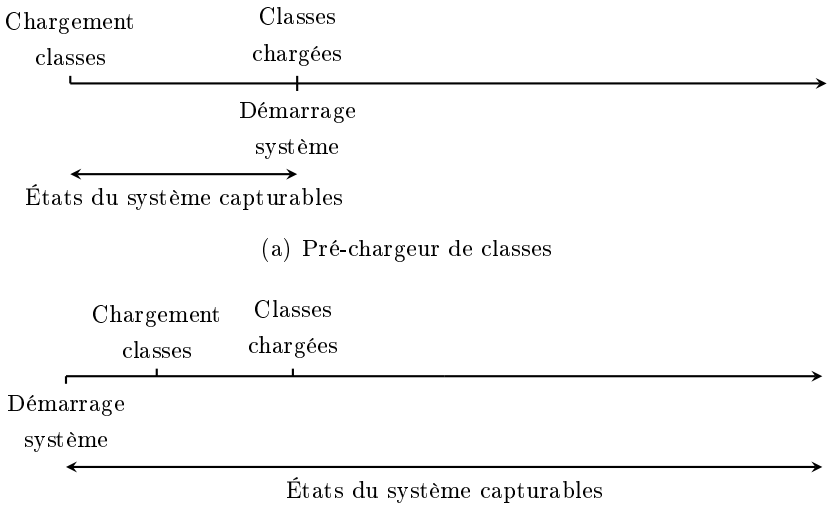
La décision d'inclure un environnement d'exécution complet dans la chaîne de romization bouleverse la façon dont celle-ci s'effectue. Afin de pouvoir exécuter le système, l'environnement de romization doit incorporer une machine virtuelle entièrement fonctionnelle, ainsi qu'une API Java permettant au système de fonctionner.

L'API Java embarquée utilisée dans l'environnement d'exécution réel est réutilisable par l'environnement d'exécution virtuel du romizer. C'est d'ailleurs indispensable pour qu'il y ait cohérence entre les deux environnements d'exécution. L'environnement d'exécution virtuel doit alors implémenter les services natifs requis par l'API, comme ses méthodes natives.

Le propre de la machine virtuelle Java est d'exécuter les applications qui lui sont soumises, mais dans le cadre de la romization, le besoin se fait plus général. Tout d'abord, il importe de pouvoir contrôler le niveau de déploiement et d'exécution effectué hors-ligne. L'exécution du système doit par conséquent pouvoir s'arrêter une fois que ce dernier a atteint l'état initial désiré sur l'équipement cible, par exemple par l'apposition de points d'arrêts à la manière d'un débogueur. Il est également nécessaire de pouvoir configurer certaines caractéristiques du système ou des applications déployées hors-ligne (par exemple, finaliser le chargement de certaines classes ou assigner une adresse IP qui restera statique pendant toute la durée de vie du système). À des fins de configuration, la personne en charge du pré-déploiement peut pour ce faire être amenée à soliciter de manière totalement arbitraire n'importe quel service déployé dans l'environnement virtuel, ce que l'outil de romization doit permettre.

Cette liberté dans l'exécution hors-ligne du système permet d'étendre l'ensemble des états capturables du système à tous les états que celui-ci peut atteindre au cours de son exécution. La figure 3.4 compare l'ensemble des états capturables selon le schéma de romization utilisé.

Il est important de prendre conscience que le fait d'utiliser un environnement d'exécution virtuel dans le romizer éloigne radicalement son mode de fonctionnement du schéma de démarrage particulier de la figure 3.1(c), pour reprendre un schéma plus classique dans



(a) Pré-chargeur de classes

(b) Environnement d'exécution virtuel

FIG. 3.4 – États du système capturables dans un schéma de romization de type pré-chargeur et environnement d'exécution virtuel.

lequel le système est tout d'abord démarré et charge lui-même les classes qu'il va exécuter (figure 3.1(a)).

Une autre conséquence importante de ce mode de fonctionnement est que l'ensemble des données manipulées hors-ligne devient beaucoup plus important. Alors qu'il suffit de capturer et de transférer les classes chargées par un pré-chargeur, la migration de l'état de l'environnement d'exécution virtuel requiert de capturer l'état de tout le système en cours d'exécution.

3.2.2 Capture et migration de l'état du système Java

L'environnement d'exécution Java offre des conditions qui facilitent la capture de certains aspects de l'état du système. Pour cette raison, il est populaire dans les applications de migration de processus, comme les agents mobiles [Wong 99]. Cependant, les mécanismes de mobilité décrits dans la spécification de Java [Lind 99] ne permettent pas une migration totale du système. Dans cette section, nous allons passer en revue les travaux effectués sur la mobilité d'applications Java, puis définir la notion de mobilité système que nous emploierons dans notre architecture de romization.

3.2.2.1 Migration faible et migration forte

Différents niveaux de migration de tâches ont été définis dans la littérature [Fugg 98, Cabr 00]. La capacité à transférer du code ainsi que des données d'initialisation d'un système à un autre est appelée *migration faible*. Dans cette forme de migration, le code migré commence son exécution à partir de son point d'entrée initial — ce type de migration ne permet pas de capturer l'état d'exécution d'une tâche et de la reprendre à partir de ce point. La migration faible est illustrée par des outils tels que le démon `rshd` (permettant d'exé-

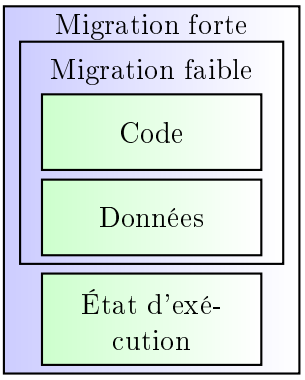


FIG. 3.5 – La migration forte est une forme de migration faible, à laquelle on ajoute la capacité à transférer l'état d'exécution de la tâche et à la reprendre à partir de cet état.

cuter un script `sh` ainsi que ses paramètres d'exécution sur une autre machine), le langage PostScript [Inc 99] utilisé sur une imprimante compatible, ou l'extension du langage TCL TACOMA [Joha 95].

Lorsqu'une tâche en cours d'exécution est entièrement migrée d'un système à un autre, avec ses données associées et son état d'exécution, la migration est alors appelée *migration forte* [Funf 98]. Cette migration est transparente pour la tâche migrée, la capture de l'état et le transfert étant assurés par le système.

Nous voyons ici qu'une migration forte est une forme de migration faible, à laquelle on ajoute la capacité à transférer l'état d'exécution d'une tâche et à la reprendre à partir de cette état sur la cible (figure 3.5).

Voyons comment ces différents niveaux de mobilité s'expriment en Java.

3.2.2.2 Mobilité des objets Java

La mobilité la plus naïve en Java est celle qui consiste à déplacer un objet depuis un système vers un autre. La sérialisation étant l'une des fonctionnalités exposées par l'API [Sun 04], la capture et la restauration d'objets est une tâche fondamentale de toute machine virtuelle Java. La structure simple des objets Java rend cette tâche triviale : les données d'un objet sont sa classe et les valeurs de ses champs (ou pour les objets de type tableau, les valeurs de ses cellules) [Rigg 96]. En fonction de l'implémentation de la machine virtuelle, d'autres données internes peuvent également exister.

Migrer un objet Java d'un système à un autre consiste donc à le sérialiser depuis la source puis à le désérialiser dans la destination. Cependant, toutes les données d'un système Java ne sont pas modélisées sous forme d'objets et ne peuvent donc pas être migrées de cette manière (figure 3.6). Ces données internes à la machine virtuelle doivent être capturées d'une manière spécifique. Parmi elles, on identifiera, en fonction de l'implémentation de la machine virtuelle :

Les classes chargées qui sont souvent créées et chargées par une fonction native du chargeur de classes. Les méthodes des classes de réflexion ne servent alors que d'interfaces publiques à ces données internes ;

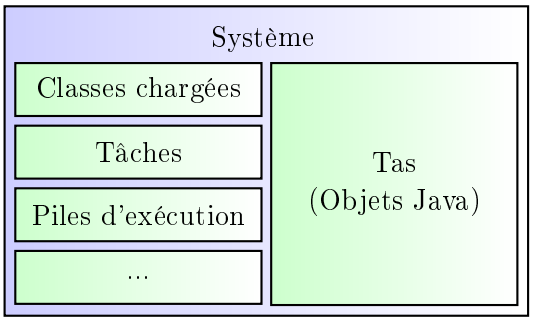


FIG. 3.6 – L'état d'un système Java comprend le tas d'objets, mais également d'autres structures internes.

Les tâches sont fréquemment associés aux tâches ou processus natifs du système d'exploitation sous-jacent, quand il existe ;

Les piles d'exécution font partie de l'état des tâches mais relèvent d'un cas particulier. Elles peuvent être soit complètement découplées des piles d'exécution du système sous-jacent, soit mélangées à celles-ci. Dans tous les cas, leur état complet n'est pas accessible aux programmes Java ;

Les moniteurs sont généralement associés aux verrous natifs de l'éventuel système sous-jacent ;

Toute autre donnée relative à l'état de la machine virtuelle par exemple l'état de l'ordonnanceur de tâches, du gestionnaire mémoire, ... en fonction de l'implémentation de celle-ci.

La quantité de données internes est ainsi totalement dépendante de l'implémentation de la machine virtuelle. Mais quelle que soit l'implémentation, ces données spécifiques font toujours partie intégrante de l'état du système. Leur capture et leur restauration peuvent alors devenir problématiques étant donné que les cibles sur lesquelles s'exécutent les environnements source et destination sont potentiellement très différentes. Nous allons voir comment la littérature a adressé ce problème.

3.2.2.3 Mobilité des classes Java

La migration d'une classe d'un système Java à un autre consiste, dans une forme naïve, à recharger la classe dans le système cible. C'est le mécanisme qui est utilisé lors de la désérialisation d'un objet Java dont la classe n'est pas encore chargée au sein du système cible. Cependant, cette forme de migration ne prend pas en compte les valeurs des champs statiques.

Dans le domaine de la romization, le pré-chargement de classes (voir 2.3.4) est également une forme de migration des classes Java, cette fois-ci dans leur forme chargée, et avec leurs champs statiques initialisés.

La capaciter à migrer des objets et des classes Java est une instanciation de la migration faible : les classes migrées correspondent au code (méthodes) et à une partie des données de la tâche (champs statiques), tandis que les objets Java sont considérés comme des données.

L'obtention d'une migration forte en Java passe en plus par la capture de l'état complet d'une tâche en cours d'exécution sur le système.

3.2.2.4 Mobilité des applications Java

La mobilité des tâches en Java est largement sujette à l'implémentation qui en est faite dans la machine virtuelle. L'API Java ne fournit en effet pas de mécanisme permettant de migrer l'état d'exécution d'une tâche, contrairement à ce qui est fait pour les objets. Différents travaux ont été effectués afin de permettre la mobilité forte des applications Java.

Truyen *et al.* [Truy 00] proposent une solution reposant sur une couche d'intergiciel et modifiant le code applicatif au niveau du bytecode, afin d'insérer des blocs chargés de sauvegarder et de restaurer l'état de la tâche. Cette solution ne nécessite pas de modifier la machine virtuelle exécutant les applications, mais requiert une modification du bytecode applicatif et l'utilisation d'une couche logicielle particulière.

Sara Bouchenak [Bouc 00, Bouc 01a, Bouc 01b] a dans ses travaux de thèse intégré des mécanismes de migration de tâches Java en cours d'exécution dans la machine virtuelle Java de Sun. Ce travail couvre la migration du contexte complet de la tâche, c'est à dire sa pile d'exécution, ses classes ainsi que les objets qu'elle utilise. Les mécanismes, fournis par une nouvelle API, permettent de capturer l'état d'une tâche sous la forme d'un objet Java qui peut alors être sérialisé puis migré vers la cible où il est désérialisé et enfin restauré.

Enfin, la machine virtuelle Aroma [Suri 01] est une implémentation de la machine virtuelle Java écrite spécialement pour gérer la migration d'agents. Elle offre une grande flexibilité dans la granularité possible des captures : elle permet ainsi de capturer et de transférer l'état de tâches individuelles (comprenant la pile d'exécution ainsi que tous les objets atteignables par la tâche) ou de toutes les tâches en cours d'exécution, auquel cas les classes chargées dans le système sont adjointes à la capture.

Les travaux cités dans cette section montrent qu'il est possible de migrer l'état complet d'une tâche Java pour peu qu'un support particulier soit intégré à la machine virtuelle qui l'exécute. Il est ainsi possible de faire de la migration forte d'applications Java. On remarquera deux caractéristiques de ces travaux : d'une part, les mécanismes de capture de l'état d'une tâche sont implémentés au niveau de la machine virtuelle à la manière du mécanisme de sérialisation d'objets (figure 3.7). C'est à l'initiative d'une application, et au travers de l'invocation de nouvelles méthodes de l'API, qu'une tâche est capturée et migrée. D'autre part et par conséquent, ils ne concernent que la capture de l'état d'une tâche ou d'un ensemble de tâches, et ne prennent pas en compte la migration totale du système. De plus, le mécanisme de migration n'est pas transparent du point du vue du système, puisque c'est lui effectue l'opération de migration.

Cette forme de migration, intervenant au niveau du système, est-elle adaptée au transfert que nous voulons effectuer dans le cadre de la romization ?

3.2.2.5 Mobilité et romization

Nous avons défini la romization comme une forme de capture et de migration d'un système en cours d'exécution. Voyons comment les différentes formes de romization s'insèrent dans les schémas de migration que nous avons étudiés.

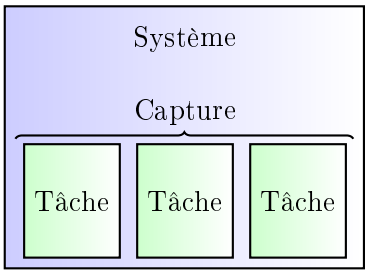


FIG. 3.7 – Les mécanismes de migration forte sont implémentés au niveau du système.

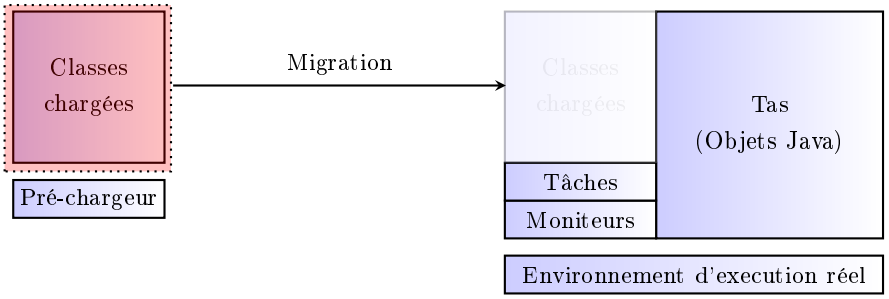


FIG. 3.8 – Utilisation d'un pré-chargeur de classes pour migrer des classes chargées.

Le pré-chargement de classes : une migration faible Comme nous l'avons vu dans la section 3.2.2.3, le pré-chargement de classes, en ne migrant que du code et des données d'initialisation, peut être assimilé à une migration faible. La proportion du système qui est migrée reste par conséquent assez limitée : ni le tas, ni les mécanismes internes du système destination ne sont affectés (figure 3.8).

Le pré-chargement de classes, en n'autorisant pas l'exécution du système, interdit par là toute forme de migration forte entre les environnements d'exécution virtuel et réel puisque la migration forte se définit comme le transfert d'une tâche en cours d'exécution. Le schéma de romization que nous voulons proposer doit permettre de démarrer des tâches hors-ligne et donc d'effectuer de la migration forte. Il doit en réalité aller plus loin que cette migration, puisqu'il nécessite un transfert non pas de l'état des tâches seules, mais de l'état complet du système (moniteurs, état mémoire, etc.).

La romization d'un système pré-déployé : une migration « encore plus forte » ? Migrer un système en cours d'exécution, de telle sorte que la migration soit totalement transparente pour celui-ci, implique de capturer son état complet (figure 3.9).

La migration forte se définit comme le fait de déplacer une tâche et son état d'exécution d'un environnement à un autre. Bien que le système comprenne en effet un ensemble de tâches qui peuvent être migrées de cette manière, il dispose également d'un état interne en dehors d'elles. Cet état comprend notamment :

- L'état de l'ordonnanceur de tâches ;
- L'état du gestionnaire de mémoire ;
- L'état des entrées/sorties critiques, comme le périphérique d'affichage ou les ports de

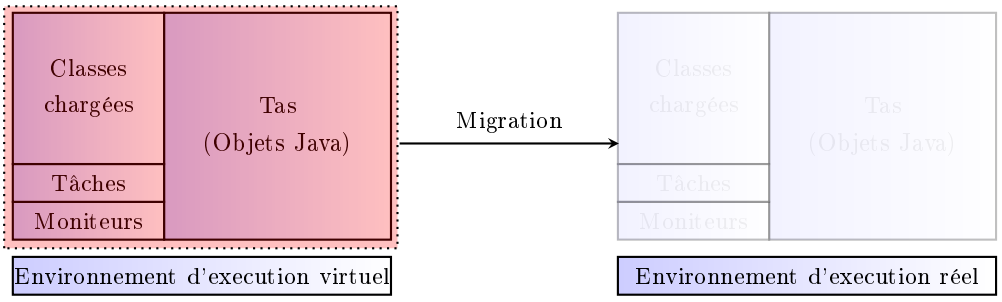


FIG. 3.9 – L'utilisation d'un environnement d'exécution virtuel pour préparer le système requiert la migration de toutes ses composantes.

communication.

Les mécanismes de migration forte à eux seuls ne sont donc pas suffisants pour permettre la migration de l'état complet du système. De plus, étant implémentés au niveau du système, ils requièrent une action de sa part pour effectuer la capture et leur opération n'est par conséquent pas neutre pour celui-ci. Ce dernier point va à l'encontre du principe de romization qui implique que le système ne soit pas conscient de sa capture et son transfert. Il est donc nécessaire de définir un nouveau niveau de migration, plus élevé que celui de la migration forte, pour prendre en compte la capture transparente de l'état du système.

3.2.2.6 La migration système

La migration forte ne satisfait pas aux besoins de notre solution de romization pour deux raisons :

1. Elle ne concerne que la capture de l'état de tâches en cours d'exécution, et non pas de l'état complet du système ;
2. Se situant au niveau du système, elle implique une gestion de sa part pour effectuer la capture, la migration et la restauration de l'état des tâches. Or, notre schéma nécessite que la migration soit transparente *y compris pour le système*.

Afin de pouvoir capturer l'état entier du système d'une façon totalement transparente pour ce dernier, le processus de migration doit se situer non pas au niveau du système (comme c'est le cas pour la migration forte) mais *au-dessus* de celui-ci.

> **Définition : La migration système** consiste à capturer et à migrer l'état complet d'un système de manière transparente pour ce dernier.

L'environnement d'exécution virtuel faisant partie du romizer et dans lequel s'exécute le système constitue le niveau idéal pour effectuer la migration système (figure 3.10). De cette manière, le système entier peut être capturé sans que cette capture n'ait d'effet de bord sur son comportement.

Ce très haut degré de migration imposé par notre solution, sur un système déjà en cours d'exécution, a un impact important sur la conception de l'environnement d'exécution embarqué qui reprendra la suite de son exécution.

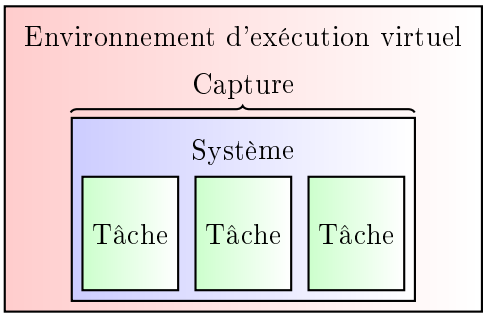


FIG. 3.10 – La migration système se situe au-dessus du système à capturer.

3.2.3 Impact sur l'environnement d'exécution embarqué

Notre architecture de romization change complètement la manière de capturer le système, et la profondeur de cette capture. L'impact de ce changement sur l'environnement d'exécution réel n'est pas nul.

Reprenant l'exécution d'un système fourni par l'environnement d'exécution virtuel, la machine virtuelle embarquée doit être une parfaite réplique d'un point de vue fonctionnel de ce dernier. Les deux environnements doivent notamment utiliser les mêmes formats et les mêmes données internes. La sémantique des bytecodes qu'ils interprètent doit aussi être strictement identique.

De plus, la machine virtuelle embarquée ne doit pas être perçue comme un système chargé de couvrir le cycle d'exécution du système dans son intégralité, mais plutôt comme un environnement permettant de reprendre l'exécution d'un système créé par le romizer. Pour pouvoir opérer, elle doit en effet être liée statiquement avec un système produit par ce dernier, qu'il soit dans un état de déploiement avancé ou au contraire à peine initialisé. Par ailleurs, Les données récupérées par l'environnement réel ne se limitent plus aux classes ayant été pré-chargées : la migration du système concernant le système dans son ensemble, la machine virtuelle embarquée doit se lier à toutes ses données, qu'elles soient objets Java faisant partie du tas ou données internes comme l'état de l'ordonnanceur de tâches. L'environnement d'exécution réel n'est donc plus qu'un support chargé de faire évoluer l'état d'un système préexistant.

L'édition de liens entre l'environnement d'exécution embarqué et le système romizé comprend également, comme nous l'avons vu au début de cette section, le placement des objets de ce dernier dans les différentes mémoires physiques de l'équipement cible. Là encore, les modalités changent par rapport au pré-chargement de classes.

3.2.4 Placement des données en mémoire

Cette tâche subit également des bouleversements profonds. Les données fournies par un pré-chargeur de classes sont affectables en mémoire selon un schéma clair : les méta-données de classes ont atteint leur état final et peuvent être placées en ROM, tandis que les champs statiques doivent se trouver en mémoire réinscriptible, car modifiables pendant l'exécution du système.

Mais dans notre schéma de romization, toutes les classes n'ont pas obligatoirement le même état de déploiement. Certaines peuvent déjà avoir été activées, d'autres se trouver dans l'état `LINKED`[6], d'autres dans l'état `LOADED`, d'autres enfin avoir à peine été créées. La staticité des méta-données n'est donc pas acquise, mais doit être considérée au cas par cas pour chaque classe existant dans le système.

Par ailleurs, le placement en mémoire concerne maintenant d'autres données que les classes. Les tâches, moniteurs et autres structures internes manipulées par la machine virtuelle sont logiquement placées en mémoire réinscriptible. Concernant le tas d'objets Java, certains sont absolument constants en fonction de leur classe (par exemple, les chaînes de caractères sont toujours constantes en Java) et peuvent inconditionnellement être placés en ROM. D'autres objets en revanche peuvent potentiellement subir des opérations d'écriture, mais cette condition dépend du comportement futur du système. Par exemple, un objet pourra ne jamais être modifié par l'exécution future du programme. Si tel est le cas, il peut être placé en ROM en toute sécurité — encore faut-il le savoir. Certaines des analyses mentionnées en 2.1.3.3 peuvent être utilisées pour repérer ces objets. Elles peuvent aussi être mises à profit afin de repérer les objets qui, bien que référencés, ne sont jamais atteints par le système dans le futur. Il est alors possible de les supprimer.

3.2.5 Spécialisation du système

Le passage d'un pré-chargeur vers un environnement d'exécution virtuel change là encore la manière dont cette préoccupation est adressée. Dans les schémas de romization classiques, la spécialisation se fait en supprimant certains champs et méthodes qui ont été détectés comme inutilisés soit par un outil d'analyse, soit par le programmeur[7]. Par ailleurs, d'autres spécialisations peuvent être effectuées au niveau du code source ou des classes par des outils tels que ceux présentés en 2.1.3.3.

Dans notre schéma de romization, la spécialisation ne s'effectue plus sur un ensemble de classes non-chargées ou pré-chargées, mais sur un système entier et déployé. Les analyses habituellement effectuées sur la forme non-déployée des programmes peuvent maintenant s'effectuer sur le système dans son ensemble, à un moment précis de son exécution. Ce point est particulièrement prometteur au regard de la qualité des spécialisations que nous pourrions obtenir. Nous y reviendrons en détail dans le chapitre 4, après avoir détaillé notre implémentation de cette nouvelle architecture de romization et justifié nos choix de conceptions.

3.3 Mise en œuvre

Cette section présente dans un premier temps l'environnement JITS, dans lequel notre architecture a été implémentée. Nous discutons ensuite de la manière dont nous représentons l'état du système dans l'environnement d'exécution virtuel, ainsi que des détails de sa capture et de sa restauration. Enfin, nous détaillons la séparation des différentes tâches du romizer en outils indépendants, intervenant chacun sur cet état du système.

[6] Pour les définitions des états des classes, voir la section 2.3.2

[7] Ainsi que le permet J2ME, voir 2.3.4.

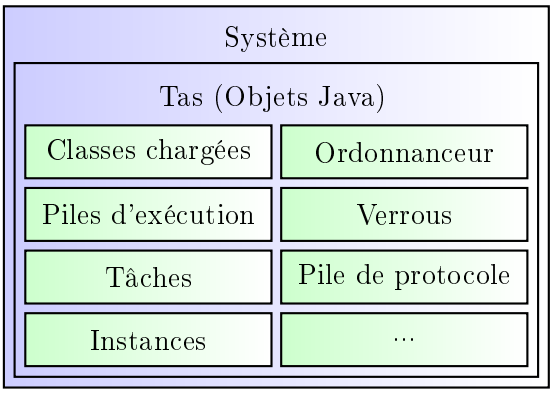


FIG. 3.11 – JITS modélise l'état complet du système sous la forme d'objets Java.

3.3.1 L'environnement JITS

L'architecture de romization que nous présentons a été implémentée pour l'environnement *JITS* (*Java In The Small* [JITS]). JITS a pour ambition de proposer un environnement Java pour petits systèmes compatible avec l'édition standard de Java [Bizz 02] et hautement personnalisable. Cette approche permet de préserver la compatibilité Java pour les applications, et de spécialiser le système non pas en fonction de ses applications supposées, comme le font les variantes embarquables de Java, mais en fonction de ses applications réelles.

La romization se situe au centre des préoccupations de JITS. En effet, la machine virtuelle embarquée est incapable de s'amorcer elle-même et nécessite d'être liée avec un système romizé. Cette étape est nécessaire en raison d'un choix de conception particulier : toutes les structures internes du système faisant partie de son état (y compris les classes, tâches, et piles d'exécution) sont représentées sous forme d'objets Java. Ce choix permet de s'affranchir totalement du problème de la capture et de la conversion des structures natives du système mentionné en 3.2.2, puisque la capture d'objets Java est l'opération de migration la plus triviale. En revanche, il implique que la machine virtuelle soit totalement dépourvue d'état propre. Ce point rend l'amorçage du système équivalent à la romization d'un système basique. La machine virtuelle de JITS n'est donc capable que de reprendre l'exécution d'un système Java romizé : celui-ci peut ne pas encore avoir été démarré, auquel cas son comportement est identique à celui de toute autre machine virtuelle, à l'exception que toutes les opérations d'amorçage ont déjà été effectuées.

Un autre choix de conception de JITS, qui est la conséquence logique du premier, est d'implémenter le plus de fonctions possibles du système en Java. Ainsi, des éléments comme le chargeur de classes, les politiques d'ordonnancement des tâches ou la pile TCP/IP sont-ils écrits en Java. Seules les fonctions nécessitant réellement une intervention directe du matériel (lecture d'un octet sur une ligne série, affichage à l'écran, etc.) ont une implémentation différente sur chaque cible. Cette décision s'accorde avec le but de JITS qui est d'obtenir des systèmes à empreinte mémoire minimale, le bytecode Java étant plus compact que le code natif, bien que moins performant. Il reste toutefois possible, lors de la romization du système, de décider d'en compiler certaines parties nativement en utilisant un compilateur *ahead-of-time* [Proe 97, Mull 97].

Ces deux points rendent JITS particulièrement favorable à l'utilisation d'un environne-

ment d'exécution virtuel dans son processus de romization, et à un découpage des préoccupations de la romization en différents outils. En effet, tout l'état du système étant contenu dans le tas d'objets Java, il est aisé de le capturer et de le transformer.

La suite de cette section décrit l'implémentation de notre environnement de romization au sein de JITS. Ce dernier étant écrit dans le langage Java, toutes les implémentations décrites ci-après ont été effectuées dans ce langage.

3.3.2 Représentation de l'état du système Java

Dans notre implémentation, un état du système consiste exclusivement en l'ensemble des objets Java qui composent son tas.

> **Définition :** L'état complet d'un système JITS est l'ensemble des objets Java de son tas. Cet ensemble comprend non seulement les instances classiques, mais également les classes, les méthodes et leur bytecode, ainsi que toutes les méta-données utilisées par la machine virtuelle qui sont eux-mêmes des objets Java classiques.

La modélisation du tas d'objets Java présents dans le système est appelée un « `ObjectsPool` ». Cet `ObjectsPool` contient un certain nombre d'instances de la classe `VMObject`, qui représente un objet Java quelconque. En outre, il comprend également une copie des méta-données des classes, méthodes et champs nécessaires au bon fonctionnement de la machine virtuelle (sous la forme d'instances des classes `VMStructClass`, `VMStructMethod` et `VMStructField`). Ces mêmes méta-données sont également présentes sous la forme d'objets Java dans le tas — toutefois, il est nécessaire de les conserver également à l'extérieur du tas, de telle sorte qu'elles restent disponibles même après une éventuelle suppression des objets correspondants[8]. Les différents outils de manipulation du système reposent en effet sur leur présence pour pouvoir fonctionner.

3.3.2.1 Représentation des objets Java

La classe `VMObject` est utilisée pour représenter un objet Java, quelle que soit sa classe. Elle comprend les champs suivants :

- Une référence vers la `VMStructClass` représentant la classe de l'objet ;
- Un tableau d'entiers, utilisé pour stocker les valeurs des champs de type primitif ;
- Un tableau de références vers des `VMObject`, utilisé pour stocker les valeurs des champs de type référence ;
- Le nom de l'objet. Ce dernier est une chaîne de caractère générée de manière unique lors de la création de l'objet, et pouvant être utilisée pour l'identifier ;
- Une table de hachage des attributs de l'objet. Les attributs sont des valeurs associées à un objet par un outil dans le but de l'annoter, et ne font pas partie du système final. Ils sont identifiés par une chaîne de caractère unique. Par exemple, l'attribut « `Partition` » indique dans quelle partition mémoire physique un objet doit être placé.

Chaque objet du système peut être représenté par cette classe. Cependant, deux sous-classes étendent la classe `VMObject`.

[8] Ceci se produit en particulier pendant la spécialisation du système, comme nous le verrons chapitre 4.

La classe `VMArray` est utilisée pour représenter tout objet de type tableau. Elle comprend en plus un membre indiquant la taille du tableau, et fournit quelques méthodes d'accès à ses membres.

La classe `VMStack` représente quant à elle un cas particulier d'objet Java : la pile d'exécution. Celle-ci est en effet un tableau pouvant contenir à la fois des valeurs et des références, et comprend également quelques membres tels que l'index du sommet de pile ou de la fenêtre courante. Sa représentation nécessite donc une classe particulière.

3.3.2.2 Représentation des méta-données des classes

Les méta-données des classes, présentes dans les instances des classes `Class`, `Method` et `Field` chargées dans le système, sont dupliquées par les classes `VMStructClass`, `VMStructMethod` et `VMStructField`. Celles-ci ne font que répliquer les méta-données constantes des classes chargées et ne font absolument pas partie du système final. Elles permettent en revanche d'assurer la disponibilité de ces données essentielles pour travailler sur le système, même si les méta-données originales de ce dernier sont supprimées.

VMStructClass Cette classe contient toutes les méta-données relatives à une classe, à savoir :

- Son nom ;
- Ses propriétés de genre et d'accès (publique, privée, interface, etc.) ;
- Son état (`LOADED`, `LINKED`, `READY`) ;
- Une référence vers la `VMStructClass` de sa superclasse ;
- La liste des `VMStructMethod` correspondant à ses méthodes et interfaces ;
- La liste des `VMStructField` correspondant à ses champs ;
- Une référence vers le `VMObject` représentant la classe.

Toute classe présente dans un des chargeurs de classes du système a une `VMStructClass` associée.

VMStructMethod La classe `VMStructMethod` remplit le même rôle que `VMStructClass` au niveau des méthodes présentes dans le système. Elle fait office de cache pour les informations suivantes concernant une méthode chargée :

- Son nom et son descripteur ;
- Ses propriétés de genre et d'accès (publique, privée, native, etc.) ;
- Pour les méthodes virtuelles, son index dans la table des méthodes virtuelles de sa classe, ou pour les méthodes d'interface sa clé d'interface ;
- Une référence vers sa classe associée ;
- Une référence vers le `VMObject` représentant la méthode au sein du pool.

VMStructField Comme pour `VMStructClass` et `VMStructMethod`, cette classe permet de garder les informations essentielles concernant un champ. Elle contient notamment :

- Le nom du champ ;
- Ses droits d'accès et son état de chargement (public, privé, static, final, ...) ;
- Une référence vers la `VMStructClass` de son type ;
- Le décalage qui permet d'accéder au champ dans un objet Java ;

– Une référence vers la classe associée au champ ;
– Une référence vers le `VMObject` représentant le champ.

3.3.2.3 Représentation du tas

La classe `ObjectsPool`, qui sert à représenter le tas d'objet et donc l'état du système, est principalement un ensemble de `VMObject`. Elle comprend une table des objets qui sont rendus accessibles par leur nom, ainsi qu'une table de toutes les `VMStructClass` représentant chacune des classes présentes dans le système. Cette dernière permet aux outils d'accéder facilement et rapidement aux méta-données du système. De plus, elle peut également contenir des attributs sur le même modèle que les attributs d'objets.

La classe `ObjectsPool` comprend également une méthode permettant d'effectuer une opération de ramasse-miettes : son effet est du retirer du pool tous les objets du système qui ne sont pas directement ou indirectement atteignables à partir d'un ensemble d'objets racines.

Cette représentation du système Java est entièrement sérialisable : la classe `ObjectsPool` comprend des méthodes permettant de sérialiser et de désérialiser l'ensemble de la représentation du système, incluant les objets, attributs, et méta-données préservées vers un flux XML. Cette fonction assure la migration du système entre deux outils, et permet de séparer les différentes préoccupations du romizer en outils réellement indépendants qui peuvent être contrôlés par différents acteurs.

3.3.3 Les outils de manipulation du système

Nous avions effectué en section 3.2 un recensement des tâches remplies par le romizer. Afin de permettre une meilleure flexibilité, nous avons implémenté chacune de ces préoccupations dans un outil indépendant qui permet de modifier l'état d'un système Java sérialisé. Celui-ci est passé d'un outil vers un autre jusqu'à ce que l'état désiré pour le système soit obtenu et que ce dernier soit transféré vers l'équipement cible.

La section 3.2 nous a donné une liste de préoccupations à implémenter sous forme d'outil :
– Une machine virtuelle Java qui fait évoluer le système en l'exécutant hors-ligne ;
– Un migreur qui crée une image mémoire du système utilisable par l'équipement cible ;
– Un placeur d'objets en mémoire ;
– Un spécialiseur de système.

À celles-ci, il faut ajouter un outil d'amorçage qui crée une image mémoire minimale, suffisante pour que la machine virtuelle s'exécute et soit en mesure de charger elle-même des classes applicatives et de les exécuter. Cet outil fait office de source : à partir d'un ensemble de classes non-chargées, il produit un `ObjectsPool` minimal qui est ensuite traité par les autres outils. Le migreur remplit le rôle opposé, celui de puits : le système qu'il prend en entrée est converti vers une forme native et ne peut plus être modifié.

La figure 3.12 montre comment les différents outils s'architecturent autour de la représentation du système que nous venons de décrire.

On remarquera qu'en lieu et place d'un schéma séquentiel dans lequel les tâches liées à la romization sont accomplies les unes après les autres dans un ordre défini, notre implémentation se base sur une schéma plus libre dans lequel les différentes préoccupations de la romization peuvent intervenir librement, le point de transition entre deux tâches étant

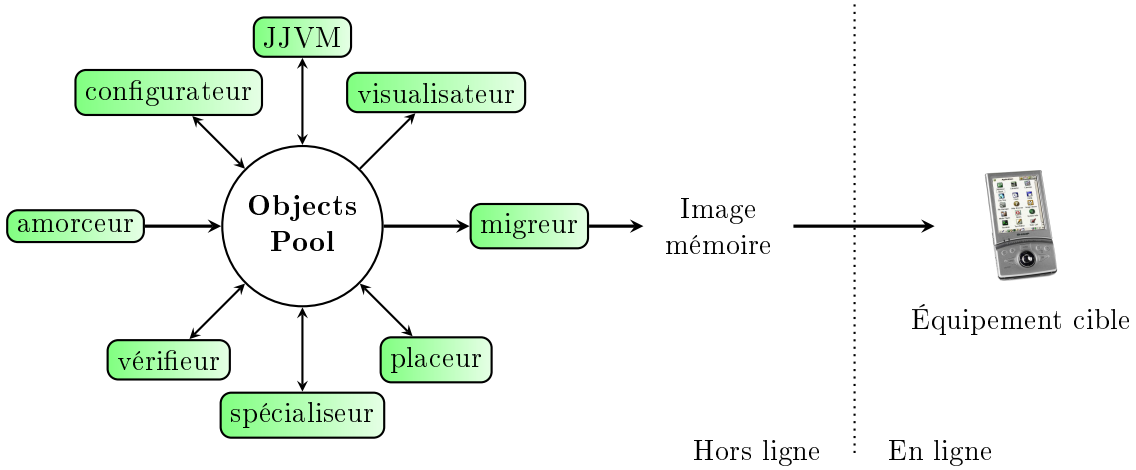


FIG. 3.12 – Organisation des différents composants du romizer de JITS

l'`ObjectsPool` sérialisé. On remarquera aussi sur la figure qu'en plus des outils correspondant aux préoccupations du romizer précédemment identifiées, nous proposons d'autres outils agissant sur un `ObjectsPool` :

- Le *configurateur* permet de modifier le système sans avoir à faire appel à la machine virtuelle, par accès direct à ses objets. Il peut notamment être utilisé pour remplir des opérations comme affecter une adresse IP et une table de routage à la pile TCP/IP du système ;
- Le *vérifieur* s'assure de la consistance du système. En particulier, il vérifie qu'aucun flux d'entrée/sortie n'est ouvert au moment de la migration ;
- Le *visualisateur* fournit une vue graphique du système, en montrant notamment les références entre objets et en donnant certaines informations globales comme le nombre d'instances d'une certaine classe. Il permet ainsi de visualiser les particularités et éventuellement les problèmes d'un système.

Dans la suite de cette section, nous allons détailler les outils prenant part au processus de romization, établir clairement leur rôle et expliquer leur fonctionnement.

3.3.3.1 L'amorceur du système

Afin de pouvoir fonctionner, tout système doit lors de son démarrage être capable de créer lui-même les structures de données nécessaires à son propre fonctionnement. Cette opération est appelée *amorçage* ou *bootstrapping*.

Dans notre schéma de romization, où le système transite d'outil en outil sous une forme sérialisée, l'amorçage se traduit par la création des objets Java du système qui sont nécessaires au bon fonctionnement de la machine virtuelle Java. En particulier, les classes essentielles contenant les structures de données de la machine virtuelle doivent être chargées et initialisées.

Cette opération est effectuée en tirant partie de la machine virtuelle sur laquelle l'outil d'amorçage s'exécute. Les classes principales du système JITS sont chargées en utilisant le

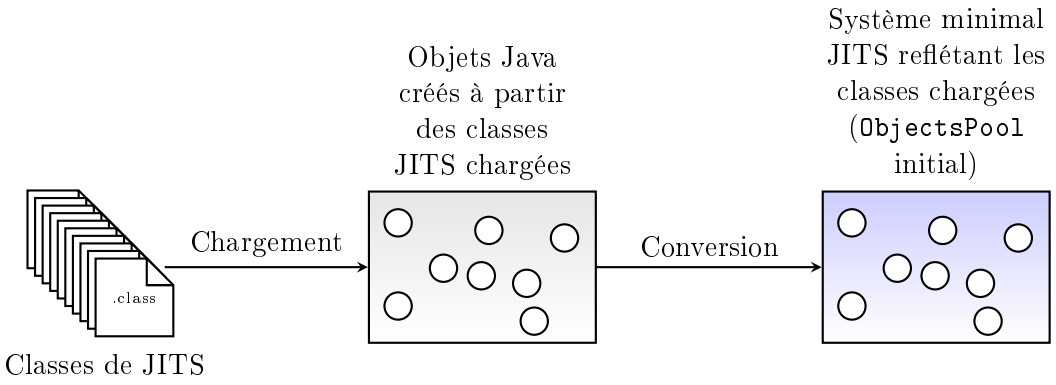


FIG. 3.13 – Le processus d'amorçage de JITS.

chargeur de classes de JITS (écrit en Java) avec la machine virtuelle hôte ; puis, les classes sont converties sous forme d'instances de la classe `VMObject` en utilisant les capacités de réflexion de Java, et sont finalement associées au chargeur de classes de JITS qui a lui-même été converti par ce processus. Les méta-données des classes chargées sont ensuite reflétées dans un ensemble de `VMStructClass`. Le système ainsi obtenu est ensuite sérialisé et constitue la base sur laquelle la machine virtuelle peut s'exécuter (figure 3.13).

3.3.3.2 La machine virtuelle Java-Java (*JJVM*)

La *machine virtuelle Java-Java*[9] (*Java-Java Virtual Machine*, ou *JJVM*) prend en entrée un `ObjectsPool`, et le fait évoluer selon la spécification de la machine virtuelle Java [Lind 99]. Il s'agit donc d'une machine virtuelle au sens premier du terme. N'ayant pas d'état interne contenu ailleurs que dans l'`ObjectsPool`, l'état du système peut être capturé à n'importe quel moment de son exécution par simple sérialisation. Pour faciliter la capture de l'état du système au moment désiré, il est possible de poser des points d'arrêt sur le code. Lorsqu'un point d'arrêt est rencontré par l'interpréteur de bytecode, l'exécution s'arrête et l'`ObjectsPool` est sérialisé.

Étant une machine virtuelle classique, la JJVM est capable de charger les classes applicatives et de créer des tâches. Elle remplace donc les pré-chargeurs de classes dans les schémas de romization que nous avons étudiés en 2.3.4.

Le modèle mémoire fourni par la JJVM est celui d'un environnement Java classique : une seule mémoire, accessible en lecture/écriture, qui en pratique est associée au tas de l'environnement Java sous-jacent. Les problématiques liées au placement des objets dans les différentes mémoires physiques de l'équipement cible ne sont pas adressés au niveau de la JJVM, mais sont exclusivement à la charge du placeur d'objets.

[9] La machine virtuelle Java-Java porte ce nom car il s'agit d'une machine virtuelle Java écrite elle-même en Java.

3.3.3.3 Le placeur d'objets en mémoire

Cet outil est chargé d'assigner une mémoire de destination physique à chaque objet Java présent dans un `ObjectsPool`, en lui ajoutant un attribut particulier selon le modèle décrit dans la section 3.2.4. Il a donc besoin d'une description des différentes mémoires présentes sur la cible et de leurs caractéristiques (taille, accès en écriture possible, performances, ...) et d'une politique de placement des objets [Marq 05].

La problématique du placement optimal des objets en mémoire fait actuellement l'objet d'une thèse au LIFL menée par Kévin Marquet [Marq 04, Grim 05].

3.3.3.4 Le spécialiseur de système

Le spécialiseur de système prend en entrée un `ObjectsPool` et en fournit une version spécialisée en sortie, fonctionnellement équivalente mais purgée de tous les objets détectés comme inutilisés dans le flot d'exécution futur du système, et dont les éléments restants sont spécialisés pour son exécution future (méthodes, structures de données, etc.).

La spécialisation du système en cours de déploiement étant la contribution principale de cette recherche, nous reviendrons plus longuement sur cet aspect dans le chapitre 4.

3.3.3.5 Le constructeur d'image mémoire (migreur)

Cet outil se situe en bout de chaîne et ne produit pas de sortie exploitable par un autre outil. Il prend un `ObjectsPool` en entrée et en produit une image mémoire sous forme de déclarations de structures C, utilisable par la machine virtuelle embarquée de JITS.

La figure 3.14 montre un exemple de sortie produite par le migreur, ici classe Java (implémentant une liste chaînée d'entiers) dont une instance est créée avant sérialisation du système, et le code C généré par ce dernier. La définition de la classe Java se traduit par la déclaration d'une structure reprenant tous ses membres ainsi que ceux de ses superclasses, ordonnancés de telle sorte que les indexes des membres de classes surchargées restent les mêmes dans les sous-classes. On voit ainsi que le membre `relatedClass`, qui figure dans la classe `java.lang.Object` et qui pointe vers la classe de l'objet, se retrouve dans la classe `IntList` en première position.

La seconde déclaration du listing C définit les méta-données de la classe `IntList`, c'est à dire l'instance de la classe `java.lang.Class` la représentant.

Enfin, la dernière déclaration définit l'instance de `IntList` allouée à la fin du listing Java. Dans le code C, un objet est une variable du type de la structure représentant sa classe, initialisé avec les valeurs qu'il possédait dans le système au moment de sa capture.

Le migreur effectue donc une compilation des objets présents dans le système romizé vers une représentation C équivalente. Il se sert des attributs de système et d'objets pour gérer les spécificités de l'architecture cible : ainsi, l'attribut de mémoire de destination ajouté par le placeur d'objets est exploité pour ordonnancer les objets en fonction de leur mémoire de destination et définir les différents secteurs mémoire.

Une fois que l'image mémoire C du système est générée, elle doit être compilée et liée avec l'environnement Java embarqué (figure 3.15). Cette tâche est accomplie par le même compilateur que celui utilisé pour compiler l'environnement Java embarqué. Ainsi, la cohérence des structures de données est assurée lors de l'édition des liens. Cette opération équivaut

Code Java romizé

```
public class IntList {
  private int value;
  private IntList next;

  public IntList(int value,
    IntList next) {
    this.value=value;
    this.next=next;
  }

  ...

}

...

  IntList list;
  list=new IntList(42,null);

...
```

Code C généré

```
typedef struct IntList {
  struct java_lang_Class *
    relatedClass;
  int value;
  IntList * next;
}

struct java_lang_Class
    class_IntList={
  &class_java_lang_Class, /*classe*/
  "IntList" /*nom*/,
  0x00ff00ff /*attributs*/,
  ....
}

struct IntList object_99={
  &class_IntList /*classe*/,
  42 /*value*/,
  NULL /*next*/
}
```

FIG. 3.14 – Exemple (simplifié) de code C généré à partir d'une classe Java, comportant une définition de classe ainsi qu'une instance de celle-ci.

à « charger » le système Java dans son environnement d'exécution embarqué, de la même façon que la JJVM chargeait sa représentation sérialisée pour pouvoir l'exécuter hors-ligne. L'image mémoire binaire ainsi obtenue peut être écrite dans les mémoires de l'équipement cible. Une fois mis sous tension, celui-ci reprend immédiatement l'exécution du système à partir de l'état dans lequel il était lorsqu'il a été déchargé.

3.4 Aspects novateurs de cette architecture

Dans ce chapitre, nous avons présenté une nouvelle architecture de romization, née de la nécessité de permettre le déploiement complet hors-ligne des applications sur un système

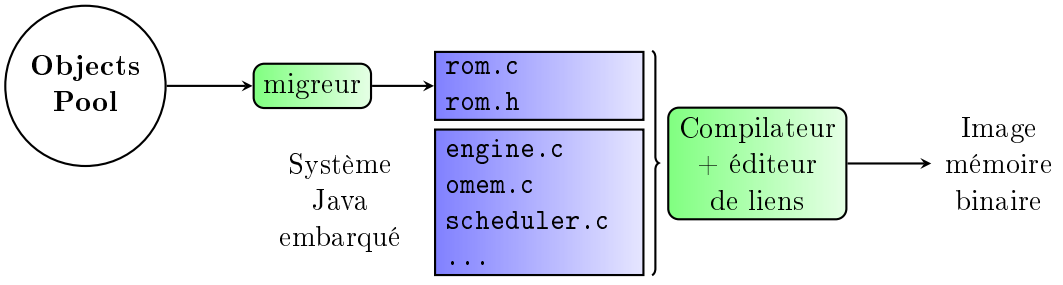


FIG. 3.15 – Obtention d'une image mémoire binaire à partir d'un système Java sérialisé.

embarqué.

Afin de clarifier le concept de romization, utilisé de manière empirique dans l'industrie, nous avons tout d'abord établi un état de l'art des solutions existantes afin d'en tirer les caractéristiques et fonctionalités principales. Sur cette base, nous avons défini la romization comme étant une opération de pré-déploiement « in-vitro » de logiciels sur un système, ce dernier migrant de manière transparente d'un environnement d'exécution virtuel à son environnement d'exécution réel une fois qu'il a atteint un état jugé satisfaisant.

Cette définition nous a amené à considérer toutes les activités du déploiement logiciel au sein du romizer, et à proposer une architecture de romization permettant d'effectuer le déploiement complet du système hors-ligne. Pour ce faire, il a été nécessaire de considérer la manière de migrer le système de façon transparente pour celui-ci. C'est pourquoi, au dessus des concepts de migration *faible* et de migration *forte*, nous avons défini la notion de migration *système*, c'est-à-dire la migration transparente d'un système complet en cours d'exécution, effectuée par l'environnement dans lequel le système s'exécute.

Nous avons ensuite implémenté cette migration dans le processus de romization de JITS, environnement Java pour lequel l'état complet du système est contenu dans le tas d'objets. Cet état complet et migrable peut transiter entre environnement d'exécution virtuel et réel, mais également entre un ensemble d'outils chargés de produire la meilleure image possible mémoire du système. En identifiant les différentes préoccupations du romizer, nous avons été capable de les séparer et d'implémenter chacune d'entre elles sous la forme d'un outil faisant évoluer l'état du système jusqu'à l'état désiré pour sa migration.

Nous pensons que la romization renferme un grand potentiel pour l'informatique embarquée — un potentiel qui a jusque-là été négligé ou sous-estimé. En effet, une fois clarifiée, la notion de romization a des implications importantes sur le déploiement du système : le romizer dispose ainsi d'une vision complète du système déployé, dans la forme qui sera la sienne dans son environnement d'exécution réel. Une telle forme correspond à un cas « idéal » pour analyser le système et inférer des informations sur ce dernier. Cette richesse d'information obtenue peut alors être utilisée sciemment afin de spécialiser le système, à condition de savoir quelles analyses effectuer, comment utiliser la forme déployée pour améliorer leur précision, et comment spécialiser le système à partir de leur résultat. Nous allons adresser ces questions dans le prochain chapitre.

Quatrième Chapitre

Spécialisation de systèmes Java pendant la romization

« S1ze d0eS M@TtEr ! »

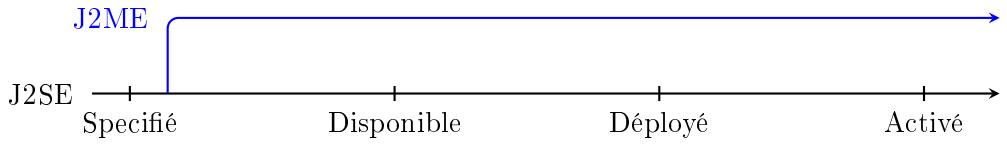
Sources variées,
Email reçu régulièrement durant ma thèse.

La taille finale d'un système embarqué a des répercussions importantes sur son processus d'industrialisation. Celles-ci peuvent se traduire par la possibilité ou pas de l'embarquer dans un équipement, ou par l'économie de centaines de milliers d'euros de silicium dans la production de millions d'unités. Le présent chapitre décrit les mécanismes permettant de tirer parti de l'architecture de romization présentée dans le chapitre précédent pour effectuer une spécialisation très forte d'un système Java générique et ainsi réduire son empreinte mémoire.

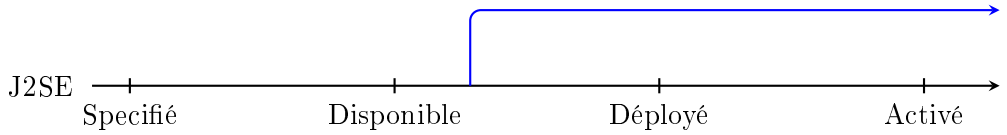
Notre approche consiste à déployer des applications sur un environnement Java standard, qui est ensuite spécialisé en fonction de celles-ci, de façon à produire un système fonctionnel minimal dédié à leur seule exécution. Cette approche est l'inverse de celle qui a donné naissance aux systèmes Java spécialisés tels que J2ME ou Java Card : ceux-ci requièrent des applications embarquées qu'elles soient écrites pour le système, car la spécialisation de ce dernier se produit *avant* leur déploiement. Notre approche, à l'inverse, conçoit un système compact adapté aux applications, spécialisé *après* le déploiement de ces dernières. Dans le premier cas, nous parlerons de **spécialisation en amont** ou **spécialisation précoce** et dans le second de **spécialisation en aval** ou **spécialisation tardive**.

Un des avantages de la spécialisation tardive est qu'elle n'enferme pas le programmeur dans un sous-ensemble de Java défini à l'avance — le système est taillé sur mesure en fonction des applications, et la taille du système spécialisé ne dépend ainsi que des fonctionnalités requises par les applications embarquées. Cette approche est déjà appliquée à la spécialisation de systèmes Java via l'extraction de sous-ensembles de classes d'applications et d'APIs [Rays 02, Tip 03], mais notre architecture apporte deux propriétés cruciales qui permettent d'appliquer les spécialisations de manière plus agressive et efficace :

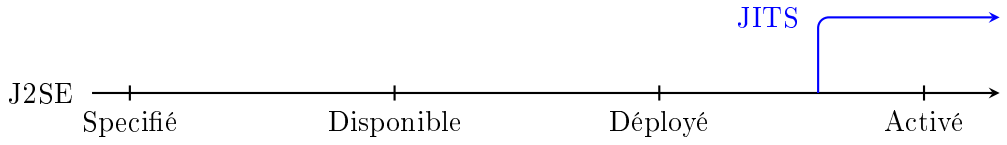
1. Le spécialiseur dispose d'une vision complète du système. Il travaille donc sur les applications à exécuter et les APIs Java, mais aussi sur les mécanismes internes du système comme ceux responsables de l'ordonnancement des tâches ou de la gestion du réseau. Le système peut donc être spécialisé en profondeur ;
2. Le spécialiseur travaille sur une image du système déjà déployé, voire en cours d'exécution. Ce dernier est donc plus proche de son état réel sur l'équipement embarqué que sa forme non-déployée : il est alors possible de prédire son comportement futur



(a) Spécialisation en amont ou précoce (J2ME).



(b) Spécialisation des APIs ([Rays 02, Tip 03]).



(c) Spécialisation en aval ou tardive (JITS).

FIG. 4.1 – Temps d'occurrence des différents types de spécialisation.

avec bien plus d'exactitude qu'en travaillant sur la forme non-chargée des classes, ce qui améliore d'autant la précision et la pertinence de la spécialisation. La figure 4.1 montre les temps d'occurrence des différents types de spécialisation sur la ligne de vie de l'édition standard de Java.

Nous décrivons, dans ce chapitre, des mécanismes de spécialisation pour les systèmes produits par notre architecture de romization capables de tirer parti de ces deux points. La forme du système sur laquelle nous travaillons est l'`ObjectsPool`, représentation chargeable d'un système Java en cours d'exécution décrite en 3.3. Notre outil de spécialisation s'intègre dans la chaîne de romization qui y est présentée : il charge cette représentation, l'analyse, la spécialise et produit en sortie une représentation d'un système sémantiquement équivalent au système d'entrée, mais spécialisé et allégé de ses éléments inutilisés pendant son exécution future.

La spécialisation opère en deux temps. Dans un premier temps, un analyseur examine le système figé et l'annote en fonction des informations qu'il infère sur son exécution future. Dans un second temps, le spécialiseur de système se sert de ces informations pour le réduire tout en assurant que sa sémantique future soit identique à celle du système non-spécialisé.

4.1 Analyse du système

Cette section décrit l'analyse que nous effectuons sur le système déployé. Elle rappelle les principes généraux inhérents au traitement de code, qu'il s'agisse de code source ou de code intermédiaire, puis précise les propriétés du système que nous souhaitons capturer ainsi que les moyens mis en œuvre pour les obtenir.

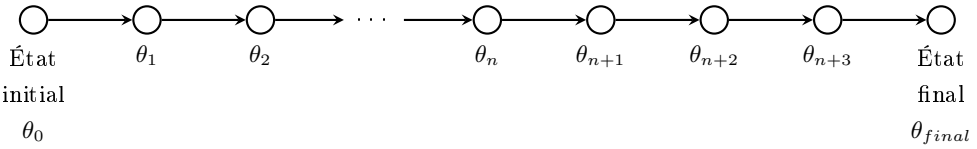


FIG. 4.2 – Un programme suit son chemin d'exécution depuis son état initial jusqu'à son état final, en passant par des états intermédiaires à l'exécution de chaque bytecode.

4.1.1 Principes généraux de l'analyse de programme

L'analyse d'un programme en vue de sa spécialisation consiste à déterminer lesquelles de ses données sont utiles à son exécution, ainsi que leurs conditions d'utilisation. Un analyseur de programme se voit fournir un **état** complet d'un programme, qui peut être un code source, un ensemble de classes non-chargées, ou dans notre cas l'image d'un système complet en cours d'exécution. Quelle que soit la forme, un état de programme est constitué d'un ensemble de **variables**, permettant de référencer au travers d'un identifiant (un nom ou une adresse) les **données** qu'elles contiennent. Une donnée est une entité de stockage numérique indivisible : en Java, elle peut être de type primitif (entier, flottant, ...) ou référence.

Un état de programme θ_n est constitué de toutes les variables du programme à un instant n de son exécution. Il peut évoluer vers l'état θ_{n+1} par avancement d'un pas de l'exécution du programme, c'est-à-dire par transformation de l'état θ_n selon les règles d'exécution du programme. Pas à pas, le programme finit ainsi par arriver dans un état final.

> **Définition :** L'état θ_n d'un programme à un instant n est l'ensemble de ses variables déclarées et présentes à l'instant n, lui permettant d'atteindre son état final par application de ses règles d'exécution.

Il est ainsi possible de représenter l'exécution du programme sous la forme d'un graphe dont les nœuds correspondent à un état et les transitions entre les nœuds à une étape d'exécution du programme. Ce graphe d'exécution a la propriété d'être un chemin[1], chaque transition faisant avancer l'état du programme d'une étape d'exécution, jusqu'à ce qu'il atteigne son état final (figure 4.2).

La spécialisation d'un programme vise à déterminer à partir d'un de ses états θ un sous-état réduit de ce dernier lui permettant de mener son exécution à terme, c'est-à-dire lui permettant de passer par tous les états intermédiaires jusqu'à son état final. Ce sous-état est composé des données de θ qui sont nécessaires pour que le programme atteigne son état final ; les autres données, inexploitées, peuvent alors être supprimées sans que l'exécution du programme n'en soit affectée. On dit alors que le programme est *spécialisé*.

> **Définition :** La spécialisation d'un programme dans un état θ consiste à retourner un sous-ensemble θ' de cet état tel que tous les états atteignables à partir de θ le soient également à partir de θ'. On dit alors que les états θ et θ' sont équivalents sémantiquement.

L'analyse idéale consisterait à faire progresser le programme d'état en état depuis l'état θ

[1] En théorie des graphes, un chemin est un graphe orienté connexe de degré entrant maximum 1, de degré sortant maximum 1 et de degré minimum 1.

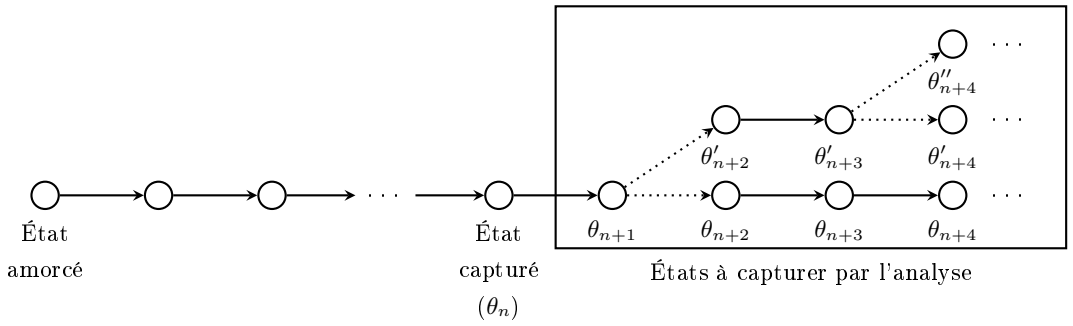


FIG. 4.3 – L'analyse d'un programme doit considérer tous les états atteignables par celui-ci à partir de son état courant.

jusqu'à son état final, et d'accumuler les états ainsi obtenus à chaque étape de l'exécution. De cette manière, l'analyse serait en mesure de fournir l'ensemble réel des états atteignables, et par conséquent des variables nécessaires à l'exécution du programme. Toutefois, cette analyse idéale est irréalisable en pratique pour différentes raisons :

- Certaines opérations ne sont pas déterministes à partir de l'état seul du programme, et peuvent conduire le programme vers des états différents entre différentes occurrences. Par exemple, la lecture de l'heure courante, ou la lecture d'un octet sur un flux ont un résultat totalement imprévisible à l'avance ;
- La taille des données à accumuler (une capture complète de l'état du programme à chaque étape) est trop imposante pour être réaliste, même sur une puissante machine de développement ;
- La durée de l'analyse serait au moins égale à la durée d'exécution du programme — et en vertu de la propriété d'indécidabilité de la terminaison des programmes, il n'est même pas garanti qu'elle se termine un jour.

Ainsi, contrairement à l'exécution réelle d'un programme qui n'emprunte qu'un seul chemin d'exécution, l'analyse de son flot d'exécution peut avoir à en évaluer plusieurs, introduits par le non-déterminisme de certaines opérations — très exactement, tous les chemins d'exécution potentiellement empruntables lors de l'exécution réelle du programme doivent être considérés par l'analyseur, comme le montre la figure 4.3.

De plus, il est nécessaire de factoriser les traitements induits par l'analyse, ainsi que les données retenues par celle-ci, pour respectivement réduire le temps d'analyse et la taille des données collectées à un niveau praticable.

Afin de réduire le volume d'information généré par l'analyse, nous proposons de garder non pas un état complet par étape de l'analyse, mais un seul état abstrait du système, utilisé à chaque étape, et capable d'exprimer tous les contextes atteignables. Cet état n'est pas associé à un moment particulier de l'exécution du système, mais représente des informations exactes à tout moment de son évolution. On dira qu'il est insensible au contexte, c'est-à-dire indépendant du moment considéré. Il est également pessimiste : on préférera représenter plus d'états du système qu'il n'y a d'états réellement atteignables plutôt que l'inverse, afin d'assurer que tous les états atteignables soient représentés dans l'état abstrait.

Cette représentation abstraite du système implique de pouvoir représenter les propriétés intéressantes des variables sous une forme unifiable : si une variable peut prendre deux valeurs

différentes pour deux états θ_n et θ_m de l'exécution du système, l'état abstrait de ce dernier doit représenter les deux valeurs possibles au travers d'une valeur abstraite exprimant l'union des propriétés de ces deux valeurs. Mais la valeur n'est pas la seule propriété utile d'une variable pour la spécialisation du système. D'autres propriétés sont également intéressantes dans ce but. Nous allons les identifier, et pour chacune d'entre elles définir sa représentation ainsi que sa règle d'unification.

4.1.2 Propriétés des variables à capturer

Au nombre des informations sur une variable que l'on peut compter capturer à l'aide d'une analyse de programme, nous discernons particulièrement :

- Le ou les types des données contenues dans la variable ;
- La ou les provenance des données stockées dans la variable (alias) ;
- La ou les valeurs de la variable ;
- Le type d'accès effectué sur la variable.

4.1.2.1 Type d'une variable

Pour toute variable u du système, le type (noté $T(u)$) indique la nature du contenu de cette variable pour l'ensemble des états considérés.

Représentation Le type d'une variable en Java est exprimé par sa classe. Le système de classes Java est une hiérarchie formant un treillis, pour lequel chaque couple d'éléments possède une borne inférieure. Par conséquent, tous les couples de classes possèdent un parent commun.

Un type abstrait peut être exact (la donnée stockée dans la variable est connue pour être exactement de ce type) ou non (la donnée de la variable est du type indiqué ou de l'un de ses sous-types). Cette notion d'exactitude du type est importante car elle influence l'évaluation des chemins d'exécution : ainsi, un appel de méthode virtuelle sur un objet dont le type est exact ne fournit qu'un seul chemin, tandis que le même appel sur une donnée dont le type n'est pas exact peut en générer plusieurs (un chemin de plus par sous-type surchargeant la méthode invoquée). Nous notons qu'un type est exact en préfixant son nom par le caractère « ! ». Ainsi, `Object` $\neq$ `!Object`, car une donnée du type `Throwable` est une sorte de `Object`, mais n'est pas une sorte de `!Object`.

Unification L'unification des types possibles d'une variable est l'union des ensembles des types de chaque variable :

$$T(u \cup v) = T(u) \cup T(v).$$

Il s'agit d'une union ensembliste. Le résultat de cette union peut être simplifié en supprimant les types pour lesquels un parent non-exact figure également dans l'ensemble (cas de redondance d'information).

Il est également possible de réduire la liste des types au prix d'une perte de précision. Pour cela, on factorise deux types présents dans l'ensemble en un seul, en les remplaçant par leur premier parent commun, non-exact. Cette opération peut également être utilisée pour

unifier deux types, en lieu et place du maintien d'un ensemble de types. Dans ce cas, le type du résultat n'est exact que si les deux type unifiés sont le même type exact.

Exemple Deux chemins d'exécution différents stockent une donnée dans la variable u, déclarée comme étant du type `Throwable`. Dans le premier chemin, cette donnée est du type `!NullPointerException` et dans le second elle est du type `!ArithmeticException`. L'unification du type abstrait de la variable u effectué lors de l'évaluation du second chemin est donc :

$$\begin{aligned} T(u) &= \{\texttt{!NullPointerException}\} \cup \{\texttt{!ArithmeticException}\} \\ &= \{\texttt{!NullPointerException}, \texttt{!ArithmeticException}\}. \end{aligned}$$

Si l'on préfère utiliser la factorisation vers le premier parent commun comme stratégie d'unification, le résultat sera :

$$T(u) = \{\texttt{Exception}\}.$$

Car `Exception` est le premier parent commun de `!NullPointerException` et `!ArithmeticException`.

4.1.2.2 Source d'une donnée de variable

Pour toute variable u du système, la source $S(u)$ indique les variables pouvant avoir fourni la donnée qu'elle contient, pour tous les états du système considérés.

Représentation Les provenances possibles d'une donnée sont regroupées dans un ensemble. Cet ensemble peut prendre la valeur spéciale $\{\Omega\}$ si la donnée peut provenir de n'importe quelle variable du système (autrement dit, que sa provenance n'est pas connue).

Unification L'unification des sources de deux variables est l'union de leurs ensembles :

$$S(u \cup v) = S(u) \cup S(v), \text{ et } S(u) \cup \{\Omega\} = \{\Omega\}.$$

Il est possible de simplifier l'ensemble des sources d'une variable en le remplaçant par l'ensemble Ω.

Exemple Un chemin d'exécution affecte à la variable u la somme des contenus des variables v et w. On a alors :

$$S(u) = \{v\} \cup \{w\} = \{v, w\}.$$

Considérons qu'un second chemin affecte dans u une donnée de provenance indéterminée. Le résultat est :

$$S(u) = \{v, w\} \cup \{\Omega\} = \{\Omega\}.$$

4.1.2.3 Valeur d'une variable

Pour toute variable u du système, la valeur $V(u)$ indique le contenu que peut avoir cette variable dans l'ensemble des états considérés.

Représentation La valeur abstraite d'une variable est un ensemble de constantes représentant les données pouvant être affectées à la variable.

On distinguera la représentation des données primitives de celle des références :

Les données primitives sont efficacement représentées par un ensemble d'intervalles numériques ;

Les références sont représentées sous la forme d'un ensemble contenant les valeurs possibles de la référence.

Pour les deux types de données, l'ensemble spécial $\{\Omega\}$ indique que la valeur de la variable n'est pas connue, c'est-à-dire que la variable peut prendre n'importe quelle valeur représentable par son type.

Unification L'unification des valeurs possibles d'une donnée est l'union des ensembles de valeurs de chaque donnée :

$$V(u \cup v) = V(u) \cup V(v), \text{ et } V(u) \cup \{\Omega\} = \{\Omega\}.$$

Cette unification est l'union ensembliste, où l'ensemble peut-être un ensemble de valeurs primitives ou d'objets :

- Pour les données primitives, il s'agit de l'union des ensembles d'intervalles représentés. Si le nombre d'intervalles dans l'ensemble devient trop important, il est possible de le réduire en factorisant les deux intervalles les plus proches. Dans ce cas, certaines valeurs non-affectées à la variable sont tout de même représentées ;
- Pour les données de type référence, on effectue l'union des ensembles de références des deux variables.

Dans les deux cas, il est possible de simplifier l'ensemble en lui substituant l'ensemble spécial $\{\Omega\}$.

Exemple Dans un même chemin d'exécution, la variable u se voit affecter successivement les valeurs 3, 4, 5 et 12. À la fin de l'évaluation du chemin d'exécution, la valeur abstraite de u sera la suivante :

$$V(u) = \{[3, 5], 12\}.$$

Si un autre chemin d'exécution affecte la valeur 50 à u, et que pour des raisons d'économie de mémoire on ne souhaite pas avoir plus de deux intervalles dans notre ensemble de valeurs, on peut factoriser les deux éléments dont les valeurs sont les plus proches au prix d'une perte de précision. Dans ce cas, on aura :

$$V(u) = \{[3, 12], 50\}.$$

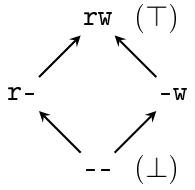


FIG. 4.4 – Treillis des états représentant les accès effectués à une donnée du système.

Le représentation n'est plus exacte, mais la règle de pessimisme dans la capture des états représentables est préservée.

4.1.2.4 Accès effectués sur une variable

Pour toute variable u du système, $A(u)$ retourne le type d'accès qui y est effectué, pour tous les états du système considérés.

Représentation Deux types d'opération d'accès peuvent être effectués sur une variable : lecture et écriture. L'accès à une variable peut donc être de quatre types : aucun accès effectué (`--`), accès en lecture seulement (`r-`), accès en écriture seulement (`-w`) ou accès en lecture-écriture (`rw`).

Unification Les quatre états d'accès forment un treillis représenté par la figure 4.4 et définissant la règle d'unification de la propriété d'accès.

Exemple Soit une variable u du système. Cette variable est lue au travers d'un chemin d'exécution, et écrite au travers d'un second. Sa propriété d'accès évolue donc de la manière suivante :

$$\begin{aligned} A(u) &\leftarrow \texttt{--} \vee \texttt{r-} = \texttt{r-} \\ &\leftarrow \texttt{r-} \vee \texttt{-w} = \texttt{rw}. \end{aligned}$$

Ces quatre propriétés (type, source, valeur et accès) d'une donnée sont regroupées dans une entité appelée **donnée abstraite**.

4.1.2.5 Définition d'une donnée abstraite

Représentation La donnée abstraite $D(u)$ d'une variable u regroupe les quatre propriétés que nous venons d'évoquer au sein d'un tuple :

$$D(u) = (T(u), S(u), V(u), A(u)).$$

$D(u)$ permet ainsi de représenter l'ensemble des informations capturées sur la variable u pour un ensemble d'états considérés. L'ensemble des données abstraites représentables est noté $\mathbb{D}$.

Unification Étant donné que la donnée abstraite est un tuple formé de propriétés indépendantes et unifiables, nous pouvons définir l'unification de deux données abstraites comme suit :

$$D(u) \cup D(v) = ((T(u) \cup T(v)), (S(u) \cup S(v)), (V(u) \cup V(v)), (A(u) \cup A(v))).$$

Opérations supportées Une donnée abstraite supporte deux opérations particulières :

$\sim$ permet de tester si deux données abstraites *peuvent* être égales, c'est-à-dire partagent au moins une valeur en commun ;

$\nsim$ permet de tester si deux données abstraites *peuvent* ne pas être égales, c'est-à-dire ont des ensembles de valeurs non-égaux.

Par exemple, pour les variables abstraites u, v et w telles que $V(u) = \{10, 20, 30\}$, $V(v) = \{30, 50\}$ et $V(w) = \{50\}$, les assertions suivantes sont vraies :

$u \nsim w$	Car elles n'ont aucune valeur en commun
$u \sim v$	Car elles partagent la valeur 30
$u \nsim v$	Car elles ont des variables non-communes

La donnée abstraite est ainsi l'équivalent de la donnée dans notre représentation abstraite du système. Tout comme dans l'état concret, les données abstraites sont adressées au travers de variables.

4.1.2.6 Variable abstraite

Une variable abstraite v, définie dans l'ensemble $\mathbb{V}$ de toutes les variables abstraites, permet d'associer à son identifiant v une donnée abstraite $D(v)$.

$$\mathbb{V} : \mathbb{D}.$$

De la même manière que l'état du système est défini par l'ensemble de ses variables, nous pouvons définir son état abstrait au travers de l'ensemble de ses variables abstraites.

4.1.2.7 L'état abstrait du système

Un état abstrait est une entité composée de variables abstraites permettant de représenter un ensemble d'états du système.

Représentation L'état abstrait est un ensemble de variables abstraites, associant une donnée abstraite à un identifiant. En guise de convention de notation, un état abstrait sera noté δ, et l'ensemble des états abstraits représentables sera noté Δ. Par opposition, les états concrets seront représentés par θ, et l'ensemble des états concrets représentables sera noté Θ. Par ailleurs, nous définissons les ensembles $\overline{\Theta_\theta} \subset \Theta$, comprenant tous les états accessibles à partir d'un état θ.

Une variable abstraite appartenant à un état abstrait peut être accédée au travers de l'opérateur « . ». Ainsi, $\delta.\alpha$ désigne la variable d'identifiant α dans l'état δ, tandis que $\delta'.\alpha$ désigne la variable portant le même identifiant, mais cette fois-ci dans l'état δ'.

$$\Delta : \mathbb{V}^*.$$

Unification L'unification d'un état abstrait δ avec un autre état abstrait δ' est un état abstrait δ'' composé de l'ensemble des variables des états δ et δ'. Les données référencées par des variables de même identifiant dans les deux états sont unifiées selon le procédé indiqué en 4.1.2.5 :

$$\delta'' = \delta \cup \delta' | \forall v \in \mathbb{V}, \text{ si } \exists \delta.v \text{ et } \exists \delta'.v \text{ alors } \exists \delta''.v \text{ tq } D(\delta''.v) = D(\delta.v) \cup D(\delta'.v).$$

La représentation des propriétés à capturer étant maintenant établie, il nous reste à définir comment s'opère leur capture proprement dite.

4.1.3 Capture des propriétés

Maintenant que nous avons identifié les propriétés du système que nous souhaitons capturer, et défini l'état abstrait capable de représenter ces informations pour un ensemble d'états du système considérés, nous avons besoin d'un algorithme capable de parcourir cet ensemble d'états et de les unifier dans un contexte abstrait δ afin qu'à l'issue de l'analyse ce dernier représente l'ensemble $\overline{\Theta_\theta}$ des états du système atteignables à partir de l'état θ analysé (représentés sur la figure 4.3).

De la même manière que l'interpréteur réel est capable de parcourir tous les états intermédiaires depuis un état donné jusqu'à un état final, nous devons définir un interpréteur abstrait, factorisant dans un état abstrait δ l'ensemble $\overline{\Theta_\theta}$.

Pour cela, il nous faut en premier lieu définir la granularité des transitions dans le graphe de parcours des états de l'interpréteur abstrait. Au niveau le plus fin, celui de l'interpréteur réel, une transition correspond à l'exécution d'un bytecode. Dans le cadre de l'interpréteur abstrait, nous avons décidé de définir les transitions entre états abstraits du système par l'exécution d'une opération sur le système, c'est-à-dire l'invocation d'une méthode. Le parcours effectué par l'interpréteur abstrait correspond ainsi à une analyse inter-procédurale du système. Voyons quelles propriétés des méthodes sont nécessaires à notre analyse.

4.1.3.1 Méthodes

Une méthode σ appartenant à l'ensemble $\mathbb{M}$ des méthodes du système comprend les éléments suivants :

- La séquence de bytecodes qui composent son corps de méthode, accessible par index au travers de l'opérateur $\sigma[index]$;
- Son type de retour déclaré, accessible par $\sigma.ret$.
- La taille occupée par ses arguments, accessible par $\sigma.nbargs$.

La spécification de la machine virtuelle Java [Lind 99] nous apprend que les méthodes Java s'exécutent au sein d'une fenêtre d'exécution, contenant ses paramètres, ainsi que ses variables locales et de travail.

4.1.3.2 Fenêtre d'exécution de méthode

Une fenêtre d'exécution de méthode est une séquence de données abstraites telle que $|f|$ soit sa taille et $f[i]$ son i^{e} élément. Cette séquence permet de représenter les arguments et variables locales d'une méthode, ainsi que sa pile d'exécution. On notera $\mathbb{F}$ l'ensemble des fenêtres d'exécutions représentables :

$$\mathbb{F} = \mathbb{D}^*.$$

Par ailleurs, $f.tos$ contient l'index du premier élément non-utilisé de la pile d'exécution.

Une fenêtre d'exécution accompagnée des paramètres permettant de lui associer un point d'exécution précis au sein d'une méthode constitue un contexte local d'exécution de méthode.

4.1.3.3 Contexte local d'exécution de méthode

Un contexte local d'exécution de méthode, noté l, est représenté par un tuple comprenant trois éléments $(m, frame, pc)$ tel que :

- m, accessible par $l.m$, représente la méthode en cours d'interprétation dans ce contexte ;
- $frame$, accessible par $l.frame$, est la fenêtre d'exécution de la méthode, telle que nous venons de la définir ;
- pc, accessible par $l.pc$, est un entier naturel qui représente l'index dans le corps de m du prochain bytecode à exécuter.

La taille en octets occupée par l'instruction de $l.m$ pointée par $l.pc$ est implémentée sous forme de fonction (notée $bcsize()$, accessible par $l.bcsize()$).

L'ensemble $\mathbb{L}$ des contextes de méthodes représentables est donc défini comme étant le produit cartésien de l'ensemble $\mathbb{M}$ des méthodes du système, de l'ensemble $\mathbb{F}$ des fenêtres d'exécution représentables, ainsi que de l'ensemble $\mathbb{N}$ des entiers naturels permettant de représenter l'index d'interprétation :

$$\mathbb{L} : \mathbb{M} \times \mathbb{F} \times \mathbb{N}.$$

Un contexte local de méthode l peut être construit à partir de l'invocation d'une méthode σ au sein d'un autre contexte local l'. Dans ce cas particulier, le contexte l est construit de la manière suivante :

- $l.m$ se voit affecter la méthode σ ;
- $l.frame$ est créée à partir des n derniers éléments de la fenêtre $l'.frame$, n étant le nombre de variables locales et d'arguments de la méthode σ ;
- $l.pc$ est initialisé à zéro, c'est-à-dire l'index du point d'entrée de la méthode σ.

Cette initialisation crée un contexte local d'entrée pour la méthode invoquée, initialisé avec les arguments de l'invocation. L'interprétation d'un contexte local de méthode permet de calculer le contexte de sortie qui lui est associé.

4.1.3.4 Contexte de sortie de méthode

La fin de l'interprétation d'un contexte local de méthode fournit le contexte de sortie r de celui-ci. Un contexte de sortie comprend la donnée abstraite représentant la valeur de sortie de la méthode (accessible par $r.out$), ainsi qu'un ensemble de données abstraites correspondant aux exceptions potentiellement lancées par la méthode (accessible par $r.exc$). L'ensemble $\mathbb{R}$ des contextes de sortie représentables est donc défini de la sorte :

$$\mathbb{R} : \mathbb{D} \times \mathbb{D}^*.$$

Un contexte de sortie étant le résultat de l'interprétation d'un contexte local dans un état donné du système, les deux peuvent être associés par une fonction spécifique.

4.1.3.5 Fonction de signature

Les contextes locaux et de retour sont liés par la fonction de signature ε, qui à partir d'un état abstrait δ donné, est capable d'associer un état de sortie r à tout contexte local l :

$$\varepsilon : \Delta \mapsto (\mathbb{L} \mapsto \mathbb{R}).$$

Ainsi, pour un état abstrait δ et un contexte local de méthode l, $\varepsilon(\delta)(l)$ donne, s'il a déjà été calculé, le contexte de retour r fourni par l'interprétation abstraite de l dans δ. Si celui-ci n'a pas encore été inféré, un contexte de retour « générique » est créé à partir du type de retour non-exact de la méthode pointée par le contexte local de la manière suivante :

$$\varepsilon(\delta)(l) = \{(T(l.m.ret), \Omega, \Omega, \texttt{--}), \{\varnothing\}\}.$$

Le calcul du contexte de retour associé à un contexte local est accompli par une analyse intra-procédurale.

4.1.3.6 Analyse intra-procédurale

Étant donnés un contexte local l, deux états abstraits δ et $\widehat{\delta}$, et une fonction de signature $\varepsilon(\widehat{\delta})$, la fonction I effectue une analyse intra-procédurale sur l et retourne les éléments suivants :

- L'état abstrait δ' reflétant les modifications sur δ détectées par l'interprétation de l ;
- Le contexte de sortie r de l sur δ, correspondant au résultat de $\varepsilon(\delta)(l)$;
- La liste des contextes locaux correspondants aux invocations de méthodes effectuées pendant l'interprétation de l sur δ.

I se définit donc de la manière suivante :

$$I : \Delta \times \mathbb{L} \times \varepsilon(\Delta) \mapsto \Delta \times \mathbb{R} \times \mathbb{L}^*.$$

La sémantique de chaque bytecode du jeu d'instruction Java devant être prise en charge, cette fonction est particulièrement volumineuse. Afin de l'expliciter sans être trop exhaustif, nous nous contenterons dans ces pages de formaliser l'algorithme général ainsi que les sémantiques des bytecodes `getfield` et `invokestatic`, représentatifs de l'ensemble des opérations réalisées par I.

FIG. 4.5 – Algorithme d'interprétation abstraite d'un contexte local, invoqué avec les paramètres δ, l et $\varepsilon(\widehat{\delta})$.

$callees \leftarrow \varnothing$
$\delta' \leftarrow \varnothing$
$r \leftarrow \varnothing$
$exceptions \leftarrow \varnothing$
$evalqueue \leftarrow \{l\}$
tantque $\exists cl \in evalqueue$ **faire**
  si $cl.frame \notin frames[cl.pc]$ **alors**
    $frames[cl.pc] \leftarrow frames[cl.pc] \cup \{cl.frame\}$
    $A(l.m[l.pc..l.pc + l.bcsize()]) \leftarrow A(l.m[l.pc..l.pc + l.bcsize()]) \vee$ `r-`
    $\varpi_{l.m[l.pc]}(cl, \delta, \varepsilon(\widehat{\delta}))$
  finsi
fin tantque
`return` $\delta', \{r, exceptions\}, callees$

Algorithme général À chaque instruction de la méthode $l.m$ est associé l'ensemble des fenêtres d'exécution abstraites rencontrées au cours des précédents passages de l'algorithme à cet endroit du code, au travers de la variable locale $frames$. Une autre variable nommée *callees* contient la liste des contextes locaux correspondant aux méthodes invoquées pendant l'interprétation de $l.m$. En outre, les variables *ret* et *exceptions* contiennent respectivement la valeur de retour abstraite de la méthode ainsi que les exceptions potentiellement lancées par celle-ci.

L'algorithme opère de la façon suivante : une file *evalqueue* des contextes locaux à évaluer est initialisée avec l. Puis, tant qu'il existe un contexte local cl de cette file, la procédure suivante est appliquée :

- Si $cl.frame$ est déjà présente dans $frames[cl.pc]$, l'interprétation de ce contexte prend fin ;
- $cl.frames$ est ajoutée à l'ensemble $frames[cl.pc]$;
- Les variables abstraites correspondant au bytecode $cl.m[cl.pc]$ et à ses arguments sont marquées comme lues pour notifier leur lecture possible par l'interpréteur de bytecode ;
- La fonction ϖ correspondant à la sémantique de l'instruction pointée par le contexte cl est appliquée. Celle-ci prend en argument le contexte cl correspondant à l'instruction à interpréter, l'état δ sur lequel cette évaluation s'effectue, ainsi que la fonction $\varepsilon(\widehat{\delta})$ des signatures connues. Elle met elle-même à jour les variables utilisées par l'algorithme en fonction de sa sémantique. Par exemple, si l'interprétation du bytecode révèle que celui-ci lance une exception, c'est ϖ qui l'ajoute dans la liste *exceptions* des exceptions lancées par la méthode. De même, ϖ ajoute elle-même les contextes locaux correspondant à ses suivants dans *evalqueue*.

$$\varpi : \mathbb{L} \times \Delta \times \varepsilon(\Delta).$$

Ce procédé est présenté sous une forme algorithmique par la figure 4.5.

Voyons maintenant la sémantique de quelques bytecodes implémentés par la fonction ϖ correspondante.

Traitement du bytecode `getfield` La sémantique du bytecode `getfield <x>` est implémentée par la fonction $\varpi_{\texttt{getfield}}$. Elle consiste à empiler la donnée située dans le champ `<x>` de l'objet o situé en sommet de pile. Du point de vue de l'interprétation abstraite, il faut également marquer la variable dont une copie est placée sur la pile comme étant lue.

Ce bytecode doit en outre gérer un cas exceptionnel : si $o = null$, alors une `NullPointerException` doit être lancée. S'il est avéré que que o a toujours cette valeur à cet endroit du code, l'interprétation de `getfield <x>` s'arrête là.

Nous distinguons ensuite deux cas, selon que les valeurs de o soient connues ou non.

Cas où la valeur de o n'est pas connue Si les valeurs possibles de o ne sont pas connues, alors la cible du bytecode peut être n'importe quel objet compatible présent dans le système au moment de sa capture, ou créé ultérieurement. Dans ce cas, il n'est possible d'inférer que le type non-exact de la cible ($\{T(o.x)\}$) ainsi que sa provenance (le champ `x` de n'importe quelle instance de $T(o)$ du système), comme le montre la figure 4.6(a).

La non-connaissance des valeurs de o a également un effet de bord important sur le système : toutes les instances de $T(o)$ présentes dans le système pouvant être concernées par ce bytecode, toutes doivent avoir leur champ `x` marqué comme étant lu.

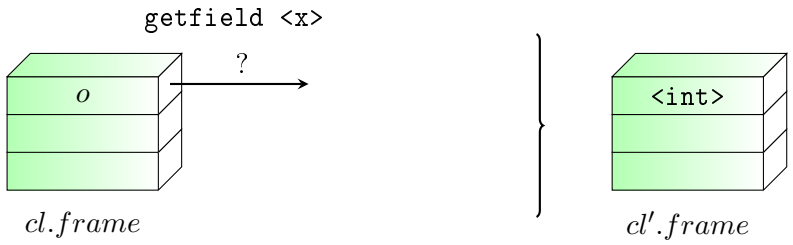
Cas où la valeur de o est connue Dans le cas où la donnée abstraite o est identifiable, une sémantique fournissant une donnée plus précise peut être employée. D'une part, le marquage du champ `x` en lecture ne concerne non plus toutes les instances du système compatibles avec $T(o)$, mais uniquement celles présentes dans l'ensemble $V(o)$. On peut alors obtenir le type et les valeurs exactes de la donnée α à empiler, ainsi que le montre la figure 4.6(b).

Selon le cas de figure, le marquage des variables concernées par la lecture est effectué et une variable abstraite α correspondant au résultat de l'interprétation du bytecode est créée. Le contexte local cl' à l'issue de l'exécution du bytecode peut alors être créé à partir de $cl.frame$ en remplaçant o par α. La figure 4.7 donne le détail de l'implémentation de la sémantique de `getfield <x>`.

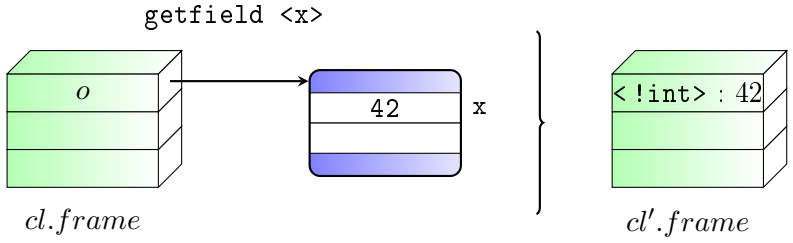
Traitement du bytecode `invokestatic <m>` Le bytecode `invokestatic <m>` est chargé d'invoquer la méthode statique `m` en lui donnant comme paramètres les données présentes au sommet de la pile. Si `m` retourne un résultat, celui-ci est empilé. La sémantique de `invokestatic <m>` est implémentée par la fonction $\varpi_{\texttt{invokestatic}}$.

L'interprétation abstraite des bytecodes d'invocation de méthodes n'évalue pas directement les contextes locaux correspondants aux méthodes appelées, pour éviter les phénomènes de récursivité. Au lieu de cela, le contexte local correspondant au point d'entrée de la méthode invoquée est ajouté dans la liste *callees* des contextes locaux correspondant aux méthodes appelées par la méthode en cours d'évaluation, afin que celle-ci soit évaluée après sa méthode appelante.

$\varpi_{\texttt{invokestatic}}$ a un suivant uniquement si la fonction appelée peut retourner normalement (c'est-à-dire que $r.out \neq \varnothing$). Si c'est le cas, le suivant est le contexte cl' composé de la pile d'interprétation purgée des arguments de la méthode invoquée, et sur laquelle le résultat éventuel est empilé. Les exceptions lancées par ce bytecode sont celles lancées par la méthode



(a) Cas général : les valeurs de o ne sont pas connues — on ne peut inférer que le type approximatif de la donnée du champ.



(b) Cas optimisable : les valeurs de o sont connues — on peut inférer le type exact de la donnée du champ, ainsi que ses valeurs possibles.

FIG. 4.6 – Comportement du bytecode `getfield` selon que les valeurs de o soient connues ou pas. Les noms donnés aux fenêtres d'exécution sont ceux des variables de la figure 4.7.

appelée, s'il y en a. Enfin, l'état δ n'est pas directement affecté par l'opération d'invocation en elle-même.

L'algorithme de traitement du bytecode est montré sur la figure 4.8, et la figure 4.9 montre l'évolution de la pile d'interprétation.

Nous avons détaillé le fonctionnement de la fonction I, permettant d'effectuer l'interprétation abstraite d'un contexte local donné, et de retourner le contexte de retour correspondant ainsi que la liste des contextes locaux correspondants aux invocations effectuées lors de l'interprétation. Nous pouvons nous servir de ce mécanisme pour effectuer l'analyse inter-procédurale du système, et ainsi couvrir tous les états potentiellement atteignables par celui-ci.

4.1.3.7 Analyse inter-procédurale du système

À gros grain, l'analyse du système consiste à effectuer une analyse inter-procédurale en invoquant la fonction I sur ses points d'entrée, puis en boucle sur les contextes locaux retournés par I, et ce jusqu'à ce que l'ensemble des contextes locaux atteignables par le système soit entièrement évalué. La fonction effectuant ce travail sera notée G. Elle prend en paramètre un état abstrait δ représentant les évolutions possibles connues de l'état évalué θ, la fonction de signature $\varepsilon(\widehat{\delta})$ comprenant toutes les signatures connues, ainsi que la liste *entries* des contextes locaux correspondants aux points d'entrée de l'évaluation. Elle retourne l'état abstrait δ' correspondant aux évolutions de δ inférées, ainsi que la fonction de signature $\varepsilon(\delta)$ des signatures calculées sur δ.

FIG. 4.7 – Sémantique de la fonction $\varpi_{\texttt{getfield <x>}}$.

$o \leftarrow cl.frame[cl.frame.tos - 1]$
si $o \sim null$ **alors**
  $exceptions \leftarrow exceptions \cup \{$ `!NullPointerException`$\}$
finsi
si $o \nsim null$ **alors**
  si $V(o) = \Omega$ **alors**
    $T(\alpha) \leftarrow T(o.x)$
    $S(\alpha) \leftarrow \bigcup_{obj \in \delta | T(o.x) \subset T(obj)} obj.x$
    $V(\alpha) \leftarrow \Omega$
    pour tout $\beta \in \delta | T(o.x) \subset T(\beta)$ **faire**
      $A(\delta'.\beta.x) \leftarrow A(\delta'.\beta.x) \cup$ `r-`
    fin pour
  sinon
    $T(\alpha) \leftarrow \bigcup_{val \in V(o)} T(val.x)$
    $V(\alpha) \leftarrow \bigcup_{val \in V(o)} V(val.x)$
    $S(\alpha) \leftarrow \bigcup_{val \in V(o)} \{val.x\}$
    pour tout $\beta \in V(o)$ **faire**
      $A(\delta'.\beta.x) \leftarrow A(\delta'.\beta.x) \cup$ `r-`
    fin pour
  finsi
  $A(\alpha) \leftarrow$ `--`
  $f \leftarrow cl.frame$
  $f[f.tos - 1] \leftarrow \alpha$
  $cl'.m \leftarrow cl.m$
  $cl'.frame \leftarrow f$
  $cl'.pc \leftarrow cl.pc + cl.bcsize()$
  $evalqueue \leftarrow evalqueue \cup \{cl'\}$
finsi

FIG. 4.8 – Sémantique de la fonction $\varpi_{\texttt{getfield <x>}}$

$f \leftarrow cl.frame[f.tos - m.nbargs..f.tos]$
$callee \leftarrow \{m, f, 0\}$
$callees \leftarrow callees \cup \{callee\}$
$r \leftarrow \varepsilon(\widehat{\delta})(callee)$
pour tout $exc \in r.excs$ **faire**
  $exceptions \leftarrow exceptions \cup \{exc\}$
fin pour
si $r.out \neq \varnothing$ **alors**
  $f' \leftarrow cl.frame$
  $f'.tos \leftarrow f'.tos - m.nbargs$
  si $T(r.out) \neq$ `void` **alors**
    $f'[tos] \leftarrow r.out$
    $f'.tos \leftarrow f'.tos + 1$
  finsi
  $cl' \leftarrow \{cl.m, f', cl.pc + cl.bcsize()\}$
  $evalqueue \leftarrow evalqueue \cup \{cl'\}$
finsi

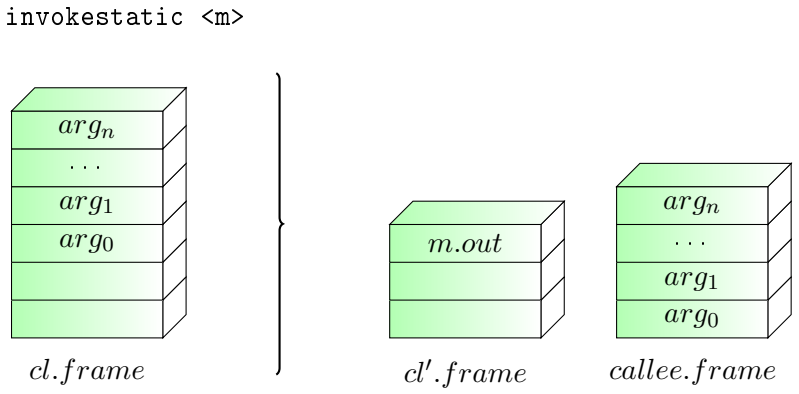


FIG. 4.9 – Évolution de la fenêtre d'exécution par l'exécution du bytecode `invokestatic <m>` (les noms des fenêtres correspondent aux noms de variables associées de la figure 4.8).

FIG. 4.10 – Algorithme d'analyse inter-procédurale, invoqué avec les paramètres (δ, $\varepsilon(\widehat{\delta})$, *entries*)

$\delta' \leftarrow \varnothing$
done $\leftarrow \varnothing$
evalqueue $\leftarrow$ *entries*
pour tout $c \in$ *evalqueue* **faire**
  $\delta'', r, callees \leftarrow I(\delta, c, \varepsilon(\widehat{\delta}))$
  $\delta' \leftarrow \delta' \cup \delta''$
  $\varepsilon(\delta)(c) \leftarrow r$
  done $\leftarrow$ *done* $\cup \{c\}$
  pour tout $c' \in$ *callees* **faire**
    si $c' \notin$ *done* **alors**
      evalqueue $\leftarrow$ *evalqueue* $\cup \{c'\}$
    finsi
  fin pour
fin pour
return $(\delta', \varepsilon(\delta))$

$$G : \Delta \times \varepsilon(\Delta) \times \mathbb{L}^* \mapsto \Delta \times \varepsilon(\Delta).$$

Dans G, δ et $\varepsilon(\widehat{\delta})$ contiennent la connaissance déjà inférée sur le système : δ contient les variables abstraites donnant des informations sur les états inférés comme atteignables depuis θ. $\varepsilon(\widehat{\delta})$ associe un contexte de retour aux contextes locaux déjà inférés.

Le parcours des contextes locaux de méthode atteignables est effectué selon l'algorithme décrit figure 4.10. La variable *evalqueue* contient une file des contextes locaux à évaluer. Elle est initialisée avec les contextes locaux d'entrée fournis en argument. Pour chaque contexte local c contenu dans *evalqueue*, la fonction I est invoquée sur le contexte, avec en paramètres l'état δ et la fonction $\varepsilon(\widehat{\delta})$ des signatures connues, fournissant la modification δ'' de l'état δ opérée par c, ainsi que son contexte de retour r et la liste *callees* des contextes locaux correspondant aux méthodes appelées depuis c. L'état δ' à retourner est alors unifié avec les modifications inférées par I sur le contexte évalué, et la fonction $\varepsilon(\delta)$ associe désormais le contexte local évalué au contexte de retour correspondant. Ce contexte local est enfin ajouté à la liste *done* pour assurer qu'il ne sera pas évalué une deuxième fois. Enfin, chaque contexte local c' invoqué par le contexte évalué c est ajouté à la liste *evalqueue* des contextes à évaluer, à condition qu'il ne figure pas déjà dans la liste *done*.

La fonction G d'analyse inter-procédurale, que nous venons de présenter, permet d'inférer un état abstrait δ' contenant une connaissance des états atteignables par δ à partir des points d'entrée donnés, avec l'aide de la connaissance de certains contextes de retour fournie par $\varepsilon(\widehat{\delta})$. Elle retourne par ailleurs les signatures qui ont été inférées durant le parcours, au travers de $\varepsilon(\delta)$. Il reste à savoir comment passer d'un état concret θ à l'état abstrait δ que prend cette fonction en paramètre, et quels sont les points d'entrée qui lui sont également fournis. Nous allons répondre à ces questions qui nous révéleront que G est en fait une fonction qui converge vers un état abstrait plus précis au fur et à mesure de ces appels.

4.1.3.8 Convergence de l'analyse

Nous sommes maintenant capables d'effectuer un parcours des contextes locaux de méthode atteignables depuis un ensemble de points d'entrée sur un état abstrait δ donné, et d'inférer ainsi un état δ' contenant une connaissance des états que le système peut atteindre au cours de cette analyse.

Cependant, l'analyse globale que nous devons faire s'effectue à partir d'un état concret θ. Afin de pouvoir lancer G, il est nécessaire de le convertir en un état abstrait δ reprenant toutes les informations qu'il contient. De plus, G doit recevoir en argument les contextes locaux correspondants aux points d'entrée de l'analyse du système. Enfin, au début de l'analyse, la fonction ε ne connaît aucune signature et est par conséquent vide. Les signatures étant calculées par I, il est nécessaire de procéder à plusieurs appels successifs de G pour pouvoir exploiter l'information de signature.

Voyons tout d'abord comment construire l'état abstrait initial regroupant les connaissances que nous avons du système à l'état θ, et comment déterminer les contextes locaux correspondants aux points d'entrée de l'analyse.

Construction de l'état abstrait initial L'état abstrait δ utilisé pour l'analyse du système est construit à partir de l'état capturé θ par la fonction ψ.

$$\psi : \Theta \mapsto \Delta.$$

Celle-ci opère de la manière suivante : pour chaque variable α de θ, est créée une variable abstraite homonyme dans δ telle que :

$$\begin{aligned} T(\delta.\alpha) &= \{!T(\theta.\alpha)\} \\ S(\delta.\alpha) &= \{\Omega\} \\ V(\delta.\alpha) &= \{V(\theta.\alpha)\} \\ A(\delta.\alpha) &= \texttt{--} \end{aligned}$$

Cette construction crée un état abstrait contenant toutes les informations connues sur l'état capturé : toutes les variables étant précisément connues dans l'état capturé, leur type ainsi que leur valeur peuvent être inférés exactement. La provenance des valeurs des variables n'est en revanche pas connue sur l'état capturé. Celle-ci est alors initialisée à $\{\Omega\}$, l'ensemble représentant toutes les provenances possibles. Enfin, l'indicateur d'accès est initialisé à `--`, les variables n'étant ni lues, ni écrites à ce moment.

Points de départ de l'interprétation L'interprétation abstraite que nous effectuons sur le système consiste à déterminer chacun de ses futurs états potentiellement atteignables, en partant de l'état capturé θ. Les points de départ de cette analyse sont les fenêtres d'exécution situées dans les piles d'exécution associées aux tâches actives du système. Chaque fenêtre présente sur une pile correspondant à une invocation de méthode, il convient de créer un contexte local pour chacune, en recopiant les données exactes de la fenêtre, la référence vers la méthode en cours d'exécution et le compteur d'exécution vers les champs correspondants du contexte local.

FIG. 4.11 – Algorithme global calculant l'état abstrait δ correspondant aux états atteignables à partir de θ.

$\delta' \leftarrow \{\varnothing\}$
$\delta \leftarrow \psi(\theta)$
$\varepsilon(\delta) \leftarrow \varnothing$
tantque $\delta \neq \delta'$ **faire**
  $\delta \leftarrow \delta'$
  $\delta', \varepsilon(\delta') \leftarrow G(\delta, \varepsilon(\delta), \rho(\theta))$
fin tantque
return δ

Certaines fenêtres, en cours d'invocation, doivent cependant retourner un résultat dans la fenêtre d'exécution de la méthode qui les appelle : ce résultat est obtenu par $\varepsilon(\widehat{\delta})(l)$, l étant le contexte local créé à partir de la fenêtre devant retourner un résultat.

La fonction ρ retourne ainsi tous les contextes locaux présents dans les piles d'exécution de l'état θ, ceux-ci étant des points d'entrées de l'interprétation dans l'état abstrait δ.

$$\rho : \Delta \mapsto \mathbb{L}^*.$$

Une fois que les points de départ de l'analyse sont disponibles, le parcours des contextes locaux de méthode atteignables peut s'effectuer.

Algorithme de convergence de G Mais si nous avons vu que G permet d'inférer une connaissance sur les états atteignables depuis δ, les données initiales que nous venons d'obtenir ne lui permet pas d'inférer tous les états atteignables. En effet, nos connaissances se limitent à l'état du système lors de sa capture. En particulier, aucune signature n'est disponible. Ces informations sont en fait calculées par G elle-même, qui est capable de les raffiner au fur et à mesure de ses itérations.

L'algorithme général permettant d'inférer l'état abstrait δ représentant tous les états atteignables à partir d'un état concret θ est implémenté par la fonction S :

$$S : \Theta \mapsto \Delta.$$

La figure 4.11 donne le détail de cet algorithme. Tout d'abord, un état abstrait δ initial est créé à partir de l'état capturé θ, en utilisant la fonction ψ. L'algorithme le fait ensuite évoluer vers un état δ' par invocation de la fonction G, qui infère tous les états atteignables à partir de δ. L'opération est alors répétée sur l'état δ' obtenu jusqu'à ce qu'un état stable soit atteint (c'est-à-dire que l'état δ' calculé au temps n soit identique à celui calculé au temps $n+1$). Cet état stable ainsi obtenu est alors retourné comme état l'état abstrait des états atteignables par θ le plus précis inférable par notre algorithme.

G converge ainsi vers un état stable pour les raisons suivantes :

- La fonction de signature $\varepsilon(\delta)$ est initialement vide. Les contextes de retour sont associés aux contextes locaux par la fonction G. Après plusieurs itérations de G, tous les contextes locaux atteignables ont été inférés et leurs contextes de retour sont connus précisément ;

– L'état δ initial ne comprend que les données correspondant à l'état θ, mais qui se complètent au fur et à mesure des itérations de G jusqu'à ce l'état inféré devienne stable.

Une fois que l'état abstrait δ obtenu est stable, il est possible d'exploiter son contenu afin de spécialiser le système.

4.2 Spécialisation du système

L'analyse que nous avons décrite dans la section précédente nous fournit, à partir de l'état capturé θ, un état abstrait δ du système riche en renseignements sur les possibles évolutions de ce dernier. Les exploitations de cette analyse que nous décrivons ici ont pour but de fournir un état θ' tel que $\theta' \in \theta$ et que chaque état concret de $\overline{\Theta_\theta}$ atteignable depuis θ le soit également à partir de θ'.

Nous allons décrire dans cette section quelques unes des spécialisations qu'il est possible d'effectuer à partir du résultat de notre analyse. La première spécialisation tire profit du fait que le système Java est un graphe d'objets pour supprimer les références inutiles entre objets, puis supprimer les objets non-référencés et ainsi effectuer une spécialisation à gros grain. Les autres spécialisations opèrent à grain plus faible, en reconcevant les objets restants : les objets réguliers sont remodelés sans leurs champs inaccédés, et les méthodes sont purgées de leur code mort. Nous verrons ensuite, dans la section suivante, comment propager ces spécialisations jusque dans l'environnement d'exécution natif qui est embarqué avec le système Java dans l'équipement cible.

4.2.1 Élimination des objets inatteignables

Les attributs de lecture/écriture présents dans chaque variable abstraite du système nous permettent de déterminer, parmi les variables de type référence, lesquelles ne sont jamais lues dans aucun des chemins d'exécution possibles. Ces références inexploitées constituent alors autant de liens inutiles entre les instances : si un objet n'est lié que par des références non-lues, c'est qu'il n'est jamais atteint au cours de l'exécution du système, et donc qu'il peut être collecté sans effet sur l'exécution de ce dernier.

4.2.1.1 Opération

Pour chaque variable v de l'état abstrait δ, l'attribut d'accès est interrogé pour savoir si v est lue dans un des chemins d'exécutions empruntables par le système. Si elle ne l'est pas, sa valeur est remplacée par *null* pour les variables de type référence, et par la valeur zéro pour les variables de type primitif.

Mise à *null* des références inutilisées La mise à *null* des références inutilisées a pour effet de couper des liens entre les objets, et donc d'isoler les objets qui ne sont jamais utilisés dans l'un des chemins d'exécution possibles du système. Une fois que toutes les variables ont été parcourues, il suffit de lancer le ramasse-miettes de l'`ObjectsPool` pour supprimer définitivement tous les objets non-référencés.

(a) Avant élimination. (b) Après élimination.

FIG. 4.12 – Exemple d'élimination des références inutilisées sur un tableau de quatre instances d'`Object`.

La figure 4.12 montre un exemple de résultat du procédé d'élimination des références inutilisées sur un tableau d'`Object` de quatre éléments. Tous les éléments du tableau pointent initialement vers un objet, mais l'attribut d'accès nous indique que deux d'entre eux ne sont jamais lus. Ces références peuvent donc être mises à *null*, car on peut être certain qu'elles ne seront pas déréférencées pendant l'exécution du système. Les objets qui deviennent non-référencés après cette opération sont alors collectés par le ramasse-miettes.

À noter que seul le positionnement en lecture d'une référence joue un rôle pour cette spécialisation. Si une référence est écrite mais jamais lue, la valeur qui s'y trouvait auparavant n'a aucune influence sur le comportement du système et peut donc être écrasée.

Mise à zéro des variables primitives inutilisées La mise à zéro des variables primitives inutilisées a pour effet d'augmenter la probabilité d'avoir des objets strictement identiques dans le système. L'intérêt de faire cela est de permettre de factoriser les objets identiques et constants, comme nous le verrons en 4.2.2.

4.2.1.2 Effet sur le système

Cette spécialisation opère à gros grain (sur les objets entiers) et permet ainsi d'obtenir la réduction du système la plus importante. La mise à *null* des références inutilisées effectue, en quelque sorte, un « déboisage » du système.

Effet sur les méthodes Bien que cette spécialisation supprime les objets non-atteints sans aucune discrimination, son effet principal est de supprimer les méthodes qui ne font pas partie du graphe d'appel. Cet effet est une des conséquences de la représentation 100% Java du système. En effet, la procédure d'appel des méthodes dans JITS est d'invoquer la méthode référencée à l'index donné en argument du bytecode d'invocation du *constant pool* ou de la table des méthodes virtuelles[2] d'une classe. Le *constant pool* et la table des méthodes virtuelles sont tous deux représentés par des objets Java, en l'occurence des tableaux de références vers des méthodes. Si l'analyse des types d'accès a révélé que certains indexes de ces tableaux ne sont jamais lus, ceux-ci sont mis à *null* et la méthode qu'ils référencent est alors potentiellement collectée si elle n'est pas référencée par l'index d'une autre table qui, elle, aura été marquée comme lue (figure 4.13).

[2]En fonction du type d'appel à effectuer : statique ou virtuel.

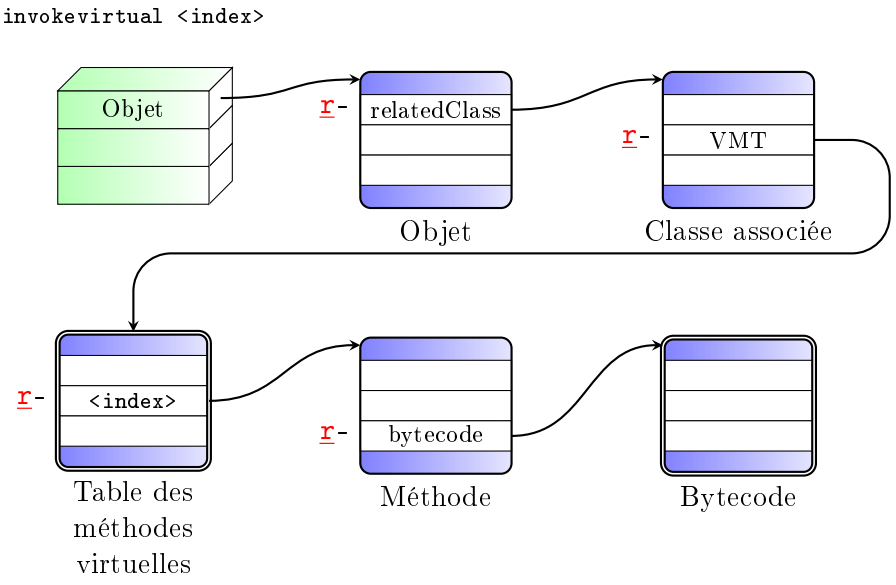


FIG. 4.13 – Marquage des champs réalisé lors de l'analyse du bytecode `invokevirtual`.

Il est intéressant de remarquer que cette opération est plus sûre et plus puissante qu'une simple suppression des méthodes non-atteintes par l'analyse inter-procédurale. D'une part, une méthode non-appelée peut être référencée et utilisée par un autre moyen, et la supprimer compromettrait alors le fonctionnement du système. D'autre part, en utilisant notre méthode ce sont non seulement les méthodes qui sont affectées par la spécialisation, mais également le mécanisme sous-jacent à leur appel. Ainsi, si aucune des méthodes d'une classe n'est appelée, non seulement toutes ses méthodes seront supprimées, mais également sa table des méthodes virtuelles puisqu'elle n'a jamais été marquée comme étant lue.

Effet sur les méta-données Les méta-données des classes sont également collectées par ce mécanisme si aucun appel n'est fait au chargeur de classes ou à l'API de réflexion. C'est vraisemblablement le cas des applications qui sont chargées entièrement dans le romizer.

Effet sur les chaînes de caractères L'efficacité de cet algorithme sur le cas particulier de l'élimination des chaînes de caractères constantes est également remarquable. En effet, les chaînes de caractères déclarées immédiatement dans le code sont référencées exclusivement au travers du constant pool. Leur accès est donc toujours effectué sans ambiguïté de valeur ; ainsi est-il possible de marquer très précisément lesquelles sont effectivement utilisées, et d'éliminer toutes les autres.

Efficacité par rapport à l'état d'avancement du système L'efficacité de cette spécialisation est grandement dépendante de l'état d'avancement du système au sein du romizer. En effet, si les classes applicatives ne sont pas encore chargées, les chemins d'exécution possibles passent par le chargeur de classes qui référence les méta-données des classes et obligera donc leur préservation, quand bien même celles-ci ne sont pas utilisées au niveau applicatif.

Il est donc impératif, pour pouvoir bénéficier d'une qualité de spécialisation satisfaisante, de déployer le système le plus possible hors-ligne.

Cette spécialisation, en supprimant les liens et par conséquent les objets inutiles au fonctionnement futur du système, permet de spécialiser ce dernier à gros grain. Les objets restants peuvent alors être reconçus au travers de spécialisations plus fines. Nous distinguons notamment la reconception de la structure des objets réguliers, ainsi que l'élimination du code mort au sein des méthodes.

4.2.2 Factorisation des objets identiques constants

Cette spécialisation regroupe deux instances strictement identiques et constantes en une seule.

4.2.2.1 Opération

Après analyse, il est probable qu'un certain nombre d'objets soient binairement identiques (même classe, même données). Certains de ces objets peuvent en outre n'être accédés qu'au travers de variables constantes (c'est-à-dire dont l'attribut de lecture est `r-`). Les objets identiques et constants peuvent être factorisés de manière sûre en une seule instance, ce qui permet de récupérer l'espace occupé par les autres instances identiques.

4.2.2.2 Effet sur le système

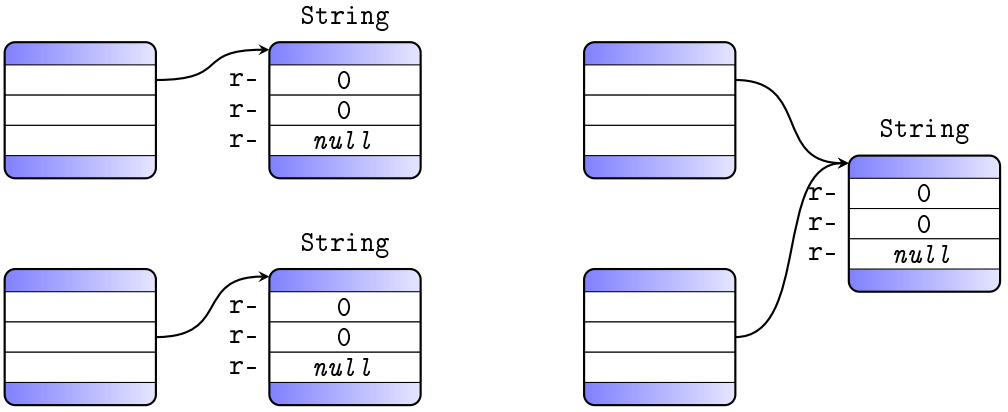
Cette factorisation, combinée avec la mise à zéro des champs primitifs inutilisés que nous venons de décrire, a un effet particulièrement bénéfique sur les chaînes de caractère restantes. Les chaînes de caractères comprennent une référence vers le tableau de caractères composant leur contenu, ainsi que deux champs donnant les indexes de début et de fin de la chaîne au sein de ce tableau. La spécialisation précédente a pour effet de transformer les chaînes de caractère dont le contenu est inexploité en chaînes dont les indexes sont tous deux égaux à zéro et dont la référence vers le tableau de caractères pointe à *`null`* : ces instances sont alors toutes identiques. Comme toutes les chaînes de caractères sont strictement constantes en Java, il devient donc possible de regrouper ces instances multiples en une seule, et de récupérer de l'espace occupé inutilement (figure 4.14).

4.2.3 Reconception de la structure des objets

Les objets réguliers sont composés de champs pouvant contenir une donnée primitive ou de type référence. Ces champs, déclarés au niveau des classes, se retrouvent (à l'exception des champs statiques) dans toutes les instances. Or, il peut arriver qu'aucun des chemins du système n'accède à un champ particulier, que ce soit en lecture ou en écriture (indicateur d'accès du champ égal à `--`). Il est alors possible et sûr de supprimer le champ concerné de la classe ainsi que des instances de celle-ci, ce qui permet de réduire leur taille.

4.2.3.1 Opération

L'observation des indicateurs d'accès des variables de type champ permet de déterminer celles qui ne sont jamais accédés, ni en lecture, ni en écriture, à aucun moment de l'exécution



(a) Deux chaînes de caractères non-utilisées et constantes sont référencées dans le système...

(b) ... il est alors possible de les factoriser en un seul objet.

FIG. 4.14 – Effet de la factorisation des objets identiques constants.

du système et pour aucune instance. Ces champs occupent de l'espace inutilement, qui peut-être récupéré par reconception des structures de données des objets et des classes Java.

Les champs éligibles pour cette spécialisation sont soit les champs statiques dont le marquage indique qu'ils ne sont jamais accédés, soit les champs non-statiques dont le marquage pour chacune de leurs instances indique qu'ils ne sont pas accédés. Il n'est pas possible de supprimer un champ d'une partie seulement des instances d'une classe. En effet, les instances étant intimement liées à leurs classe, il est impossible d'avoir deux instances d'une même classe ayant une structure différente.

La suppression des champs se fait en différentes étapes. Dans un premier temps, les méta-données des classes sont mises à jour. Dans un second temps, les instances ou la zone statique de la classe sont remodelées. Enfin, le bytecode des méthodes Java référençant les autres champs de la classe est mis à jour.

Mise à jour des méta-données Dans JITS, les méta-données associées à un champ comprennent son nom, son type et son index. L'index correspond au décalage à ajouter à l'adresse de début de l'objet ou de la zone statique de la classe pour accéder au champ, selon que celui-ci soit non-statique ou statique. Toute classe Java chargée comprend une liste des champs reprenant ces informations. La première chose à faire est de supprimer toute trace d'existence du champ dans cette liste, en y retirant l'entrée correspondante. Les indexes des champs suivant le champ supprimé sont alors décrémentés de la taille du champ supprimé[3].

Cette mise à jour est également répercutée au niveau du cache des méta-données maintenu par l'`ObjectsPool` (voir 3.3.2). Le `VMStructField` correspondant au champ à enlever est supprimé, et les indexes des champs suivants sont décrémentés de la taille du champ supprimé.

Une fois les méta-données mises à jour, les instances de la classe sont remodelées pour correspondre à la nouvelle forme.

[3]Cette opération n'est bien entendu réalisée que si les méta-données des classes sont toujours présentes dans le système après l'élimination des objets inatteignables.

Remodelage des instances Durant cette phase, les instances de la classe modifiée sont remodelées de façon à ce qu'elles correspondent à sa nouvelle structure. On distinguera ici deux modes opératoires, selon que le champ supprimé soit statique ou pas.

Champs statiques Ces champs sont stockés dans la zone statique de la classe à laquelle ils appartiennent et n'ont pas de manifestation par instance. Leur suppression se fait par simple redimensionnement du tableau des champs statiques, et décalage des entrées correspondant aux champs ayant un index plus élevé.

Champs d'instances Pour ces champs, la tâche est plus lourde que pour les champs statiques. En effet, chaque instance de la classe modifiée doit être remodelée. Cela se traduit par une recherche de toutes les instances de cette classe ou d'une de ses sous-classes. Celles-ci sont alors redimensionnées pour être conformes à la nouvelle taille de l'objet, et les valeurs y sont recopiées avec un décalage pour les champs suivant le champ supprimé.

Cette modification de la structure des objets a, à son tour, des répercutions sur le bytecode des méthodes, qui accèdent aux champs directement à partir de leur décalage.

Mise à jour du code des méthodes Le bytecode Java, dans sa forme originale, accède aux champs en passant par les instances de `Field` pour connaître leur index. Les bytecodes `getfield` et `putfield` prennent en effet comme argument un index du constant pool de la classe courante qui pointe vers l'instance de `Field` correspondante.

Cependant, pour des raisons de performance, JITS introduit des variantes accélérées de ces bytecodes, nommées `getfield_quick` et `putfield_quick`. Celles-ci remplacent les bytecodes originaux durant la phase d'édition des liens des classes (voir 2.3.2.2). Ces variantes changent la sémantique du bytecode en ce que l'argument n'est plus un index du constant pool pointant vers une instance de `Field`, mais le décalage à appliquer à l'objet au sommet de pile ou à la zone statique de la classe pour accéder au champ. En plus de rendre l'exécution du bytecode plus rapide, ces variantes permettent de ne plus dépendre des méta-données des classes une fois que leur édition des liens est effectuée [Ripp 04a].

L'index de certains des champs de la classe modifiée ayant été changé, le bytecode de toutes les méthodes du système doit donc être réévalué afin de corriger les arguments de ces bytecodes si leur cible a changé de position dans l'objet ou la zone statique de la classe.

4.2.3.2 Effet sur le système

Cette spécialisation opère à grain faible : ses effets se limitent à la classe dont le champ est supprimé et à ses instances. Elle n'a pas d'effet de bord particulier en dehors de la réduction de la taille des zones statiques ou des instances de classe.

À noter qu'une variante de cette technique peut s'appliquer sur les champs uniquement écrits. L'opération est la même, avec en plus le remplacement des bytecodes d'écriture au champ supprimé par le bytecode `pop` correspondant à la taille du champ.

4.2.4 Élimination du code mort

Durant l'exécution du système, certaines méthodes ne sont jamais appelées et peuvent être supprimées par élimination des objets inatteignables (voir 4.2.1). Les autres méthodes sont appelées, mais l'intégralité de leur code n'est pas forcément utilisée. Des exemples de tel code sont le code de gestion des exceptions alors qu'il a été déterminé qu'aucune exception n'est lancée dans la méthode, ou du code exécuté sur une condition qui a pu être totalement inférée pendant l'analyse. Du fait du caractère spécialisé du système, ce genre de code est fréquent, notamment dans le code de l'API Java. Ainsi, une méthode d'API lançant une exception en cas de passage d'arguments invalides, mais appelée dans un contexte toujours valide, ne jette jamais d'exception. La partie du corps de la méthode gérant les erreurs n'est donc jamais atteinte, et peut être supprimée du corps de la méthode.

4.2.4.1 Opération

Le marquage en lecture des bytecodes parcourus par l'analyse intra-procédurale nous permet de savoir par où l'interpréteur réel est susceptible de passer. En effet, le bytecode des méthodes est une donnée Java classique de type tableau de byte, et est donc marqué en lecture au fur et à mesure que l'interpréteur abstrait le parcourt, comme montré sur la figure 4.5. Les indexes non-marqués du tableau de bytecode n'étant pas atteignables par l'interpréteur réel, ceux-ci peuvent être supprimés et le bytecode redimensionné. Cette redimension implique de mettre à jour les instructions de saut présentes dans la méthode : leurs indexes de destination seront décrémentés de la quantité de code mort situé entre elles et leur destination.

4.2.4.2 Effet sur le système

Cette spécialisation est une spécialisation à grain fin sur une classe particulière d'objets : les tableaux de `byte` servant à stocker le bytecode des méthodes. Cette classe nécessite un traitement particulier car son opération est basée sur la sémantique des bytecodes de saut. C'est pourquoi le remodelage simple des tableaux n'est pas suffisant.

Toutes les spécialisations que nous avons étudiées concernent la couche haute (Java) du système. Or, celui-ci s'exécute sur un environnement d'exécution embarqué qui sert d'interface entre le système Java et le matériel. Il est également souhaitable de voir ce dernier spécialisé en fonction de l'usage réel qui en est fait.

4.3 Spécialisation de l'environnement d'exécution embarqué

Cette spécialisation est une extension de la précédente à l'environnement d'exécution natif. L'environnement JITS est, comme nous l'avons dit en 3.3.1, écrit autant que possible en Java. Cette partie importante du système est efficacement spécialisée par les procédures que nous venons de décrire — cependant, une partie de l'environnement d'exécution, celle ayant trait aux interactions avec le matériel et aux opérations de bas niveau, reste écrite en langage natif et ne peut donc pas profiter directement de ces spécialisations. Cette section montre comment nous permettons la spécialisation de l'environnement d'exécution embarqué sur lequel le système Java s'exécute.

La partie native du système comprend les éléments suivants :

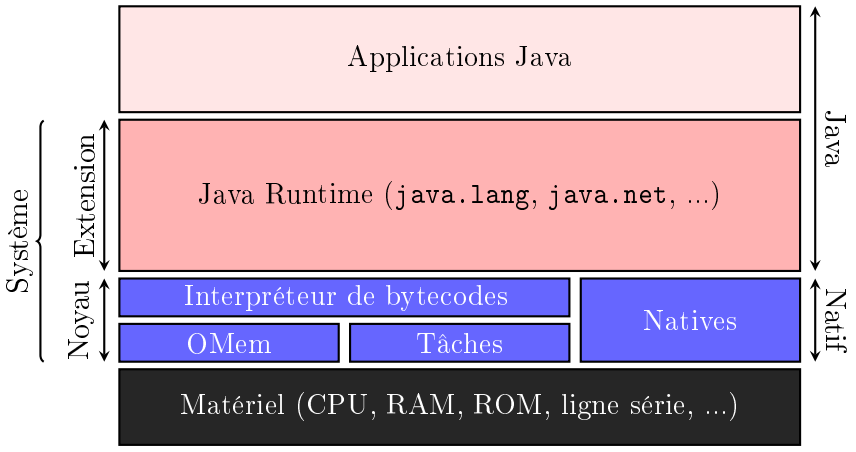


FIG. 4.15 – L'architecture de JITS, en maximisant la proportion du système écrite en Java, s'inspire des micro-noyaux.

- L'interpréteur de bytecode, chargé d'exécuter le système en respectant la sémantique des bytecodes Java ;
- Le gestionnaire mémoire, qui gère les différents types de mémoire présents sur l'équipement, l'allocation d'objets, ainsi que leur collection ;
- Le gestionnaire de tâches, qui fait basculer le système d'une tâche à une autre ;
- Les implémentations des méthodes natives.

Tous ces composants sont architecturés comme sur la figure 4.15. Nous désirons être en mesure de spécialiser chacun d'entre eux efficacement, et au besoin de renoncer à les embarquer s'ils ne s'avèrent pas utiles à l'exécution du système. Bien qu'ils ne soient pas écrits en Java et que par conséquent ils ne soient pas parcourables par notre analyse, ils sont comme tout composant logiciel formés de code et de données qui se référencent. En supprimant ces références, certaines de ces données se retrouvent isolées et peuvent être éliminées par ramasse-miettes, selon le modèle utilisé en 4.2.1. Notre façon de faire est donc de supprimer des références existantes entre la couche Java et la couche native du système, de façon à supprimer certaines des fonctions de la couche native devenues inutilisées. Afin de pouvoir opérer, ces spécialisations sur la couche native du système doivent tout d'abord avoir conscience de cette dernière, ainsi que de ses interactions avec les parties plus hautes écrites en Java.

4.3.1 Les interfaces entre environnement Java et environnement natif

La figure 4.15 montre que les composants natifs au contact de la couche Java sont l'interpréteur de bytecodes et les méthodes natives. On ajoutera qu'il existe également des références depuis la partie native du système vers des champs statiques de classe.

4.3.1.1 Les bytecodes

Cette première interaction vient directement du fait que le bytecode Java soit interprété. Chaque bytecode présent dans un des chemins d'exécution possibles du système est une réfé-

rence implicite vers son implémentation dans l'interpréteur de bytecodes de l'environnement natif. C'est en effet la rencontre d'un bytecode donné qui va diriger l'interpréteur natif vers son implémentation. Les parties de l'interpréteur implémentant des bytecodes jamais rencontrés dans le système ne sont jamais empruntées, et peuvent ainsi être supprimés de ce dernier.

4.3.1.2 Les champs statiques

Dans JITS, les couches profondes du système sont écrites autant que possible en Java. Pour cette raison, la partie native du système doit parfois utiliser des données présentes dans cette dernière. Ces accès se font au travers des champs statiques, car ce sont les seules entités du système référençables avec un nom fixe. Le migreur (voir 3.3.3.5) émet ainsi pour chaque champ statique une macro d'accès à nom fixe, qui référence la zone statique correspondante de la classe auquel ce champ appartient.

Un exemple d'une telle utilisation se trouve dans le gestionnaire de tâches. Les politiques d'ordonnancement des tâches sont écrites en Java, en utilisant la classe interne `Scheduler`. L'instance à utiliser pour la politique globale du système est accessible via le champ statique `scheduler` de la classe interne `Engine`. Lorsque la partie native a besoin d'opérer un changement de tâche (parce qu'une tâche a consommé son quantum de temps, ou parce qu'un moniteur a été relâché), elle appelle la méthode correspondante de l'instance de `Scheduler`. Pour ce faire, elle a besoin de passer par le champ statique correspondant de la classe `Engine`.

4.3.1.3 Les méthodes natives

Les appels aux méthodes natives sont le dernier type de lien entre l'environnement Java et l'environnement natif : le code de la méthode exécutée ne se situe pas dans la couche Java, et l'exécution passe d'un mode interprété à un mode natif.

Ces méthodes natives, exprimées en C, ne sont pas analysables par notre analyse intra-procédurale. Or, il est possible qu'elles aient un effet de bord sur l'état du système, par exemple en accédant à des champs d'instance ou statiques, ou en appelant d'autres méthodes. Ces effets de bord doivent absolument être reportés dans l'état abstrait δ inféré. Afin de ne pas omettre des variables ou des chemins d'exécution importants pour le système, ces effets doivent donc être exprimés de telle sorte que l'analyseur puisse les connaître, et marquer les éléments du système nécessaires à l'exécution des méthodes natives invoquées.

Expression des dépendances des méthodes natives Nous définissons, pour chaque classe implémentant une ou plusieurs méthodes natives, un fichier de description indiquant pour chaque méthode native les champs auxquels elle accède, les méthodes qu'elle appelle, et les exceptions qu'elle peut potentiellement lancer. Cette description doit être maintenue en même temps que le code de la méthode pour préserver la consistance entre l'analyse du système et son exécution réelle. Ainsi, l'analyseur est capable, lorsqu'il rencontre une méthode native, d'effectuer les opérations liées aux accès aux champs et aux appels de méthodes.

Le listing 4.1 montre les dépendances natives de la classe `java.lang.System`. Celle-ci ne propose qu'une méthode native, `arraycopy`, qui dépend de la présence de la classe `byte` dans le système, ainsi que des champs `DEFAULT` des classes `NullPointerException`, `ArrayStoreException` et `ArrayIndexOutOfBoundsException`, accédés en lecture. Lorsque

Listing 4.1 – Fichier de description des dépendances de la classe `java.lang.System`.

```
1  <?xml version='1.0' encoding='iso-8859-1'?>
2  <dependencies>
3  <class name="java.lang.System">
4    <method name="arraycopy"
5      descriptor="(Ljava.lang.Object;ILjava.lang.Object;II)V">
6      <require>
7        <class name="[B"/>
8        <field classname="java.lang.NullPointerException" name="DEFAULT"
9          access="r"/>
10       <field classname="java.lang.ArrayStoreException" name="DEFAULT"
11         access="r"/>
12       <field classname="java.lang.ArrayIndexOutOfBoundsException"
13         name="DEFAULT" access="r"/>
14     </require>
15   </method>
16 </class>
17 </dependencies>
18 <out>
19   <throws exception="java.lang.IndexOutOfBoundsException"/>
20   <throws exception="java.lang.ArrayStoreException"/>
21   <throws exception="java.lang.NullPointerException"/>
22 </out>
```

l'analyseur opère sur cette méthode, il se sert de cette description pour effectuer les opérations d'accès aux champs indiqués, ce qui a pour conséquence de les marquer en fonction de l'accès indiqué et de les inclure dans le système final, préservant ainsi le fil de l'interprétation jusque dans les couches natives. Le contexte de retour est précisé après l'expression des dépendances, en listant les exceptions lancées par cette méthode.

On s'aperçoit que l'expressivité du langage de description des dépendances est plus limitée que celle permise par l'interpréteur abstrait. Ainsi, certaines informations sur les variables abstraites sont perdues lors de l'interprétation d'un appel vers une méthode native : c'est le cas pour les valeurs exprimées dans le code natif, qui ne peuvent pas être inférées par l'interpréteur. Quel que soit le degré de précision des variables passées en entrée d'appel, le résultat de l'interpréteur reste toujours générique, car dans la version actuelle l'expression des dépendances ne permet pas de donner des informations précises sur le contexte de retour d'une méthode. L'analyse gagnerait en précision si ce contexte était mieux exprimable, d'autant plus qu'avec JITS les méthodes natives effectuent des opérations généralement simples et courtes.

L'expression des dépendances natives étant faite, reste le problème de leur inclusion dans le système final. De la même manière que la spécialisation du système Java permet de supprimer les méthodes Java inutilisée par ce dernier, il est désirable de ne pas embarquer les méthodes natives qui ne sont jamais appelées. Ce problème rentre dans le cadre plus large de l'élimination des références entre les deux environnements.

4.3.2 Élimination des références entre environnement Java et environnement natif

La machine virtuelle fournit de nombreux types de services. À très gros grain, on peut ainsi identifier l'interpréteur de bytecodes, le gestionnaire mémoire et le gestionnaire de tâches, comme le montre la figure 4.15. Cependant, l'interpréteur de bytecodes comprend plus de 200 sémantiques pour les bytecodes et le gestionnaire mémoire comme l'interpréteur de tâches remplissent différentes fonctions qui ne sont pas nécessairement utiles pour un système Java donné. De la même manière, l'API Java spécifie de nombreuses méthodes natives qui ne sont pas toutes nécessaires.

4.3.2.1 Méthodes natives non référencées

Dans JITS, le protocole d'appel d'une méthode, quel que soit son type, passe par l'accès au bytecode ou au code natif via l'instance de la classe `Method` correspondant à la méthode appelée. Le code d'une méthode, qu'il soit bytecode ou code natif, est donc inaccessible pour le système si cet objet n'est pas présent. Les méthodes non-référencées étant supprimées du système par l'élimination des objets inatteignables (voir 4.2.1), la présence de l'instance de `Method` correspondant à une méthode donnée suffit pour conditionner l'inclusion d'une méthode native à l'environnement d'exécution embarqué.

Cette inclusion conditionnelle se fait de la manière suivante : pour chaque instance de `Method` présente dans le système, le migreur déclare dans le fichier `rom.h` une macro qui sert à déclencher l'inclusion de la méthode native correspondante au système. Ces dernières sont encadrées par des `#ifdef` conditionnant leur inclusion à la déclaration de leur macro correspondant dans le fichier `rom.h`.

Un premier degré de modularité est ainsi atteint dans la couche native du système. Cependant, la partie la plus importante de cette couche est la machine virtuelle, pour laquelle la granularité de spécialisation n'est pas aussi aisément définissable.

4.3.2.2 Bytecodes non utilisés

Comme nous l'avons vu, les bytecodes constituent l'interface principale entre les parties Java et native du système. Nous comptons sur la suppression de certains bytecodes ainsi que sur l'absence de certaines fonctions natives référençant les autres modules de la machine virtuelle pour pouvoir supprimer des fonctions de ces derniers. Cette approche est en fait le prolongement dans la partie native du système de la logique appliquée lors de l'élimination des objets inatteignables (voir 4.2.1) : de même que l'on supprime les références non-utilisées dans le système pour collecter les objets non-référencés, on peut supprimer les fonctions non-référencées de l'environnement d'exécution natif pour collecter les parties natives non-atteintes. L'efficacité de cette spécialisation dépend principalement de la quantité de bytecodes utilisés par l'environnement Java.

De l'utilisation des bytecodes dans les programmes Java Le jeu d'instructions Java couvre un ensemble d'opérations très large : arithmétique et logique sur entiers et flottants de 32 et 64 bits, invocation de méthodes, synchronisation de tâches, etc. Cependant, peu de programmes Java font usage de toutes ces fonctions — et particulièrement les petits

programmes. Par exemple, beaucoup d'applications embarquées n'utilisent pas les nombres flottants, à tel point que leur support n'a pas été inclus dans la spécification Java Card. Il est également commun, pour des applications embarquées suffisamment déployées, de ne jamais allouer de mémoire.

Ainsi, comme nous le verrons dans le chapitre des résultats expérimentaux, un algorithme de chiffrement n'utilise que 90 bytecodes du jeu d'instructions Java, tandis qu'un programme aussi complexe qu'un lanceur de rayons n'utilise que 131 bytecodes différents.

Les bytecodes inutilisés peuvent donc être supprimés de l'interpréteur embarqué. Cette suppression a pour effet de casser des références entre l'interpréteur de bytecodes et les autres modules de la machine virtuelle. Par exemple, le bytecode `new` est chargé d'allouer la quantité de mémoire nécessaire pour stocker un objet d'une classe donnée. Pour ce faire, il appelle une fonction du gestionnaire mémoire chargée d'allouer la mémoire nécessaire. Si la mémoire libre disponible est insuffisante, cette fonction appelle le ramasse-miettes. Ce mécanisme est reproduit, avec d'autres fonctions, pour les bytecodes `newarray` et `anewarray`. Si le bytecode `new` est supprimé de l'interpréteur, la fonction d'allocation d'objet n'est plus référencée nulle part dans le code de la machine virtuelle. Si aucun des bytecodes d'allocation de mémoire n'est utilisé, alors non seulement leurs fonctions d'allocation respectives ne sont plus référencées, mais également le ramasse-miettes[4], qui peuvent donc tous être supprimés du système. Le gain obtenu en éliminant tous les bytecodes d'allocation mémoire est plus important que leur gain individuel cumulé.

Tous les bytecodes n'ont pas le même intérêt à être supprimés. Les bytecodes d'allocation mémoire apportent un gros gain en termes d'empreinte mémoire, car ils s'appuient sur des mécanismes relativement imposants. Mais d'autres bytecodes à la sémantique très simple, comme ceux gérant l'arithmétique, n'offrent qu'un gain négligeable. Nous avons remarqué que les deux tiers de l'empreinte mémoire de la machine virtuelle embarquée de JITS sont utilisés pour implémenter un dixième des bytecodes : ceux qui sont responsables de l'allocation mémoire, de la synchronisation des tâches, du lancement des exceptions et de l'invocation des méthodes.

Cependant, toutes les fonctionalités de la machine virtuelle ne sont pas dépendantes de la présence de certains bytecodes. Par exemple, un changement de tâche peut être déclenché par un bytecode (`monitorenter` ou `monitorexit`), mais également par d'autres évènements (par exemple, une tâche a utilisé l'intervalle de temps qui lui était consacré). Afin de décider de l'utilité de ces fonctionalités, il est nécessaire de procéder à une autre analyse du système.

Analyse des conditions d'utilisation du système Certaines des fonctionalités de la machine virtuelle, comme la gestion des tâches, sont référencées par l'interpréteur de bytecodes indépendamment de la présence ou non de certains bytecodes. Ces mécanismes sont en effet toujours appelés par la machine virtuelle elle-même.

Suppression de la gestion des tâches Par exemple, une fois qu'un quantum de temps est écoulé, la machine virtuelle change de tâche à exécuter. La désactivation de ces fonctionalités passe par une directive particulière de compilation, qui sera donnée pour les systèmes ne nécessitant pas leur utilisation.

[4] À la condition que la méthode `System.gc()` ne soit pas non plus référencée dans le code Java.

La gestion des tâches peut ainsi être purgée de la machine virtuelle sous les conditions suivantes :

- Il n'existe qu'une seule tâche active au moment de l'analyse ;
- La méthode `Thread.start()` n'est jamais invoquée par un des chemins atteignables par le système ;
- Aucun code n'est jamais chargé depuis l'extérieur.

Dans ces conditions, il est garanti que le système n'exécutera jamais plus d'une tâche à la fois, et les mécanismes associés de la machine virtuelle peuvent être désactivés en toute sécurité, coupant le seul lien hors bytecodes qui existe entre l'interpréteur de bytecodes et le gestionnaire de tâches.

L'analyse des conditions de désactivation de la gestion des tâches peut entrer en conflit avec la désactivation du support pour certains bytecodes. Par exemple, considérons un système qui remplit toutes les conditions pour être monotâche, mais dont les chemins d'exécution possibles rencontrent des instances de `monitorenter` ou `monitorexit` (par exemple, lors de l'appel d'un méthode synchronisée). Dans ce cas, le lien interpréteur/gestionnaire de tâches est bien coupé, mais d'autres liens existent du fait de la présence des bytecodes de gestion des tâches. Or, dans un système monotâche, l'effet de ces derniers est nul. On pourra, dans ces cas-là, supprimer toute instance des bytecodes `monitorenter` et `monitorexit` dans le système Java afin de supprimer les liens entre interpréteur et gestionnaire de tâches, et gagner encore un peu d'espace sur la taille des méthodes.

Une analyse similaire peut-être effectuée afin du supprimer la gestion des exceptions de certains bytecodes.

Suppression de la gestion des exceptions pour certains bytecodes Des bytecodes comme `getfield`, ou tout autre bytecode déréférançant une référence, peuvent potentiellement lancer une exception si certaines conditions de sûreté ne sont pas remplies. L'analyse du système est capable de détecter, à condition que l'information contextuelle soit suffisante, si ces conditions peuvent être remplies. Si tel n'est jamais le cas pour toutes les instances d'un bytecode, alors sa gestion des exceptions est inutile et peut être enlevée de sa sémantique.

Cette spécialisation est très locale, mais permet néanmoins de gagner un peu d'espace sur l'empreinte mémoire de l'interpréteur de bytecodes. De plus, elle coupe les références existant entre l'interpréteur et les exceptions à lancer. Enfin, on peut s'attendre à un léger gain dans l'exécution du système du fait que les conditions exceptionnelles ne sont plus testées.

Afin de permettre le degré de flexibilité requis dans l'interpréteur de bytecodes, nous avons choisi de générer le code de ce dernier à partir des informations inférées sur le système.

Génération de l'interpréteur de bytecodes L'interpréteur de bytecode est principalement composé d'une boucle se branchant sur diverses implémentations de bytecodes en fonction du bytecode courant. En générant cette boucle, il est possible d'omettre le support pour certains bytecodes, ou de changer la sémantique de certains d'entre eux, par exemple en omettant de générer le code chargé de la gestion des exceptions. Cette génération se base sur un ensemble de macros, chacune implémentant soit une condition exceptionnelle à tester, soit la sémantique d'un bytecode (listing 4.2).

Une fois que le code de l'interpréteur natif est généré, et que tous les composants de la

Listing 4.2 – Exemples de génération personnalisée de la sémantique de certains bytecodes.

```
1  switch (bytecode) {
2  ...
3  /* le bytecode GETFIELD peut lancer une NullPointerException */
4  case GETFIELD:
5    CHECK_NULLPOINTEREXCEPTION;
6    GETFIELD_BODY;
7    break;
8  /* ce qui n'est pas le cas pour le bytecode PUTFIELD */
9  case PUTFIELD:
10   /* CHECK_NULLPOINTEREXCEPTION; */
11   PUTFIELD_BODY;
12   break;
13 ...
14 }
```

machine virtuelle sont compilés, leurs éléments inutilisés peuvent être purgés en effectuant une opération de ramasse-miettes lors de l'édition des liens.

4.3.2.3 Ramasse-miettes lors de l'édition des liens

L'équivalent pour l'environnement d'exécution natif du ramasse-miettes effectué lors de l'élimination des objets inatteignables est fait lors de l'édition des liens de l'environnement natif. En effet, il est possible avec de nombreux compilateurs C de demander à éliminer les symboles non référencés lors de la création d'un binaire statique. Avec le compilateur GCC [Stal], cette opération est effectuée en passant l'option `-ffunction-sections` lors de la compilation et `-gc-sections` lors de l'édition des liens. Ainsi, toute entité C de la machine virtuelle non-référencée dans le binaire est supprimée. On obtient ainsi une spécialisation assez fine, non basée sur des directives de compilation, et surtout dans le prolongement de la spécialisation du système Java, puisque les spécialisations effectuées sur le système Java permettent aux optimisations du compilateur d'opérer plus efficacement.

4.3.3 Spécialisations opérées par l'optimisation du code faite par le compilateur

Comme nous l'avons vu en 4.3.1.2, les références de la partie native vers la partie Java sont des références vers des champs statiques. Cette caractéristique, combinée aux analyses décrites en 4.1 et aux capacités d'optimisation du compilateur C, permettent d'opérer des spécialisations à grain plus fin dans le code.

Lorsqu'un champ statique est déterminé comme étant constant par l'analyse, le migreur peut émettre la macro de ce champ non plus comme une référence vers la zone statique de la classe correspondante, mais comme une valeur immédiate constante. Cette valeur immédiate peut alors être propagée par le compilateur pour optimiser le code natif, ce qui correspond pour nous à une spécialisation de la couche native en fonction des caractéristiques de la couche Java.

Les conséquences de cette propagation peuvent être la suppression de chemins d'exécution

dans la couche native, permettant non seulement de réduire la taille du code natif, mais également de supprimer des références à d'autres sections natives référencées dans le code mort. Ces dernières pourront être collectées par le ramasse-miettes de l'éditeur de liens.

4.4 Apports de la spécialisation tardive

La spécialisation tardive consiste à spécialiser un logiciel à un moment avancé de son cycle de vie, c'est-à-dire après son déploiement, voire pendant son exécution. Pour permettre ceci, la spécialisation intervient durant la romization, qui d'après notre définition et notre architecture du chapitre 3 permet de couvrir toutes les étapes de la vie du logiciel.

Cette intervention tardive de la spécialisation lui permet de travailler dans un contexte bien plus favorable qu'un logiciel non-déployé : c'est non seulement le logiciel, mais également toutes les bibliothèques du système qui sont spécialisées, avec un contexte de départ parfaitement connu. C'est pourquoi l'analyse peut travailler avec des valeurs inférées plus précises, comme par exemple les instances présentes dans le système. Cette phase d'analyse fait usage d'un interpréteur abstrait pour parcourir tous les chemins d'exécution possibles du système, et l'annote avec les informations qu'elle a inférées sur ses futurs états atteignables.

Prenant la suite de l'analyse, les spécialisations du système tirent parti de son résultat pour réduire le plus possible la taille de ce dernier, sans pour autant changer son comportement. Nous distinguons deux niveaux dans la spécialisation : celle du système Java, et celle de la couche native du système, servant d'interface entre le système Java et le matériel et fournissant des services natifs tels que l'interprétation des bytecodes.

La spécialisation du système Java s'effectue tout d'abord par la mise à *null* des références du système que les analyses ont déterminé comme non-atteignables. Cette seule opération a pour effet d'isoler les objets inatteignables du système, incluant les classes et méthodes jamais atteintes, qui sont alors collectés par le ramasse-miettes. Cette spécialisation opère à grain élevé (sur des objets complets) et offre le plus gros gain en termes d'empreinte mémoire. Peuvent alors intervenir des spécialisations plus spécifiques et à grain plus fin, comme la suppression des champs d'instances et statiques inutilisés, ou l'élimination du code mort au sein des méthodes.

La spécialisation de la couche native est moins évidente, celle-ci n'étant pas soumise à une analyse particulière et opérant comme une « boîte noire » pour l'interpréteur abstrait. Nous avons choisi de la spécialiser de la même manière que la partie Java, c'est-à-dire en supprimant des références entre les éléments qui la composent, ici principalement les fonctions C. Les points de départ de cette spécialisation sont les interfaces entre la partie Java et la partie native : l'interpréteur de bytecodes et les méthodes natives. Les bytecodes ainsi que les méthodes natives non-rencontrés dans le système Java sont omis dans la couche native, ce qui a pour effet potentiel de couper toute référence aux mécanismes sous-jacents de la machine virtuelle qu'ils utilisent. Ces parties jamais référencées peuvent alors être purgées par un mécanisme de ramasse-miettes inclut dans l'éditeur de liens de la chaîne de compilation. Par ailleurs, des spécialisations plus fines peuvent être réalisées de manière automatique par le compilateur C, qui peut par exemple propager la valeur d'un champ Java constant et ainsi optimiser le code généré.

Cinquième Chapitre

Résultats expérimentaux

> « Si nous ne trouvons pas des choses agréables, nous trouverons du moins des choses nouvelles. »
>
> **Voltaire**.

L'architecture de romization que nous avons décrite chapitre 3, associée aux procédés d'analyse et de spécialisation du chapitre 4, nous permettent de produire automatiquement à partir de d'image mémoire d'un système Java déployé une version spécialisée de celle-ci dont l'empreinte mémoire est fortement réduite.

Le présent chapitre vérifie expérimentalement cette revendication et son efficacité au travers de plusieurs expériences. Celles-ci ont été effectuées en utilisant l'implémentation de notre architecture de romization décrite en 3.3. L'analyse du système, ainsi que sa spécialisation, ont été effectuées à l'aide d'une implémentation des procédés expliqués dans le chapitre 4.

Nous allons dans un premier temps préciser les conditions expérimentales, les caractéristiques de notre prototype et les applications testées. Ensuite, nous évaluerons le pessimisme de l'analyse du système, selon que celui-ci soit démarré ou pas, et l'effet de la réduction du pessimisme sur la quantité de variables marquées comme utiles à l'exécution du système. Ceci nous amènera à mesurer l'impact de la spécialisation sur les données gardées dans le système, puis sur l'empreinte mémoire du système Java. Nous verrons enfin l'effet de la spécialisation effectuée sur l'environnement d'exécution embarqué.

5.1 Conditions expérimentales

Nous allons dans un premier temps décrire les conditions dans lesquelles ces expériences ont été réalisées. Elles ont été effectuées dans l'environnement JITS [JITS], et sont reproductibles en récupérant la branche *gnuroumetrics* dans le dépôt Subversion du projet.

Cette branche comprend les sources du projet JITS, les scripts permettant d'effectuer les expériences et de générer les graphes et les tableaux figurant dans ce chapitre, ainsi que les programmes ayant servi à l'évaluation et que nous allons présenter maintenant.

5.1.1 Programmes mesurés

Afin de valider notre approche, nous avons voulu tester un échantillon varié de programmes, reflétant à la fois des programmes présentant des fonctionnalités communes dans

Listing 5.1 – Code source du programme `helloworld`.

```
1  System.out.println("Hello");
2  System.out.println("World");
```

le monde de l'embarqué, et d'autres applications plus hétéroclites. L'implémentation incomplète de l'API Java par JITS[1] a sensiblement limité le choix des applications que nous pouvions tester — toutefois, celles que nous avons été en mesure d'y déployer nous ont fourni des résultats encourageants. Il est à noter que ces applications ont été écrites pour J2SE et ne sont pas initialement prévues pour être embarqués.

5.1.1.1 `helloworld`

Ce programme est l'exemple type du programme minimal, qui devrait par conséquent afficher une empreinte mémoire minimale. Il nous intéresse car du fait de sa simplicité, il permet d'observer les moindres actions de l'analyseur et du spécialiseur. Il se compose des deux lignes du listing 5.1.

5.1.1.2 `raytracer`

Ce benchmark tiré de la suite du *Java Grande Forum*, version 2, est utilisé pour mesurer les performances en calcul flottant. Il génère une image à partir d'une scène 3D créée au démarrage de l'application. Bien qu'un scénario d'utilisation de ce genre d'application dans un environnement embarqué ne soit pas crédible, il nous permet de tester le comportement de notre spécialiseur sur un programme de calcul « lourd ».

Il se déroule en deux phases de fonctionnement :

- La scène 3D est générée par le programme ;
- L'image est générée à partir de la scène 3D (phase **Render**) ;

La scène rendue se compose de 64 sphères placées uniformément, et de 5 sources de lumières. La taille de l'image rendue pour nos tests est de 5×5 pixels.

5.1.1.3 `crypt`

`crypt` est le benchmark de cryptographie de la suite du *Java Grande Forum*, version 2. Il nous permet d'adresser des programmes plus proches des préoccupations des systèmes embarqués. Le calcul de clés cryptographiques et l'échange de données cryptées sont en effet des tâches fréquentes dans un système embarqué.

Cette application travaille avec des nombres entiers et n'utilise de l'API Java que le générateur de nombres aléatoires, ce qui lui donne un plan proche d'une véritable application embarquée. Elle opère en différentes phases :

1. Les données utiles au programme sont construites, et les clés de chiffrement/déchiffrement sont calculées (phase **Init**) ;
2. Un bloc de données est chiffré à l'aide d'une fonction de chiffrement (phase **Crypt**) ;

[1] En effet, JITS ne supporte actuellement qu'un sous-ensemble de l'API de J2SE.

3. Le bloc chiffré est déchiffré en utilisant la même fonction (phase **Decrypt**) ;
4. Le résultat de l'opération de chiffrement/déchiffrement est comparé avec les données originales. Si elles ne sont pas identiques, un message d'erreur est affiché sur la sortie standard (phase **Valid**).

Pour nos tests, nous avons sélectionné une taille de données à chiffrer de 1 Ko[2].

5.1.1.4 `compress`

Ce programme est tiré de l'ensemble de benchmarks *JVMSpec98* pour J2SE et permet de mesurer les performances des opérations de compression/décompression. En utilisant l'API Java d'entrées/sorties, il lit le contenu d'un fichier dans un tampon, puis le compresse dans un autre tampon de même taille avant de le reconstituer à partir de cette forme compressée. Les différentes phases de ce programme sont donc les suivantes :

1. Allocation et remplissage du tampon contenant le fichier original (phase **Read**) ;
2. Compression du fichier depuis le tampon de lecture vers un second tampon (phase **Compress**) ;
3. Reconstitution du tampon original depuis sa forme compressée (phase **Decompress**).

Le fichier que nous lisons pour ce test a une taille de 92 Ko.

5.1.2 Caractéristiques du prototype

Pour effectuer nos tests, nous avons réalisé une implémentation de l'analyseur et des opérations de spécialisation présentées dans le chapitre 4.

5.1.2.1 L'analyseur

Notre implémentation de l'analyseur travaille sur les images mémoire produites par l'architecture de romization de JITS, représentant un système en cours d'exécution. À l'aide des procédés décrits en 4.1, il annote le système avec les informations nécessaires à sa spécialisation. Le prototype que nous avons réalisé présente toutefois certaines limitations par rapport à la spécification que nous avons donnée, notamment en ce qui concerne le suivi des valeurs. Celui-ci ne peut s'effectuer que sur les références ou les données entières présentes en pile, ou sur les données constantes du système. Dans le cas de ces dernières, le fait qu'elles soient constantes nous permet d'empiler une variable abstraite comprenant leur valeur au moment de la capture.

Afin d'obtenir le résultat de l'analyse dans un temps raisonnable, l'analyseur simplifie les données qu'il manipule s'il s'avère qu'il effectue les mêmes analyses avec des données différentes un trop grand nombre de fois. Ainsi,

– Si une même méthode est évaluée plus de 100 fois, les arguments variants entre ces évaluations sont changés afin de couvrir tout leur intervalle possible de valeurs (valeur spéciale Ω) ;

[2] Cette valeur faible permet de limiter le temps passé à chiffrer, et ainsi de faire mieux ressortir les différentes phases du programme dans nos graphes.

– Si au cours de l'analyse d'un contexte local de méthode, l'analyseur passe plus de 10 fois au même endroit de la méthode, la même simplification est effectuée sur les données variantes de la pile.

C'est une fois annoté avec les résultats de l'analyse que le système peut être spécialisé.

5.1.2.2 Spécialisations effectuées

Nous avons également implémenté un spécialiseur de système, prenant en entrée une image mémoire d'un système annotée par l'analyseur, et produisant en sortie une image mémoire du même système après qu'il ait subi les opérations suivantes :

– Élimination des objets inatteignables[3] ;
– Factorisation des chaînes de caractères identiques[4] ;
– Reconception de la structure des objets[5] ;
– Élimination du code mort[6].

Ces deux outils ont été utilisés selon le mode opératoire suivant.

5.1.3 Mode opératoire

Les critères que nous souhaitons évaluer grâce à ces expériences sont :

– L'influence du fait que le système analysé soit totalement déployé et en cours d'exécution (par opposition à pré-chargé et non-démarré) sur la précision de l'analyse ;
– L'empreinte mémoire effective qu'il est possible d'obtenir avec notre solution, par rapport à celle obtenue avec un système non-démarré.

L'évaluation de ces critères s'est faite au travers de l'exécution complète des programmes évalués au sein de l'environnement d'exécution virtuel de JITS. Au cours de leur exécution, une série de captures est réalisée (une cinquantaine par programme environ), celles-ci sont alors analysées et spécialisées et différentes mesures sont effectuées :

– Sur la qualité de l'analyse :
  – Proportion d'invocations de méthodes virtuelles et d'interface résolues ;
  – Proportion d'instructions de saut conditionnelles résolues ;
  – Quantité de variables du système déployé atteintes par l'analyse.
– Sur la qualité de la spécialisation :
  – Nombre et état des classes présentes dans le système ;
  – Nombre de méthodes présentes dans le système et quantité de code employée ;
  – Nombre d'instances restant présentes dans le système.
– Sur l'empreinte mémoire finale atteinte par le système Java spécialisé.

En guise de référence, nous avons confronté nos mesures avec celles obtenues en employant le même analyseur sur la capture au temps zéro de l'exécution du système, mais amputé de sa capacité à manipuler les instances de celui-ci : ainsi, son comportement est comparable à celui d'un analyseur travaillant sur un système pré-chargé et non-démarré[7] dans lequel

[3] Voir 4.2.1.
[4] Voir 4.2.2.
[5] Voir 4.2.3.
[6] Voir 4.2.4.
[7] Avec l'exception toutefois que certaines classes du système sont déjà initialisées, ce qui n'est normalement pas le cas dans un environnement non-démarré.

aucune instance n'est disponible pour aider l'analyse.

Nous avons également mesuré les empreintes mémoires qu'il est possible d'obtenir sur l'environnement d'exécution embarqué, selon les fonctionalités qui y sont incluses.

5.1.4 À propos des chiffres présentés

Les chiffres donnés à propos du système Java prennent en compte non seulement les applications chargées dans le système ainsi que l'API utilisée, mais aussi toute la partie du système écrite en Java. Celle-ci comprend des mécanismes internes à l'exécution du système, comme les politiques d'ordonnancement des tâches. Ceci est la conséquence des choix d'écrire autant que possible le système en Java[8], et de le spécialiser tout entier comme une entité unique, sans considérer de séparation entre couche système et couche applicative. L'API Java est en effet utilisée sans distinction par les applications et les classes du système, et la spécialisation fait que la barrière entre le niveau système est le niveau applicatif n'est pas facilement saisissable dans un système JITS déployé.

À ce titre, JITS se distingue des autres solutions Java embarquées. Les mécanismes internes au système, comme les politiques d'ordonnancement des tâches ou le chargeur de classes, sont généralement écrits en langage natif et se retrouvent ainsi dans la couche native. De même, les méta-données des classes et les méthodes sont généralement représentées dans un format natif. JITS fait au contraire le choix de représenter ces données du système par des objets Java. Ceci a pour conséquence qu'une partie non-négligeable des données habituellement présentes dans la partie native du système se retrouve dans la représentation Java.

Ces précisions faites, nous pouvons effectuer les évaluations en commençant par celles concernant le pessimisme de l'analyse.

5.2 Évaluation du pessimiste de l'analyse

Afin de garantir le parcours de tous les chemins d'exécution possibles du système, l'analyseur se doit d'être volontairement pessimiste : inclure trop de chemins d'exécution réduit la qualité de l'analyse, mais n'a pas de répercussion sur le bon fonctionnement général du système (phénomène de *sous-spécialisation*). En revanche, omettre un chemin d'exécution peut s'avérer désastreux si celui-ci est emprunté lors de l'exécution réelle du système (phénomène de *sur-spécialisation*). Une bonne analyse du système est donc celle qui inclut tous les chemins atteignables au cours de l'exécution réelle du système, tout en limitant l'inclusion de chemins non-atteignables.

Dans cette section, nous allons mesurer le pessimisme engendré par notre analyse tout au long de l'exécution du système, et le comparer avec le pessimisme engendré par l'analyse sur le système non-démarré. Dans notre architecture de romization, l'analyseur prend en entrée une image complète du système, capturée à un moment donné de son exécution. Il dispose donc de tous les éléments nécessaires pour mener l'exécution à son terme : le pessimisme est alors introduit soit par des conditions qui échappent au domaine inférable par l'analyse (par exemple, la lecture d'un octet sur une entrée), soit par des approximations de l'analyseur lui-même.

[8] Il s'agit d'un choix de conception de JITS — pour plus d'informations, voir la section 3.3.1.

Il existe de nombreux facteurs d'introduction de pessimisme qui font évaluer des chemins supplémentaires à l'analyseur. Nous allons en mesurer deux principaux, à savoir les invocations de méthodes virtuelles et d'interface, et les instructions de branchement[9]. Nous mesurerons ensuite l'effet global de ces deux mesures sur la quantité de variables estimées comme nécessaires par l'analyseur, ainsi que sur les types d'accès inférés sur ces variables.

5.2.1 Résolution des invocations de méthodes virtuelles et d'interface

L'invocation d'une méthode virtuelle ou d'interface peut engendrer plusieurs chemins d'exécution si elle n'est pas entièrement résolue. On considère qu'une invocation est entièrement résolue si les informations concernant l'instance sur laquelle la méthode est invoquée sont suffisantes pour déterminer son type avec exactitude, et par conséquent pour déterminer la méthode exacte à appeler. Le tableau 5.1 donne le pourcentage d'invocations entièrement résolues par rapport aux invocations rencontrées lors de l'analyse du système non-démarré. La figure 5.1 donne la même mesure, mais cette fois sur un système démarré, en fonction de l'avancement de son exécution.

On constate tout d'abord que l'état démarré du système réduit grandement le pessimisme lors de la résolution des invocations de méthodes. On passe ainsi à des taux de résolution proches de 100%, sauf lors de la phase de lecture de fichier de `compress`. L'incapacité à résoudre ces invocations sur un système non-démarré s'explique par le fait que même si la résolution des invocations ne nécessite qu'une connaissance du type exact, celle-ci se perd, par exemple, lorsque des accès sont faits sur des champs dont on ne connaît pas l'instance. De tels accès obligent en effet l'analyseur à se rabattre sur un type non-exact.

Il est à noter qu'en pratique, une invocation non-résolue par l'analyseur ne donne lieu à plusieurs candidats que s'il existe des sous-classes de la classe à laquelle appartient la méthode invoquée, ou d'autres classes implémentant la même interface. Une invocation non-résolue n'introduit donc pas forcément de pessimisme dans l'analyse. Toutefois, si cette assertion est souvent vérifiée dans les programmes que nous avons évalués, elle n'est en revanche pas la règle, notamment pour des systèmes plus complexes. C'est pourquoi nous considérons comme plus pertinent pour cette mesure de tenir compte de la résolution ou non de la méthode, et non pas des éventuels candidats supplémentaires introduits.

5.2.2 Résolution des instructions de branchement conditionnelles

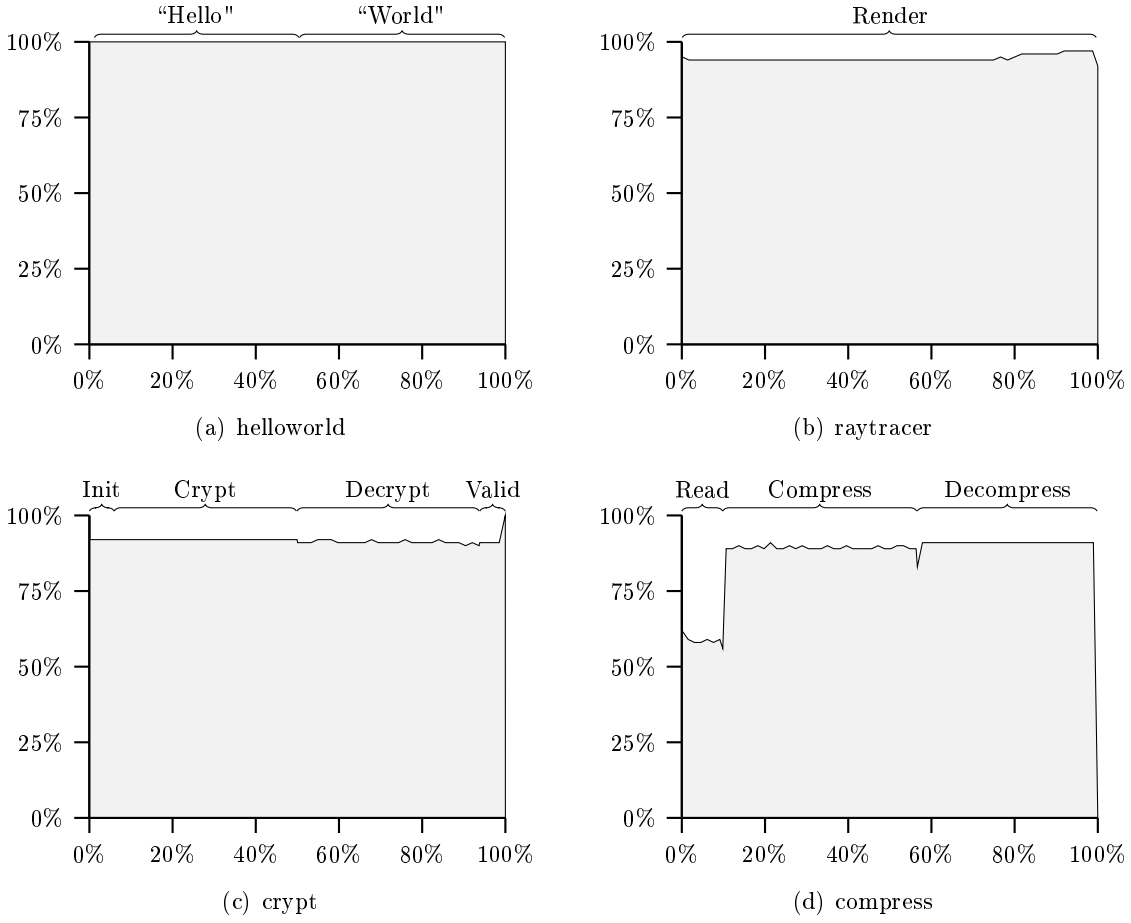
Tout comme les invocations de méthodes virtuelles et d'interface, les instructions de branchement conditionnelles peuvent introduire de nouveaux chemins à analyser et ainsi augmenter le pessimisme de l'analyse si elles ne sont pas résolues. Au contraire, un chemin unique peut être emprunté si l'on dispose de suffisamment d'information sur la ou les valeurs testées.

Par exemple, pour les bytecodes `ifnull` ou `ifnonnull`, il suffit de savoir si la référence testée peut ou au contraire ne peut pas être nulle à cet endroit du code. Les bytecodes de test numériques (`ifeq`, `iflt`, `if_icmpeq`, etc) vérifient l'égalité, l'infériorité ou la supériorité d'une donnée numérique du système sur une autre ou sur la valeur zéro. Ainsi, il n'est pas

[9] Il en existe par ailleurs bien d'autres : lancement des exceptions, accès à un index non-constant de tableau, etc.

TAB. 5.1 – Pourcentage d'invocations de méthodes virtuelles ou d'interface résolues sur le système non-démarré.

Programme	% résolues
helloworld	81 %
raytracer	80 %
crypt	72 %
compress	59 %



(a) helloworld (b) raytracer

(c) crypt (d) compress

FIG. 5.1 – Pourcentage d'invocations de méthodes virtuelles ou d'interfaces résolues lors de l'analyse, en fonction de l'avancement de l'exécution du système.

toujours nécessaire de connaître les valeurs exactes des données testées pour déterminer le chemin à emprunter par l'analyseur.

L'évaluation de la résolution des instructions de branchement conditionnelles a été faite de la manière suivante : à chaque instruction de branchement rencontrée par l'analyseur, les valeurs possibles des données comparées sont confrontées afin de déterminer si le branchement peut conduire vers un ou plusieurs chemins dans l'interpréteur abstrait. Nous considérons qu'une instruction de branchement conditionnelle est résolue si elle mène vers un seul chemin, et non-résolue si elle mène vers plusieurs. Le pourcentage d'instructions de branchement conditionnelles résolues lors de l'analyse du système non-démarré est donné par le tableau 5.2. La même mesure obtenue sur le système démarré est donnée figure 5.2.

On constate ici que l'état démarré du système n'a que peu d'influence en général sur la réduction du pessimisme, sauf pour un programme très simple comme `helloworld`. En effet, sur des programmes plus complexes, les limites fixées par l'analyseur sont rapidement atteintes, ce qui a pour effet de donner la valeur universelle Ω aux données abstraites considérées. Dans ces conditions, la connaissance des instances du système n'apporte que peu de précision en plus.

Un fait intéressant est toutefois à mentionner : sur `helloworld`, 100% des invocations de méthodes et 100% des instructions de branchement conditionnelles ont pu être résolues avec le système démarré, ce qui signifie que l'analyseur a été capable d'inférer l'unique chemin d'exécution possible pour ce programme. Cette propriété n'est pas remplie pour les autres programmes, même après que ceux-ci aient atteint un état déterministe (par exemple, `compress` est parfaitement déterministe après la lecture du fichier), en raison des bornes fixées sur l'analyseur pour limiter son temps d'exécution.

La réduction du pessimisme lors de l'analyse a pour conséquence directe une réduction du pessimisme quant aux accès effectués sur les variables du système, que nous allons à présent évaluer.

5.2.3 Mesure des variables atteintes et finalisées en lecture

Nous reprenons la définition de la variable donnée en 4.1.1, à savoir qu'une variable est un emplacement mémoire dans lequel peut être stockée une donnée atomique de type primitif ou référence. Ainsi, les champs d'instances, les champs statiques et les éléments de tableaux sont des variables du système.

La base de notre spécialisation consiste à déterminer quelles variables sont nécessaires au bon fonctionnement du système. Nous pouvons également aller plus loin en distinguant, pour ces variables, les types d'accès qui y sont effectués (lecture ou écriture). Cette information est particulièrement pertinente avec notre implémentation de l'analyseur, car celle-ci ne peut connaître la valeur d'une variable que dans le cas où elle est strictement constante[10].

La figure 5.3 donne, en fonction de l'avancement de l'exécution du système, le nombre de variables que l'analyseur a déterminé comme étant nécessaires au fonctionnement du système spécialisé, ainsi que le type d'accès qu'elles subissent : lecture/écriture (■), lecture seulement (■) ou aucun accès (□). Le tableau 5.3 donne la même mesure réalisée sur le

[10] Un autre intérêt à connaître les variables accédées en lecture seulement est que les données accessibles uniquement au travers de variables de ce type peuvent être placées en mémoire à lecture seule sur l'équipement cible.

TAB. 5.2 – Pourcentage d'instructions de saut conditionnelles résolues sur le système non-démarré.

Programme	% résolues
helloworld	28 %
raytracer	12 %
crypt	31 %
compress	24 %

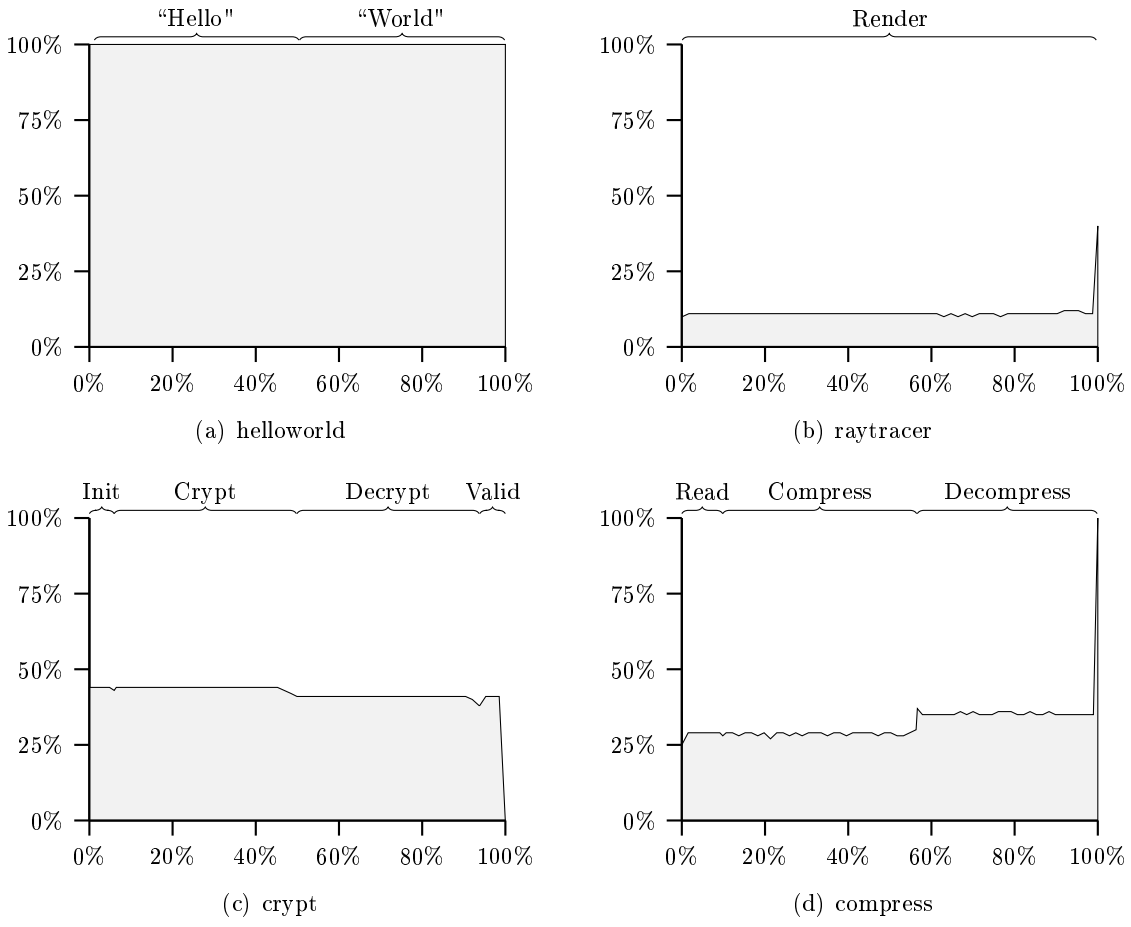


FIG. 5.2 – Pourcentage d'instructions de saut conditionnelles résolues lors de l'analyse, en fonction de l'avancement de l'exécution du système.

système non-démarré.

Il est légitime de se demander pourquoi l'analyse juge que des variables qui ne sont jamais accédées sont nécessaires au fonctionnement du système. En fait ces variables, bien que jamais référencées au cours de l'exécution, sont nécessaires car elles font partie d'une instance dont une autre variable est référencée et qui doit donc être inclue dans le système final. Par exemple, la classe `java.lang.Class` comporte un champ privé nommé `classloader` référençant le chargeur de la classe. Celui-ci n'est accédé qu'en cas de chargement d'une autre classe, ou d'appel à la méthode `getClassLoader()` sur cette instance. Si aucune de ces conditions n'est rencontrée durant l'analyse, le champ `classloader` ne subit donc aucun accès, mais reste toutefois présent car il appartient à une instance utile au système. Le même cas de figure se présente avec les tableaux dont tous les indexes ne sont pas lus (c'est notamment le cas pour le bytecode des méthodes comportant du code jamais atteint par l'analyse).

Il est cependant possible d'éliminer une partie de ces variables inutilisées grâce à deux des spécialisations que nous avons implémentées. Si un champ d'instance n'est jamais accédé pour toutes les instances présentes dans le système d'une même classe, celui-ci peut être supprimé[11]. De même, le code mort présent dans les méthodes peut être éliminé de manière sûre[12]. Après cette élimination, la quantité de variables non-utilisées restant malgré tout nécessaire (car s'agissant de champs non-éliminables ou d'entrées de tableaux non-redimensionnables) est délimitée par ⬚.

Nous constatons alors sans grande surprise que l'analyseur a été en mesure de déterminer que `helloworld` n'écrit dans aucune variable située hors de la pile[13]. Le chemin exact ayant été inféré par l'analyseur permet de n'inclure que les données strictement nécessaires à son exécution. Il ressort également que la moitié des variables faisant partie des objets inclus dans le système ne sont pas nécessaires à son fonctionnement. Cependant, les mécanismes de reconception des objets et d'élimination du code mort permettent de grandement réduire cette quantité. Le même phénomène est observable sur les autres programmes évalués pendant leur exécution.

Les programmes fonctionnant en plusieurs phases (`crypt`, `compress`) voient ainsi ces dernières révélées par l'analyse des accès faits à leurs variables. On observe ainsi dans `crypt` que beaucoup de variables ne sont plus écrites après la phase de chiffrement. Le même phénomène est à nouveau observable avec la phase de validation, qui ne fait que comparer deux tableaux de données.

Les phases de `compress` apparaissent également très clairement. La lecture du fichier se traduit par une importante allocation et une écriture de toutes les variables ainsi allouées. À la compression et la décompression correspondent de nouvelles allocations, tandis que les données lues à partir du fichier ne sont alors plus modifiées.

Enfin, la différence de variables marquées entre analyse sur système démarré et non-démarré est assez flagrante, surtout en ce qui concerne les variables accédées en écriture. Les chemins supplémentaires évalués par l'interpréteur le font ainsi passer par des cas où des exceptions peuvent être construites et lancées, ce qui augmente très considérablement la

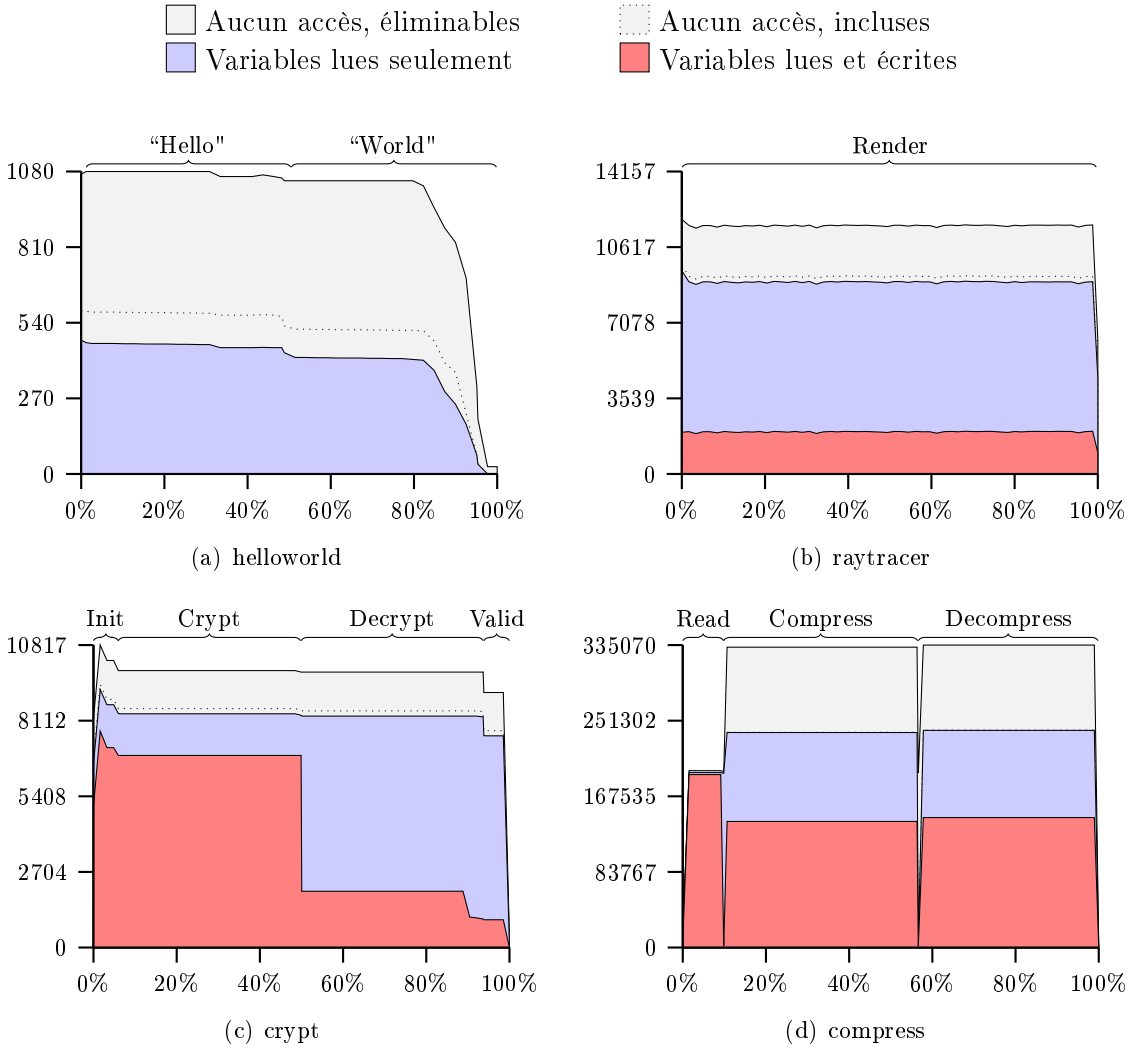
[11] Voir section 4.2.3.

[12] Voir section 4.2.4.

[13] ... à l'exception d'une seule variable, non-visible sur le graphe : celle correspondant au bytecode `getstatic` de la méthode principale du programme, changé en `getstatic_quick` dès les premiers bytecodes exécutés. À noter que cette modification aurait pu être effectuée avant l'exécution du système.

TAB. 5.3 – Quantité de variables présentes dans le système spécialisé non-démarré en fonction de leur type d'accès déterminé par l'analyse.

Programme	Aucun accès		Lues seulement	Lues, écrites	Total
	Éliminables	Incluses			
helloworld	1371	193	1026	3767	6357
raytracer	2211	553	1954	10187	14905
crypt	1497	213	1270	6075	9055
compress	1947	472	2137	10017	14573



(a) helloworld (b) raytracer

(c) crypt (d) compress

FIG. 5.3 – Quantité de variables présentes dans le système spécialisé déterminées comme lues et écrites, lues seulement et jamais accédées en fonction de l'état d'avancement du système (hors piles d'exécution).

quantité de données pouvant être écrites. En effet, pour créer une exception, l'interpréteur passe par la création d'une chaîne de caractères ce qui entraîne l'analyseur à considérer que les chaînes présentes dans le système peuvent être écrites, ce qui n'arrive pas.

Nous allons à présent mesurer l'impact que cette élimination des variables a sur les données présentes dans le système spécialisé.

5.3 Élimination et spécialisation des données du système

Éliminer des variables permet de réduire le nombre de références pointant sur les instances présentes dans le système[14]. Ce faisant, certaines d'entre elles deviennent totalement isolées et peuvent alors être définitivement supprimées. Dans cette section, nous allons mesurer l'évolution du nombre et de l'état de certains types de données significatifs du système.

5.3.1 Nombre et état des classes présentes dans le système

Comme nous l'avons vu, une classe Java passe par plusieurs états avant d'être utilisable par le système[15] : elles est tout d'abord chargée (état `LOADED`), puis ses liens avec les classes externes du système sont résolus (état `LINKED`), et enfin, son initialiseur est exécuté lors de son premier usage actif (état `READY`). Elle devient alors prête à l'utilisation.

Le chargement d'une classe jusqu'à l'état `LINKED` n'est pas contraint temporellement et son temps d'occurrence est laissé au choix du concepteur du système — ainsi, dans le monde de l'embarqué, est-elle réalisée de manière précoce de telle sorte à ne pas perturber l'exécution du système. En revanche, l'initialisation de la classe doit impérativement se produire lors du premier usage actif de cette dernière.

Nous nous intéressons ici à la réduction du nombre de classes présentes dans le système obtenue avec la spécialisation. Le tableau 5.4 donne le nombre de classes chargées (état `LINKED`) et prêtes (état `READY`) initialement présentes dans le système non-démarré[16], puis le nombre restant après spécialisation de celui-ci.

Nous pouvons alors comparer ces chiffres avec l'évolution du nombre et de l'état des classes montré sur la figure 5.4, qui donne ces chiffres mesurés sur le système démarré, et en fonction de l'avancement de son exécution.

On constate en premier lieu qu'un grand nombre de classes présentes dans le système non-spécialisé sont éliminées dès la spécialisation du système non-démarré. Ces classes font soit partie des mécanismes de chargement de classes devenus inutiles à l'exécution du système, soit des APIs référencées par une autre classe mais jamais utilisées. On remarque également que le nombre de classes présentes dans le système spécialisé non-démarré et dans le système spécialisé au début de son exécution sont pratiquement les mêmes, à l'exception de `helloworld`. Pour ce dernier, les classes supplémentaires sont principalement induites par les exceptions que l'analyseur a considérées comme jettables sur le système non-démarré.

On constate ensuite qu'au cours de l'exécution du système, certaines classes deviennent inutilisées et peuvent être supprimées. Cela se produit surtout après que des fonctionalités

[14] Voir 4.2.1

[15] Voir 2.3.2.

[16] En réalité, si le système n'était vraiment pas démarré, le nombre de classes prêtes serait bien moins élevé — il ne comprendrait que les classes de types primitifs et tableaux.

TAB. 5.4 – Nombre de classes chargées et prêtes sur le système non-démarré, avant et après sa spécialisation.

Programme	Avant spécialisation		Après spécialisation	
	Chargées	Prêtes	Chargées	Prêtes
helloworld	214	213	36	36
raytracer	227	219	57	50
crypt	215	214	39	39
compress	225	213	60	49

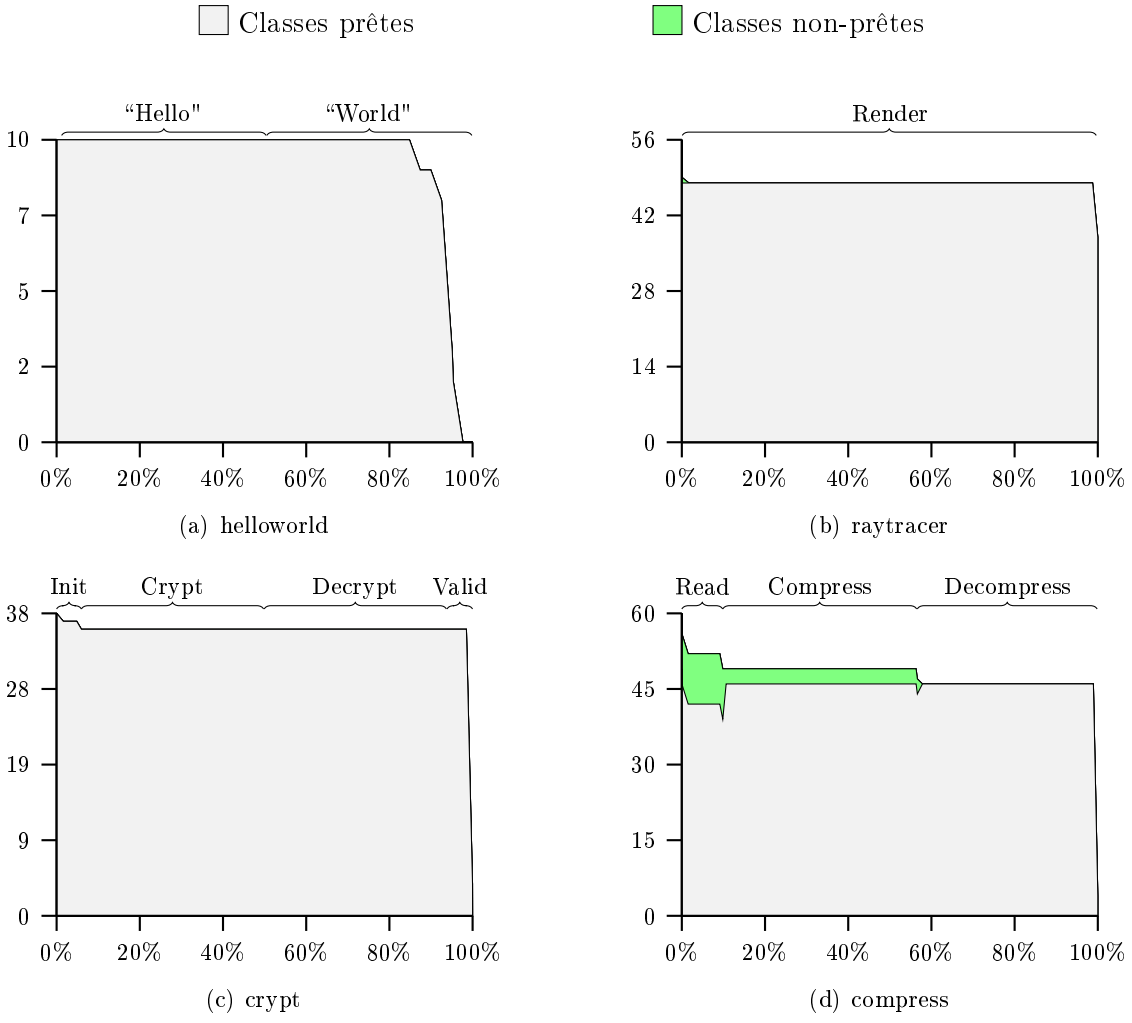


FIG. 5.4 – Nombre et état des classes présentes dans le système, en fonction de l'avancement de son exécution au moment de sa capture.

du programme ne sont plus utilisées : c'est notamment le cas après les phases d'initialisation de `raytracer`, `crypt` et `compress`. `compress` perd encore quelques classes après la phase de compression, car les fonctionalités de compression et de décompression sont implémentées dans des classes différentes. On constate enfin que les classes chargées de la compression et de la décompression ne passent à l'état `READY` qu'au moment de l'entrée dans la phase correspondante, ce qui nécessite d'embarquer les méthodes d'initialisation de ces classes jusque là.

À un niveau plus fin, nous pouvons également mesurer l'impact de la spécialisation sur le nombre de méthodes présentes et le volume que le code occupe dans le système.

5.3.2 Volume de code présent dans le système

La réduction du nombre de chemins évalués par l'analyseur réduit la quantité de code présente dans le système final à travers deux facteurs :

1. Au niveau inter-procédural, de par la non-invocation de certaines méthodes ;
2. Au niveau intra-procédural, de par la non-interprétation de certains des bytecodes d'une méthode par l'interpréteur abstrait, ceux-ci ne se trouvant pas dans un chemin d'exécution empruntable.

Ces deux facteurs se traduisent par deux types de spécialisation du système : la suppression des méthodes non-référencées lors de l'élimination des objets inatteignables, et la spécialisation des méthodes restantes qui sont amputées de leur code mort.

5.3.2.1 Élimination des méthodes non-invoquées

Commençons par mesurer l'effet de la spécialisation sur le système non-démarré. Le tableau 5.5 indique le nombre de méthodes présentes dans le système non-démarré avant et après spécialisation.

Ces chiffres sont à mettre en contraste avec la figure 5.5, qui nous donne le nombre de méthodes présentes dans le système en fonction de l'état d'avancement de son exécution. Parmis celles-ci, nous distinguons les méthodes implémentant réellement une fonctionalité (■) des méthodes abstraites ou d'interface (□).

On observe que grâce à l'élimination des objets non-atteignables, toutes les méthodes non-parcourues par l'interpréteur abstrait sont absentes du système final. Cela est valable non seulement pour les méthodes de l'API qui ne sont jamais appelées par les programmes déployés, mais également pour les méthodes appelées précédemment dans l'exécution et qui ne sont plus appelées ultérieurement. Ce dernier effet s'observe particulièrement dans les programmes fonctionnant en plusieurs phases, comme `crypt` ou `compress` : une fois qu'une phase du programme est terminée, les méthodes utilisées exclusivement par cette phase sont purgées du système spécialisé. Cela signifie que notre solution est en mesure d'isoler certaines fonctionalités intéressantes d'un programme à partir du moment où il est possible de déterminer que celles-ci ne sont plus appelables après un certain degré d'exécution. Ainsi dans `compress`, dont le mode opératoire est purement séquentiel, le code chargé de la compression d'un bloc de données est éliminé du système une fois la phase de compression terminée.

Nous observons également que le code chargé de l'initialisation d'un programme est éliminé une fois cette phase terminée. Ce phénomène est particulièrement intéressant dans les

TAB. 5.5 – Nombre de méthodes présentes dans le système non-démarré avant et après sa spécialisation.

Programme	Avant spécialisation	Après spécialisation
helloworld	1839	62
raytracer	1908	157
crypt	1854	80
compress	1879	138

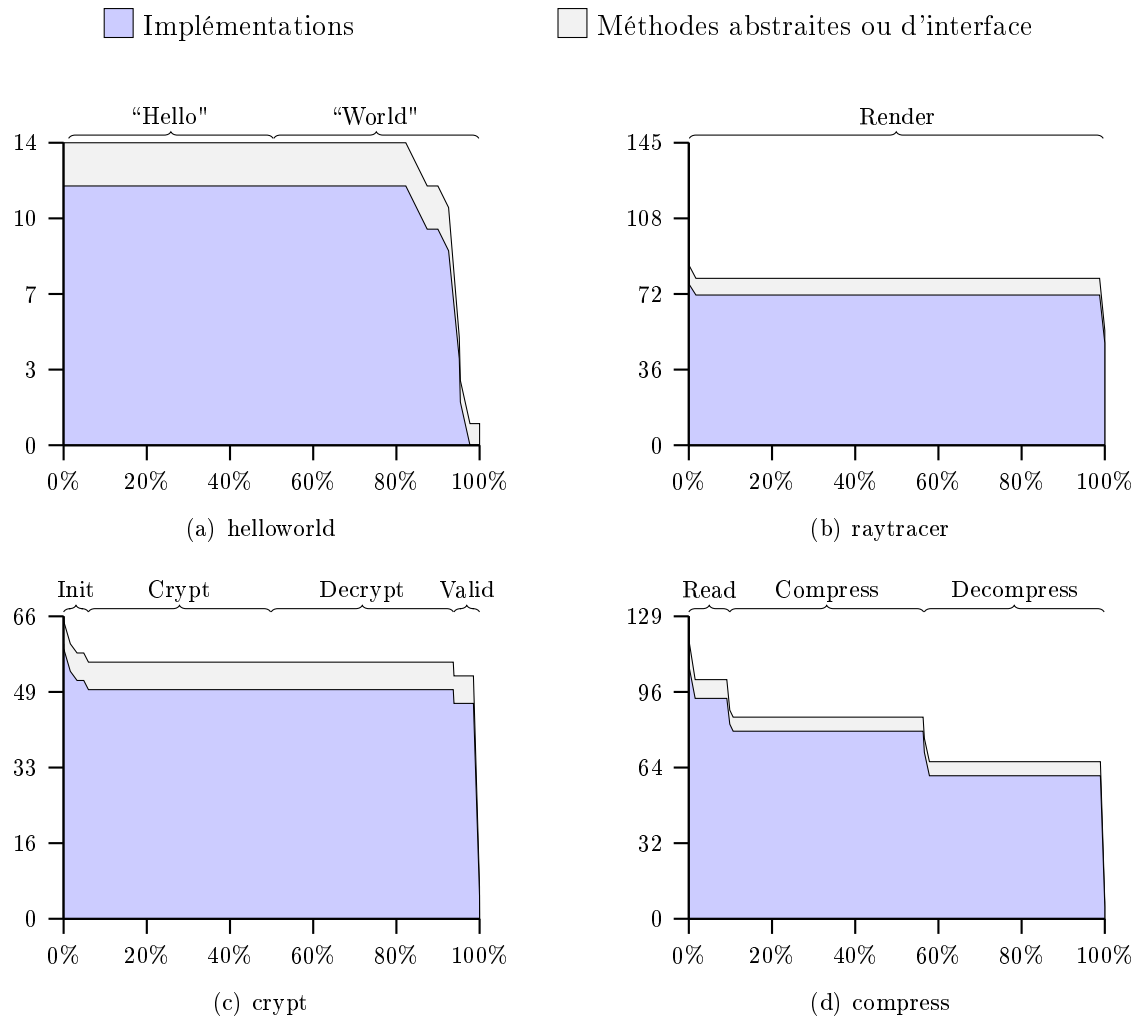


FIG. 5.5 – Nombre de méthodes présentes et utilisées dans le système spécialisé, en fonction de son état d'avancement au moment de sa capture.

schémas de déploiement embarqués pour lesquels l'initialisation de l'équipement se fait chez le fabricant, puisqu'elle permet de récupérer l'espace occupé par le code d'initialisation du système, tout en permettant à ce dernier d'assumer lui-même cette phase. Nous pouvons mettre ce fait en perspective avec notre exemple de la `PhotoCard` émise avec un ensemble de photos pré-chargées en mémoire morte[17]. En effet, les photos présentes en mémoire morte sont chargées par le fabricant de la carte avant l'émission de celle-ci. Cette phase se déroule donc dans l'environnement d'exécution virtuel de notre architecture de romization. Une fois que l'ensemble de photos désiré est chargé, la carte peut être « fermée » par la mise à *vrai* d'un booléen interdisant le chargement de nouvelles photos. De cette manière, l'analyseur considère que le chargement de photos n'est pas un chemin empruntable, et que par conséquent les photos ne sont accédées qu'en lecture seule : le système spécialisé est alors expurgé de la fonctionalité de chargement indésirée, et le placeur mémoire peut décider de placer les photos dans une mémoire non-réinscriptible.

Enfin, la résolution statique des invocations de méthodes virtuelles et d'interface est également un facteur d'éliminations de méthodes. Les différences observables dans le taux de résolution entre le tableau 5.1 et la figure 5.1 sont également visibles dans le tableau 5.5 et la figure 5.5.

5.3.2.2 Élimination du code mort

La figure 5.6 nous donne la taille en octets du code présent dans le système final. Nous distinguons le code présent avec élimination du code mort (■) et sans celle-ci (□). Ces résultats sont à comparer avec le tableau 5.6, qui présente les mêmes mesures sur le système non-démarré.

On constate sans surprise que le plus gros nombre de méthodes présentes dans le système non-démarré conduit à un volume de code plus important. La mesure la plus intéressante est celle effectuée sur le système en cours d'exécution.

Ainsi, outre le fait que la quantité de code diminue sensiblement avec le nombre de méthodes (on constate ainsi que la phase de lecture de `compress` occupe plus de 2 Ko de code), on peut constater qu'il reste une quantité non-négligeable de code inaccessible au sein même des méthodes restantes. En effet, certaines méthodes de l'API sont appelées avec des paramètres constants, et une partie de leur code dépendant de ces paramètres ne peut donc pas être atteint durant l'exécution du système. Le même phénomène peut être observé avec les méthodes d'une application, ses paramètres d'exécution devenant de plus en plus constants au fur et à mesure de son exécution.

À titre d'exemple, observons le code de la méthode `String.charAt()`, qui retourne le caractère de la chaîne dont l'index est fourni en argument (listing 5.2). Celui-ci comporte une partie totalement inutilisée dans tous les programmes que nous avons évalués et qui font usage des chaînes de caractères.

Dans JITS, les chaînes de caractères sont codées en utilisant le codage UTF-8 [Yerg 98], ceci afin d'économiser l'espace qu'elles occupent. UTF-8 code en effet les caractères occidentaux non-accentués selon leur code ASCII, de telle sorte que ceux-ci n'occupent qu'un octet dans une chaîne de caractère. Les autres caractères occupent 2, 3 ou 4 octets en fonction de leur fréquence. Une chaîne de caractère encodée en UTF-8 ne comportant que des caractères

[17] Voir 3.1.2.2.

TAB. 5.6 – Quantité de code (en octets) présente dans le système non-démarré avant et après sa spécialisation.

Programme	Avant spécialisation	Après spécialisation	
		Code présent	Code éliminé
helloworld	109447	2990	132
raytracer	114182	8533	131
crypt	111475	5029	132
compress	113021	8116	132

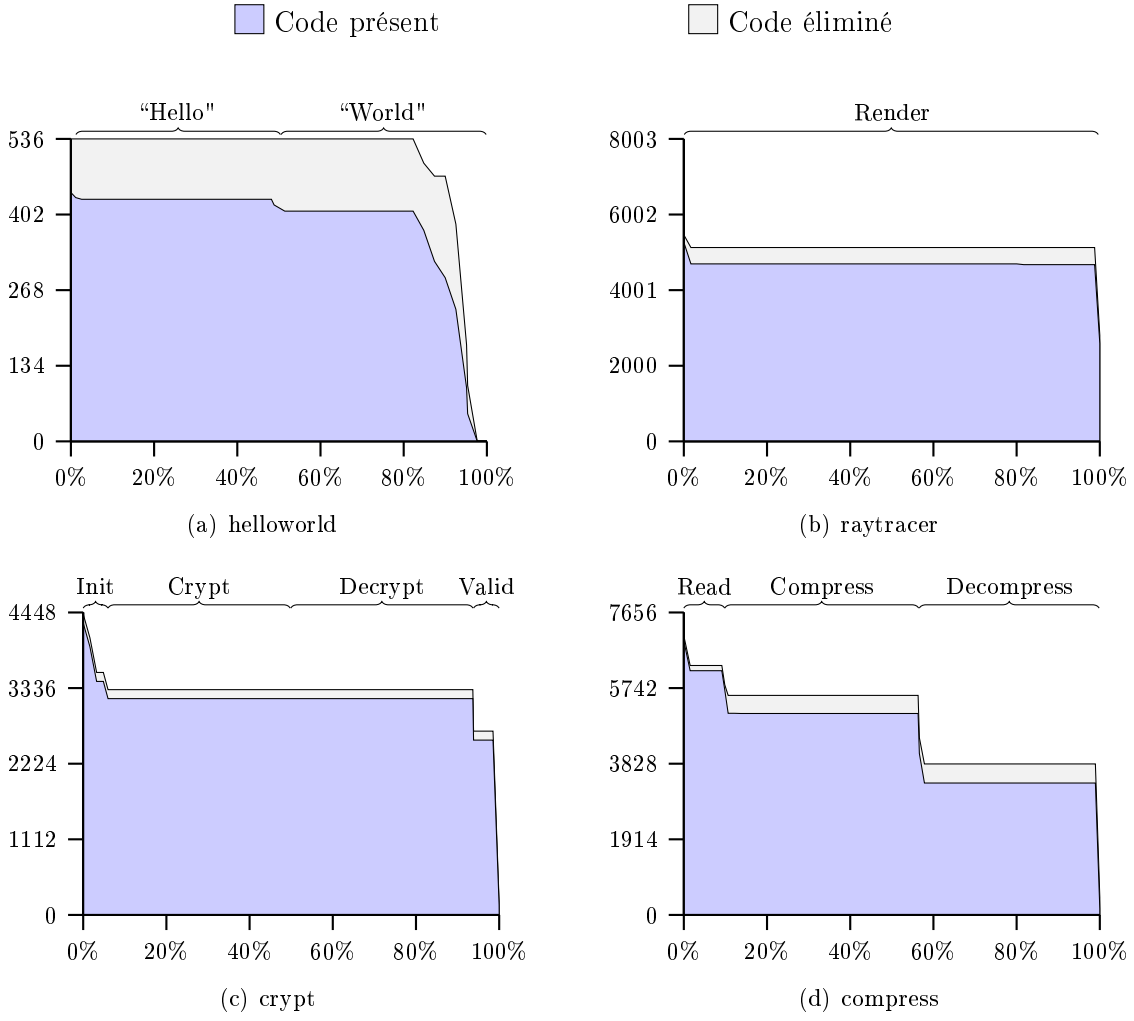


FIG. 5.6 – Quantité de code (en octets) présente dans le système spécialisé en fonction de l'état d'avancement de son exécution au moment de sa capture.

Listing 5.2 – La méthode `String.charAt`.

```
1  public char charAt(int index) {
2    if (index < 0 || index >= count)
3      throw new StringIndexOutOfBoundsException(index);
4
5    if (isASCII)
6      return (char) this.value[this.start + index];
7    else
8      return codec.decodeUTF8ToChar(this.value,
9        codec.getIndexCharAt(this.value, this.start, index));
10 }
```

occidentaux non-accentués, telle qu'une phrase en anglais, peut donc se lire strictement de la même manière qu'une chaîne ASCII. Dans l'implémentation de JITS de la classe `String`, le booléen `isASCII` permet de savoir si l'on a affaire à une telle chaîne. Si c'est le cas, la position d'un caractère dans le tableau d'octets de la chaîne est égale à l'index donné en argument ; sinon il est nécessaire de parcourir les caractères un à un jusqu'à arriver au caractère désiré. C'est le rôle de la méthode `getIndexCharAt`, ligne 9.

Dans les lignes 5 à 9, la méthode teste la nature de la chaîne de caractère afin de savoir quelle démarche adopter. Or, dans tous les programmes testés, les chaînes de caractères, qu'elles soient constantes ou construites, sont toutes des chaînes ASCII. L'analyseur est donc en mesure de déduire que la seconde partie de la condition n'est pas accessible, et de l'éliminer du système spécialisé.

Pour compléter cette étude sur les données incluses dans le système spécialisé, nous allons mesurer le nombre d'instances présentes, sans considération pour leur type.

5.3.3 Nombre d'instances présentes dans le système

Le nombre total d'instances détectées comme nécessaires dans le système non-démarré, avant et après sa spécialisation, est donné par le tableau 5.7. Le nombre d'instances détectées comme nécessaires après analyse du système démarré est quant à lui donné figure 5.7. Dans les deux cas, nous distinguons les objets uniquement lus et pouvant être placés en mémoire morte des objets lus et écrits nécessitant d'être placés en mémoire réinscriptible.

On constate que le nombre d'instances présentes décroît logiquement avec l'exécution du système (à l'exception des objets alloués par le système lui-même), en partie à cause des méthodes et classes qui ne sont plus utilisées. Mais le phénomène s'étend aux autres objets du système : ainsi, `helloworld` utilise toutes les méthodes présentes initialement jusqu'à la fin de son exécution, pourtant on peut observer que certains objets disparaissent après l'affichage de la chaîne "Hello" : précisément, les objets ayant servi à représenter cette chaîne, qui ne sont plus utilisés.

Les chiffres donnés sur le nombre d'instances présentes dans `raytracer` peuvent surprendre : il semble en effet en regardant le graphe qu'il y a moins d'instances marqués comme nécessaires dans le système non-démarré que dans le système démarré. En réalité, le nombre d'instances présentes à l'instant zéro de l'exécution de `raytracer` est égal à 743. Cependant, la phase d'initialisation alloue de nombreuses autres instances pour créer la scène,

TAB. 5.7 – Nombre d'instances présentes dans le système non-démarré avant et après sa spécialisation.

Programme	Avant spécialisation	Après spécialisation	
		Lues seulement	Lues et écrites
helloworld	12024	170	198
raytracer	12537	245	533
crypt	12146	188	273
compress	12430	310	437

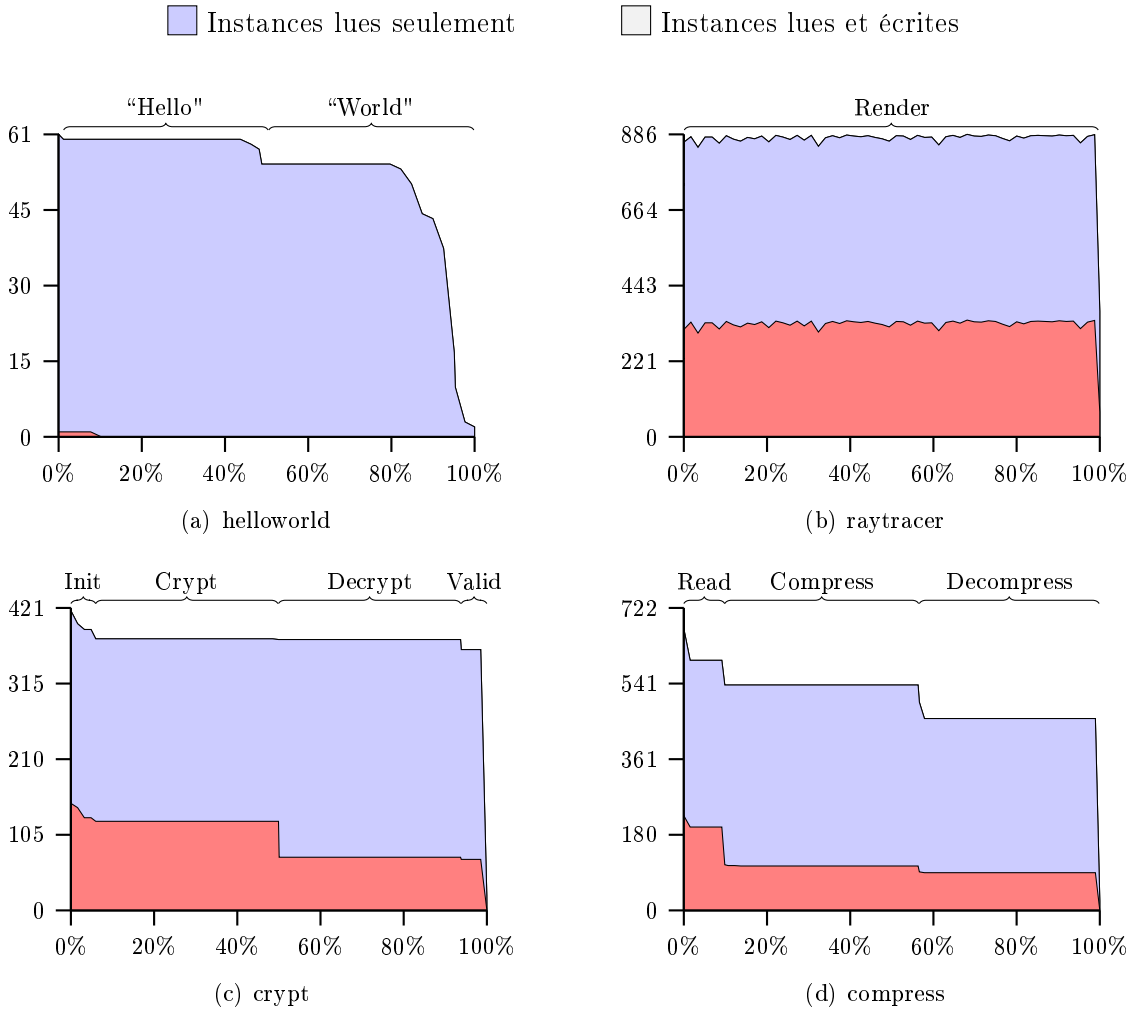


FIG. 5.7 – Nombre d'instances présentes dans le système spécialisé en fonction de son degré d'avancement au moment de sa capture.

et celle-ci est trop courte pour apparaître sur le graphe.

Le fait de supprimer des instances dans le système a un impact direct sur l'empreinte mémoire de celui-ci, ce qui est précisément la finalité de la spécialisation.

5.4 Impact sur l'empreinte mémoire du système

La réduction du pessimisme de l'analyse, du nombre de classes et de méthodes présentes, de la quantité de code dans chaque méthode et du nombre d'instances en général permettent de réduire la taille du système Java spécialisé dans son ensemble. Nous allons maintenant mesurer l'effet global de nos spécialisations sur la taille finale du système.

5.4.1 Empreinte mémoire du système non-spécialisé

Cette mesure sert de mesure témoin, afin d'évaluer le gain réellement obtenu par la spécialisation. Le tableau 5.8 donne les tailles initiales du système non-démarré, avant et après sa spécialisation. La figure 5.8 donne la taille du système non-spécialisé, en fonction de l'avancement de son exécution.

Le système non-spécialisé comprend tout le mécanisme de chargement de classes, utilisé pour charger les classes du système ainsi que celles des applications, et toutes les dépendances nécessaires à son fonctionnement : mécanismes d'entrées/sorties, exceptions pouvant être lancées, etc. Les mesures données correspondent à la taille du tas d'objets sur lequel a été effectué une opération de ramasse-miettes afin de ne garder que les objets effectivement référencés.

Cette mesure permet de voir que la taille du système non-spécialisé n'évolue pas au cours de son exécution, sauf allocation mémoire effectuée par l'application elle-même. Ainsi, `compress` alloue environ 500 Ko de données pour effectuer l'opération de compression, et environ 200 Ko pour celle de décompression (figure 5.8(d)). Une autre observation importante est à faire ici : un système déployé faisant tourner une application aussi minimaliste que `helloworld` référence tout de même plus de 379 Ko d'objets tout au long de son exécution.

Cela signifie que tous les objets référencés au démarrage du système le sont encore à la fin de l'exécution de celui-ci, quand bien même ils ne sont d'aucune utilité pour accomplir sa tâche. Par la spécialisation, nous sommes en mesure de couper les références non-exploitées par le système pour le réduire aux tailles données ci-après.

5.4.2 Empreinte mémoire du système spécialisé

La figure 5.9 donne les empreintes mémoires du système Java obtenues pour nos différents programmes de test, en fonction de l'état d'avancement de celui-ci.

La réduction par rapport au système non-spécialisé est flagrante, que l'analyse ait été effectuée sur un système démarré ou non. Mais on observe également un fort contraste entre l'empreinte mémoire obtenue par analyse d'un système non-démarré et celle obtenue par analyse d'un système démarré. Le seul programme affichant une faible différence entre les deux mesures est `raytracer`. L'état démarré du système n'a pas eu beaucoup d'influence sur le résultat de l'analyse, probablement parce que son déroulement est extrêmement linéaire et suffisamment prévisible sans disposer de beaucoup d'information de contexte.

TAB. 5.8 – Taille (en octets) du système non-démarré avant et après sa spécialisation.

Programme	Avant spécialisation	Après spécialisation	
		Lu seulement	Lu et écrit
helloworld	379192	8745	5485
raytracer	395385	11494	18941
crypt	383776	9750	8656
compress	391813	14745	13952

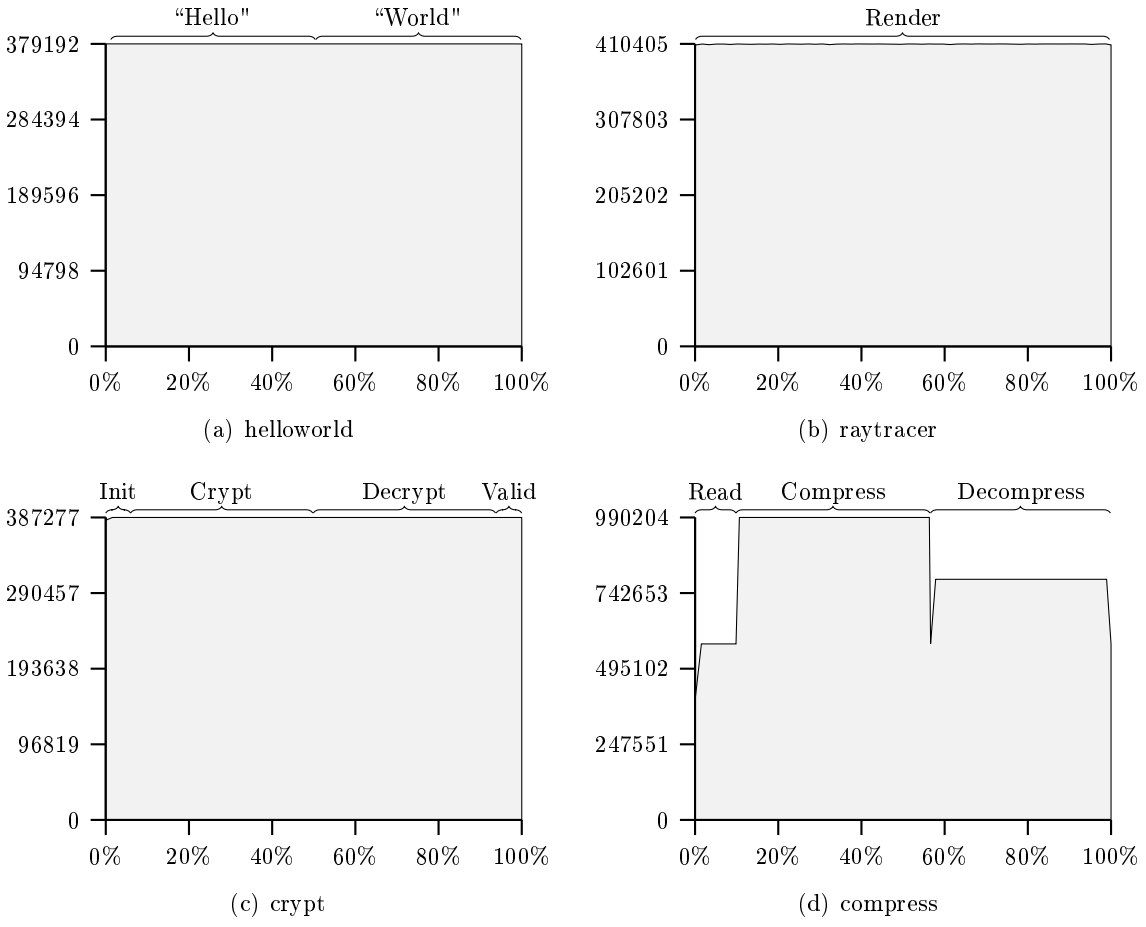


FIG. 5.8 – Taille (en octets) du système non-spécialisé, en fonction de l'avancement de son exécution.

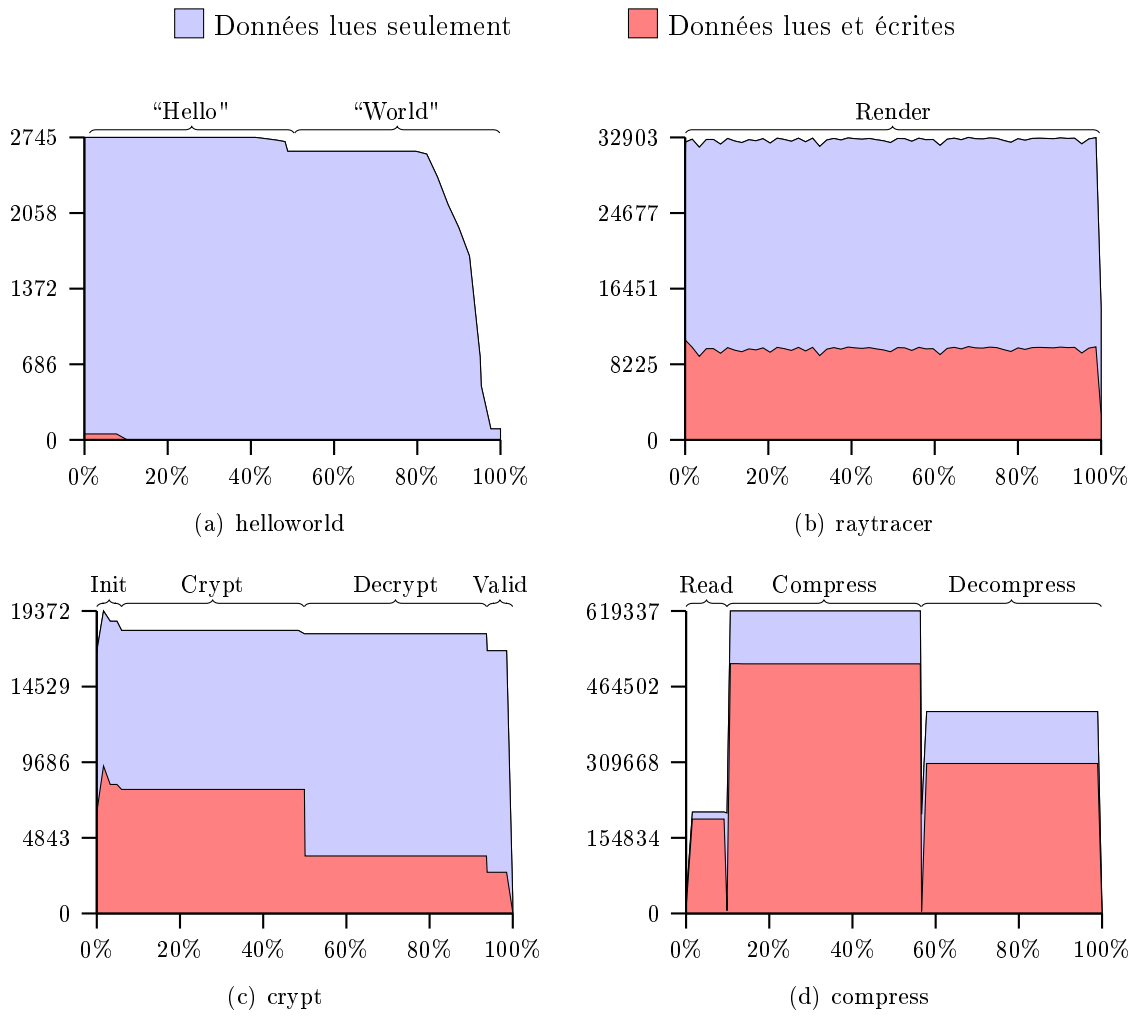


FIG. 5.9 – Taille (en octets) du système spécialisé en fonction du degré d'avancement de son exécution au moment de sa capture.

La partie Java du système ayant été évaluée, nous pouvons maintenant nous intéresser à l'évaluation de la spécialisation de la partie native.

5.5 Réduction de l'empreinte de l'environnement d'exécution embarqué

Les sections précédentes ont permis de mesurer la réduction de l'empreinte mémoire du système Java. Or, celui-ci repose sur un environnement d'exécution lui servant d'interface avec matériel sous-jacent, et fournissant la machine virtuelle dans laquelle il s'exécute. Dans la section 4.3, nous avions donné la procédure employée pour sa spécialisation : d'une part, l'analyseur du système maintient une liste des bytecodes rencontrés pour chaque application, ce qui lui permet de générer le code de l'interpréteur de bytecode embarqué ne supportant que ceux-ci. D'autre part, il observe si certaines propriétés du système sont vérifiables pour paramétrer la compilation de l'environnement. Dans cette section, nous allons tout d'abord

mesurer les fonctionalités de l'environnement d'exécution à inclure en fonction des différents programmes, puis les empreintes mémoire obtenues.

5.5.1 Mesure du nombre de bytecodes utiles

La partie essentielle de la spécialisation de l'environnement d'exécution embarqué consiste en la suppression des liens qui existent entre l'environnement Java et l'environnement natif. Ces liens sont essentiellement créés par les bytecodes présents dans les chemins d'exécution considérés par l'analyseur, qui nécessitent d'inclure leur support dans la machine virtuelle native. Le support pour les bytecodes non-présents, en revanche, peut être omis, ce qui permet de couper des références entre l'interpréteur de bytecodes natif et les autres fonctionalités de l'environnement d'exécution.

Notre prototype d'analyseur est en mesure de recenser les bytecodes nécessaires au fonctionnement du système, et de détecter si l'ordonnanceur de tâches peut fonctionner en mode monotâche. Ce dernier point n'est possible que si une seule tâche est présente tout au long de l'exécution du système. Comme c'est le cas pour chacune des applications que nous avons testées, ce paramètre est toujours vrai dans nos tests.

La mesure qui nous intéresse plus particulièrement est le nombre de bytecodes employés par chaque programme. Sachant que le jeu d'instructions de JITS comprend 242 bytecodes différents[18], le tableau 5.9 donne la mesure du nombre de bytecodes considérés comme effectivement nécessaires à l'exécution du système, alors que celui-ci n'est pas encore démarré. La figure 5.10 donne la même mesure, cette fois au cours de l'exécution du système.

Sans surprise, les chemins d'exécution supplémentaires introduits par l'état non-démarré du système ajoutent quelques bytecodes supplémentaires à supporter. On observe, également sans surprise, que le nombre de bytecodes utilisés décroît avec l'avancement de l'exécution du système, au fur et à mesure que ses fonctionalités deviennent inutiles. Dans `helloworld`, le bytecode `getstatic_quick` n'est ainsi plus utilisé dès le début de l'affichage de la seconde chaîne de caractères : en effet, dans ce programme il n'est utilisé que pour obtenir le champ `out` de la classe `System`.

5.5.2 Mesure de la taille des machines virtuelles obtenues

L'élimination des bytecodes inutilisés permet, comme nous l'avons dit, de couper des liens entre le système Java et son environnement d'exécution natif. Ces liens ont à leur tour des répercussions sur les parties de l'environnement natif à inclure : par exemple, si tous les bytecodes d'allocation mémoire sont omis, et que la méthode native `System.gc()` n'est pas référencée, alors il n'y a plus aucune référence vers les fonctions du gestionnaire mémoire qui peuvent être supprimées de l'environnement natif. Cette suppression se fait grâce à l'aide du compilateur : en effet, le compilateur GCC permet, lors de la construction d'un binaire statique, de demander à supprimer toutes les entités du binaire non-référencées. Ce mécanisme a été expliqué plus en détail dans la section 4.3.2.3.

Afin de mesurer la variété des empreintes mémoires que nous pouvons obtenir, nous avons généré aléatoirement 300 machines virtuelles (VM), chacune supportant un nombre aléatoire de bytecodes choisis aléatoirement. La décision d'inclure ou pas le support pour le

[18] Ceux-ci comprennent, bien entendu, les bytecodes du jeu d'instruction Java standard, mais aussi des variantes « accélérées » de certains d'entre eux.

TAB. 5.9 – Nombre de bytecodes différents présents dans les chemins d'exécution, inféré sur le système non-démarré.

Programme	Nombre de bytecodes
helloworld	78
raytracer	131
crypt	98
compress	102

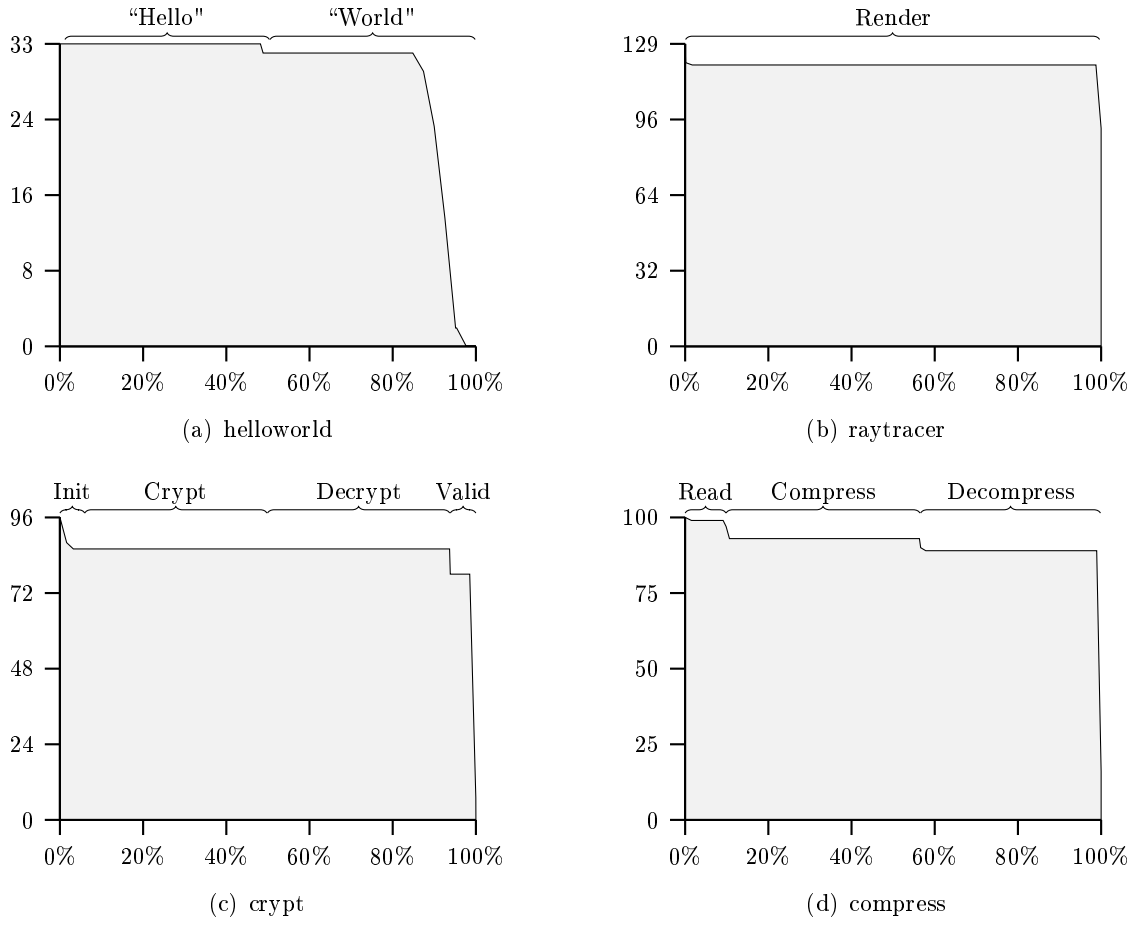


FIG. 5.10 – Nombre de bytecodes différents présents dans les chemins d'exécution inférés sur le système démarré, en fonction de l'avancement de son exécution.

multitâche a également fait l'objet d'un tirage aléatoire. Les compilations ont été effectuées par le compilateur GCC version 3.4.3, pour la plate-forme x86, avec le niveau d'optimisation 2. La figure 5.11 positionne chacune des machines virtuelles générées en fonction de son empreinte mémoire et du nombre de bytecodes qu'elle supporte.

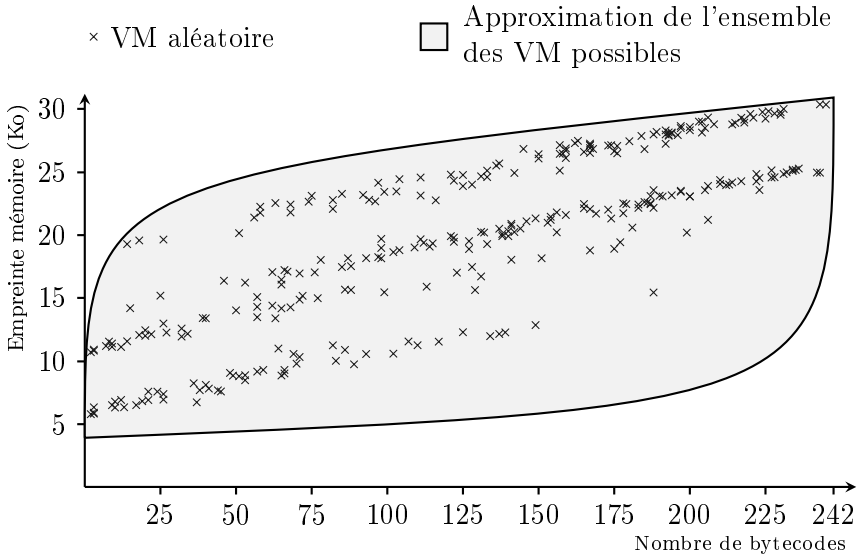


FIG. 5.11 – Empreintes mémoires obtenues sur le support d'exécution natif par rapport au nombre de bytecodes supportés, pour un ensemble de 300 machines virtuelles générées aléatoirement. La zone grisée donne une approximation empirique de l'ensemble des machines virtuelles pouvant être générées.

On constate logiquement que l'empreinte mémoire a tendance à augmenter avec le nombre de bytecodes supportés par la machine virtuelle, mais également qu'il existe une grande disparité dans les tailles obtenables pour des machines supportant le même nombre d'opérations. Tous les bytecodes n'ont en effet pas un coût égal : tandis qu'une instruction de la famille `iconst_*` a une implémentation immédiate (empiler une constante), une opération plus sophistiquée telle que `new` ou `invokeinterface` nécessite d'inclure d'autres mécanismes de la machine virtuelle (allocation mémoire ou résolution de méthode d'interface).

On peut également observer que le nuage de points formé par l'ensemble des machine virtuelles tend à s'organiser en paliers dont la croissance est linéaire : ici, on peut en observer quatre. Ils correspondent chacun à l'inclusion d'une fonction essentielle de la machine virtuelle. Le pallier le plus bas regroupe les machines virtuelles ne faisant ni appel au gestionnaire mémoire, ni à l'ordonnanceur de tâches. Les deux paliers centraux, eux, correspond aux machines virtuelles incluant l'une de ces deux caractéristiques. Enfin, le pallier le plus haut est composé des machines virtuelles nécessitant et le gestionnaire mémoire, et le gestionnaire de tâches.

Quoi qu'il en soit, cette mesure montre que le support d'exécution natif peut également passer à l'échelle inférieure pour ne supporter qu'un sous-ensemble de ses fonctionnalités originales, et ainsi réduire son empreinte mémoire.

5.6 Discussion sur les résultats expérimentaux

Dans ce chapitre, nous avons évalué nos travaux en mesurant divers critères de leur fonctionnement sur un prototype de l'analyse et de la spécialisation du système.

Il est apparu que le fait d'effectuer l'analyse sur l'image du système démarré réduit grandement le pessimisme de celle-ci, et par conséquent améliore le système spécialisé qui est produit. Plus intéressant encore, nous avons pu observer que les fonctionnalités non-utilisées des applications et du système disparaissent de la version spécialisée dès lors qu'elles ne sont plus utilisées, y compris quand elles se situent dans des blocs conditionnels de méthodes de l'API du système. Cette observation est particulièrement prometteuse puisqu'elle entend qu'il est possible avec notre solution de déployer un système générique et d'en obtenir une version très spécialisée. Nos chiffres, bien qu'obtenus sur des applications relativement simples, tendent à confirmer cette observation.

En effet, les résultats présentés dans ce chapitre correspondent à des résultats obtenus sur des programmes pratiquement imposés par les limitations de la plate-forme sur laquelle nous travaillons, et avec une implémentation partielle (mais reprenant les fonctionalités principales) de l'analyse que nous proposons. Malgré ces contraintes, nous sommes parvenus à réduire les programmes analysés vers des tailles qui les rendent embarquables, bien qu'ils n'aient pas été écrits dans cette optique. Plus important, nous sommes parvenus à démontrer que l'état démarré du système permet d'améliorer grandement la qualité de la spécialisation, validant ainsi notre postulat de départ qui est d'encourager le déploiement des systèmes hors-ligne afin d'améliorer leur potentiel de spécialisation. Par ailleurs, nous avons pu montrer qu'une réduction de l'empreinte mémoire et des fonctionnalités du support d'exécution natif est tout à fait possible si les parties non-utilisées par le système Java qui l'utilise ne sont plus référencées par ce dernier. Les outils de compilation permettent alors de supprimer ces parties de l'environnement d'exécution par un mécanisme de ramasse-miettes.

Nous considérons donc que ces résultats constituent une première validation de notre approche, et nous nous attendons à pouvoir les confirmer et les améliorer dès qu'une implémentation complète du processus de spécialisation sera disponible.

Sixième Chapitre

Conclusion

« Un étudiant demanda à Jigoro Kano :
- Quel est le secret du Judo ?
Kano répondit :
- Ne jamais cesser l'entraînement. »

En guise de conclusion, ce chapitre résume les contributions qui ont été présentées tout au long de ce document, souligne leurs limites et ouvre sur différentes perspectives d'utilisation de notre travail.

6.1 Résumé des contributions

Nous nous sommes intéressés à la problématique des systèmes Java embarqués. Les environnements Java pour systèmes embarqués sont conçus comme des variantes dégradées de l'édition standard de Java et sont incompatibles avec elle. Ils imposent leurs restrictions de manière arbitraire, dès la spécification, ce qui ne leur permet pas de convenir de manière concluante aux applications qui les utilisent. De plus, leur schéma de fonctionnement ne leur permet pas d'assumer leur déploiement complet hors-ligne, ce qui pose un autre frein à leur utilisation efficace dans le monde de l'embarqué.

Notre travail a apporté une réponse à ces problèmes au travers des contributions suivantes :

- Une définition unifiée du concept de romization, avec séparation des différentes préoccupations de ce processus ;
- À partir de cette définition, une architecture de romization permettant de déployer et de commencer l'exécution du système hors-ligne, puis de le capturer et de le transférer de manière transparente vers l'équipement cible au travers d'une *migration système*.
- Une méthode d'analyse de l'état du système capturé, permettant de synthétiser dans un état abstrait un ensemble de propriétés vraies pour tous les états atteignables à partir de celui-ci ;
- Divers procédures de spécialisation, permettant de spécialiser de manière très agressive l'état capturé du système Java vers un état équivalent sémantiquement, ce qui permet de réduire très fortement son empreinte mémoire avant son transfert vers l'équipement cible ;
- Une méthodologie pour spécialiser l'environnement d'exécution embarqué lié au système Java sur l'équipement cible, tirant avantage de la spécialisation opérée sur le système et des informations inférées par l'analyse ;
- Une évaluation du potentiel de spécialisation apporté par notre architecture de romization sur divers programmes réels.

De ces contributions, nous pouvons tirer une manière novatrice de concevoir des systèmes Java embarqués compatibles avec l'édition standard de Java. Il devient ainsi possible d'utiliser Java sur des équipements contraints sans avoir besoin de recourir à une version dédiée de l'environnement comme J2ME ou Java Card : les applications peuvent être déployées hors-ligne dans l'environnement de romization, puis le système tout entier être spécialisé en fonction de son utilisation future, fournissant un système Java taillé *sur mesure* pour les applications qu'il exécute. Les empreintes mémoires obtenues sont ainsi très faibles, car le système n'inclue que la partie nécessaire pour exécuter les applications qu'il embarque.

Ces travaux ont donné lieu à plusieurs publications dans des évènements d'audience internationale avec comité de sélection : sur le principe de romization des systèmes embarqués et l'intérêt de la spécialisation tardive [Cour 05a, Marq 05], sur la spécialisation sur mesure du support d'exécution embarqué [Cour 05b], sur un mécanisme de chargement de classes à faible empreinte mémoire [Ripp 04a] et enfin sur la fourniture d'informations de contexte sur les applications permettant de générer du code natif optimisé à partir du code Java [Cour 06].

Les perspectives d'exploitation de ces travaux restent à définir, mais l'on peut déjà imaginer un scénario industriel, dans lequel une entreprise pourrait proposer à ses clients d'effectuer eux-mêmes le déploiement d'un système embarqué dans l'environnement d'exécution virtuel de notre architecture de romization, à travers une interface Web. Le client pourrait ensuite décider lui-même de l'état initial utile de son système, le capturer, et obtenir à la volée une estimation de l'empreinte mémoire de son application déployée, de la quantité de silicium nécessaire à la réalisation de son produit, et du coût de production en fonction du nombre d'unités. Compte tenu de l'explosion des petits systèmes spécialisés, un tel scénario qui permettrait à tout un chacun de créer ses propres systèmes embarqués ne semble pas improbable, même si notre étude comporte encore certaines limites.

6.2 Limites de l'approche

Celles-ci concernent surtout les difficultés d'extensibilité des systèmes ainsi produits, et l'évaluation que nous avons faite de notre solution.

6.2.1 Extensibilité du système spécialisé

La limite la plus notable quant à notre approche concerne les possibilités d'extensibilité du système spécialisé. L'une des caractéristiques principales de Java concerne la possibilité de charger dynamiquement de nouveaux composants logiciels. Si cette faculté est pleinement préservée dans l'environnement de romization, elle est en revanche plus difficile à maintenir sur le système spécialisé et déployé dans son environnement d'exécution réel.

Notre spécialisation de l'environnement Java standard pour en obtenir une variante embarquable est certes tardive, ce qui permet de préserver la compatibilité de l'environnement au niveau applicatif, mais elle n'en garde pas moins les autres caractéristiques propres aux spécialisations que nous avions vues en 2.1.3.3 : notamment, le fait que la spécialisation soit irréversible et compromet l'extensibilité du système.

Si l'ensemble possible des classes chargeables dynamiquement est connu à l'avance, il est cependant possible de les inclure dans l'analyse du système. Ainsi, le système spécialisé est capable de prendre en charge non seulement les classes présentes en son sein au cours de

l'analyse, mais également l'ensemble des classes qu'il pourra être amené à charger dynamiquement. Le prix à payer est cependant une empreinte mémoire plus importante, même si les classes supplémentaires ne sont pas encore chargées.

Une autre solution contournant ce problème serait d'envoyer à l'équipement une mise à jour du système comprenant non seulement de nouvelles classes à charger, mais également les parties binaires qui lui manquent. Les mises à jour incrémentales [Kosh 05a] permettent de le faire de manière efficace, mais sans permettre toutefois d'assurer la sûreté de la mise à jour. De plus, elles interdisent quasiment le placement en ROM de certaines parties du système, puisque ce procédé peut potentiellement toucher n'importe quel aspect de celui-ci.

6.2.2 Limites de l'évaluation effectuée

L'implémentation de l'architecture de romization ainsi que des outils d'analyse et de spécialisation, de surcroît dans en environnement Java en cours de développement, s'avère être un énorme travail de génie logiciel. En conséquence de cela, tout n'a pas pu être implémenté de la manière souhaitée, et certaines concessions ont été nécessaires afin de satisfaire aux impératifs de temps.

Notamment, l'implémentation partielle de l'analyseur (bien que les éléments essentiels y soient) laisse à penser que les résultats obtenus pourraient être améliorés si celui-ci supporte l'ensemble de la spécification que nous avions donnée en 4.1.

De meilleurs résultats auraient également pu être obtenus pour la spécialisation de la partie Java par l'implémentation de spécialisations plus perfectionnées dans le contexte très avantageux de notre romizer : par exemple, la déclassification [Powe 05], la re-factorisation des prototypes de méthodes, l'optimisation du code des méthodes, etc. Toutefois, ces spécialisations viendraient se greffer au-dessus des spécialisations déjà existantes, et il est permis de penser que leur gain serait finalement assez anecdotique, compte-tenu des tailles déjà faibles obtenues avec les spécialisations de base.

En revanche, un type de spécialisation pratiquement ignorée dans notre étude concerne les spécialisations des méthodes Java. Nous n'avons effectué que la collection du code mort des méthodes, mais bien d'autres spécialisations peuvent être appliquées : reconception des prototypes de méthodes pour enlever les arguments constants, évaluation partielle pour spécialiser en profondeur le code des méthodes et réduire leur taille [Cons 93, Jone 97, Schu 03a], etc. Nous pensons que les résultats de ce genre de spécialisation sur ce qui reste du système serait significatif, l'environnement de romization leur permettant d'opérer dans d'excellentes conditions.

Plus généralement, nous pensons que la romization renferme un énorme potentiel pour le monde de l'embarqué, un potentiel qui est resté inexploité pendant de nombreuses années. Notre travail a permis de le révéler et d'en évaluer une des facettes, mais bien d'autres travaux qui sortent du cadre de notre étude restent à faire sur ce procédé.

6.3 Perspectives

Notre étude, bien que centrée sur la spécialisation de systèmes Java embarqués, ouvre des perspectives dans de nombreux autres domaines, notamment en raison de l'environnement particulièrement propice aux analyses que fournit notre architecture de romization.

Le contexte idéal offert pour l'analyse peut ainsi permettre d'inférer des informations utiles à bien d'autres tâches en dehors de la spécialisation du système. Le domaine de la preuve de programme, également important dans le monde de l'embarquer, pourrait ainsi gagner énormément à analyser des systèmes entièrement déployés et dont la richesse d'information permet de borner plus facilement des boucles ou d'inférer plus précisément les valeurs possibles de variables.

Une autre perspective du travail présenté ici serait de l'appliquer à d'autres systèmes que les environnements Java embarqués. Ceux-ci ont constitué un cadre avantageux pour notre étude, notamment de par leur utilisation d'une machine virtuelle. Mais dans quelle mesure la romization telle que nous l'avons décrite, et la spécialisation qui découle de l'analyse du système pré-déployé, peuvent-elles s'appliquer à des systèmes 100% natifs ? Nous pourrions notamment imaginer la romization de systèmes embarqués natifs (tels que Linux) au sein d'un émulateur de matériel, permettant à ceux-ci d'être instantanément fonctionnels, sans temps de démarrage, une fois placés dans leur équipement cible. Toutefois, les problématiques soulevées par la romization de tels systèmes semblent importants : contrairement à Java, ceux-ci ne disposent pas des capacités d'introspection qui nous ont permis d'analyser et de personnaliser le système finement. Il resterait également à définir les bonnes propriétés d'un environnement d'exécution virtuel natif, permettant de capturer l'intégralité du système qu'il est en train d'exécuter, et de le projeter non pas vers une machine virtuelle, mais vers une cible matérielle, avec toute la complexité que cela implique.

Enfin, nous pouvons également envisager de descendre encore plus bas dans les couches du système que nous spécialisons. La figure 4.15, page 94, nous montre que nous avons été capables de spécialiser les couches Java et natives du système. Il en reste toutefois une sur la figure que nous n'avons pas addressée : la couche matérielle. Étant donné que la cible du système n'existe pas encore pendant la romization, nous ne voyons aucune opposition à ce que celle-ci soit également spécifiée à partir de l'analyse du système, par exemple en language VHDL. De cette manière, le système fonctionnerait sur un matériel adapté, ce qui permettrait de réduire encore plus les coûts de celui-ci et appliquerait ainsi notre méthode à la chaîne d'exécution complète.

Bibliographie

[Alle 70] F. E. Allen. "Control flow analysis". In : *Proceedings of a symposium on Compiler optimization*, pp. 1–19, 1970.

[Ande 94] G. Andert. "Object Frameworks in the Taligent OS.". In : *COMPCON*, pp. 112–121, 1994.

[Beck 96] M. Beck, H. Bohme, U. Kunitz, R. Magnus, M. Dziadzka, and D. Verworner. *Linux Kernel Internals*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[Bers 94] B. N. Bershad, C. Chambers, S. J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. "SPIN - An Extensible Microkernel for Application-specific Operating System Services". In : *ACM SIGOPS European Workshop*, pp. 68–71, 1994.

[Bizz 02] G. Bizzotto. *JITS : Java In The Small*. Master's thesis, Université de Lille 1, 2002.

[Bluetoot 01] "Specification of the Bluetooth System v1.1". 2001.

[Bouc 00] S. Bouchenak and D. Hagimont. "Pickling Threads State in the Java System". In : *TOOLS '00 : Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, p. 22, IEEE Computer Society, Washington, DC, USA, 2000.

[Bouc 01a] S. Bouchenak. "Making Java Applications Mobile or Persistent". In : *6th USENIX Conference on Object-Oriented Technologies and Systems*, San Antonio, Texas, Jan. 2001.

[Bouc 01b] S. Bouchenak. *Mobilité et Persistance des Applications dans l'Environnement Java*. PhD thesis, INRIA Rhône-Alpes, Octobre 2001.

[Brun 02] E. Bruneton, T. Coupaye, and J. Stefani. "Recursive and dynamic software composition with sharing". In : *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga (Spain), June 2002.

[Cabr 00] G. Cabri, L. Leonardi, and F. Zambonelli. "Weak and Strong Mobility in Mobile Agent Applications". In : *Proceedings of the 2nd International Conference and Exhibition on The Practical Application of Java*, The Practical Application Company Ltd., Manchester, UK, April 2000.

[Camp 93] R. H. Campbell, N. Islam, D. Raila, and P. Madany. "Designing and implementing Choices : an object-oriented system in C++". *Commun. ACM*, Vol. 36, No. 9, pp. 117–126, 1993.

[Camp 96] R. Campbell, J. Coomes, A. Dave, N. Islam, Y. Li, W. Liao, S. Lim, T. Qian, D. Raila, E. Roush, A. Sane, M. Sefika, A. Singhai, and S. Tan. "Customizable Object-Oriented Operating Systems". 1996.

[Carz 98] A. Carzaniga, A. Fuggetta, R. S. Hall, A. van der Hoek, D. Heimbigner, and A. L. Wolf. "A Characterization Framework for Software Deployment Technologies". Tech. Rep. CU-CS-857-98, Dept. of Computer Science, University of Colorado, April 1998.

[Chen 00] Z. Chen. *Java Card Technology for Smart Cards : Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[Comi 03] M. Comini, R. Gori, G. Levi, and P. Volpe. "Abstract interpretation based verification of logic programs". *Sci. Comput. Program.*, Vol. 49, No. 1-3, pp. 89–123, 2003.

[Cons 93] C. Consel and O. Danvy. "Tutorial notes on partial evaluation". In : *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 493–501, ACM Press, 1993.

[Cost 05] R. Costa and E. Rohou. "Comparing the size of .NET applications with native code". In : *CODES+ISSS '05 : Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 99–104, ACM Press, New York, NY, USA, 2005.

[Cour 05a] A. Courbot, G. Grimaud, and J.-J. Vandewalle. "Romization : Early Deployment and Customization of Java Systems for Restrained Devices". Tech. Rep. RR-5629, INRIA Futurs, Lille, France, July 2005.

[Cour 05b] A. Courbot, G. Grimaud, J.-J. Vandewalle, and D. Simplot. "Application-Driven Customization of an Embedded Java Virtual Machine". In : *proceedings of the Second International Symposium on Ubiquitous Intelligence and Smart Worlds (UISW2005)*, Nagasaki, Japan, December 2005.

[Cour 06] A. Courbot, M. Pavlova, G. Grimaud, and J.-J. Vandewalle. "A Low-Footprint Java-to-Native Compilation Scheme Using Formal Methods". In : *Proc. 7th IFIP Conference on Smart Card Research and Advanced Applications (CARDIS'06)*, Springer-Verlag, Berlin, Tarragona, Spain, 2006.

[Cous 96] P. Cousot. "Abstract interpretation". *ACM Comput. Surv.*, Vol. 28, No. 2, pp. 324–328, 1996.

[Cram 97] T. Cramer, R. Friedman, T. Miller, D. Seherger, R. Wilson, and M. Wolczko. "Compiling Java Just in Time : Using runtime compilation to improve Java program performance". *IEEE Micro*, Vol. 17, No. 3, pp. 36–43, /1997.

[De M 97] G. De Michell and R. Gupta. "Hardware/software co-design". *Proceedings of the IEEE*, Vol. 85, No. 3, pp. 349–365, 1997.

[Deny 02] G. Denys, F. Piessens, and F. Matthijs. "A survey of customizability in operating systems research". *ACM Comput. Surv.*, Vol. 34, No. 4, pp. 450–468, 2002.

[Devi 02] D. Deville and G. Grimaud. "Building an "impossible" verifier on a Java Card". In : *Second Workshop on Industrial Experiences with Systems Software (WIESS'02)*, Boston (USA), 2002.

[Edwa 03] L. A. Edwards. *Embedded System Design on a Shoestring*. Newnes, 2003.

[Engl 95a] 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*, pp. 251–266, 1995.

[Engl 95b] D. R. Engler and M. F. Kaashoeker. "Exterminate all operating system abstractions". In : *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, p. 78, IEEE Computer Society, 1995.

[Engl 98] D. R. Engler. *The Exokernel Operating System Architecture.* PhD thesis, Massachusetts Institute of Technology, 1998.

[Fass 01] J.-P. Fassino. *THINK : vers une architecture de systèmes flexibles.* PhD thesis, École Nationale Supérieure des Télécommunications, December 2001.

[Fass 02] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. "THINK : A Software Framework for Component-based Operating System Kernels". In : *USENIX'02*, pp. 73–86, USENIX, Monterey, CA, USA, 2002.

[Fink 03] K. Finkenzeller. *RFID Handbook : Fundamentals and Applications in Contactless Smart Cards and Identification.* John Wiley & Sons, Inc., New York, NY, USA, 2003.

[Ford 97] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. "The Flux OSKit : A Substrate for Kernel and Language Research". In : *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pp. 38–51, ACM Press, 1997.

[Fugg 98] A. Fuggetta, G. P. Picco, and G. Vigna. "Understanding Code Mobility". *IEEE Trans. Softw. Eng.*, Vol. 24, No. 5, pp. 342–361, 1998.

[Fuji 05] K. Fujinami, K. Okada, F. Kawsar, and T. Nakajima. "Living with Sentient Artefacts". In : *7th International Conference on Ubiquitous Computing Video Proceedings (Ubicomp 05)*, 2005.

[Funf 98] S. Fünfrocken. "Transparent Migration of Java-Based Mobile Agents.". In : K. Rothermel and F. Hohl, Eds., *Mobile Agents*, pp. 26–37, Springer, 1998.

[Gajs 94] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and design of embedded systems.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[Gans 03] J. Ganssle and M. Barr. *Embedded Systems Dictionary.* CMPBooks, 2003.

[Gosl 96] J. Gosling and H. McGilton. *The Java Language Environment : A White Paper.* 1996.

[Grim 00] G. Grimaud. *CAMILLE : un Système d'Exploitation Ouvert pour Carte à Microprocesseur.* PhD thesis, Univ. Lille 1, France, dec 2000. in french.

[Grim 05] G. Grimaud, K. Marquet, and D. Simplot-Ryl. "Optimization of the Root Set for Object-Oriented Memory Management of Smart Objects". In : *Proc. 11th ECOOP Workshop on Mobile Object Systems (MOS-11)*, Glasgow, UK, 2005.

[Hall 02] S. D. Halloway. *Component development for the Java platform.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Han 05] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. "A dynamic operating system for sensor nodes". In : *MobiSys '05 : Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pp. 163–176, ACM Press, New York, NY, USA, 2005.

[Hong 05] H. S. Hong, I. Lee, and O. Sokolsky. "Abstract Slicing : A New Approach to Program Slicing Based on Abstract Interpretation and Model Checking.". In : *5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2005), 30 September - 1 October 2005, Budapest, Hungary*, pp. 25–34, 2005.

[Inc 99] C. A. S. Inc. *PostScript language reference (3rd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[J Co 02] J-Consortium. *JEFF Draft Specification*. March 2002.

[J2ME] "J2ME". http ://java.sun.com/j2me/.

[J2ME 00] *J2ME Building Blocks for Mobile Devices*. Sun Microsystems, 2000.

[Java 03] *Java Card Virtual Machine Specification*. 2003.

[JDie] "JDiet". http ://spoon.gforge.inria.fr/JDiet/Main.

[JITS] "JITS : Java In The Small". http ://jits.gforge.inria.fr.

[Joha 95] D. Johansen, R. van Renesse, and F. Schneider. "An Introduction to the TACOMA Distributed System Version 1.0". Tech. Rep. 95-23, University of Tromso, Department of Computer Science, 1995.

[Jone 93] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., 1993.

[Jone 94] N. D. Jones and F. Nielson. "Abstract Interpretation : a Semantics-Based Tool for Program Analysis". In : *Handbook of Logic in Computer Science*, Oxford University Press, 1994. 527–629.

[Jone 97] N. D. Jones. "Combining Abstract Interpretation and Partial Evaluation (Brief Overview)". In : *SAS '97 : Proceedings of the 4th International Symposium on Static Analysis*, pp. 396–405, Springer-Verlag, London, UK, 1997.

[Kicz 93] G. Kiczales, J. Lamping, C. Maeda, D. Keppel, and D. NcManee. "The need for customizable operating systems". In : *Fourth Workshop on Workstation Operating Systems*, pp. 165–169, 1993.

[Kosh 05a] J. Koshy and R. Pandey. "Remote incremental linking for energy-efficient reprogramming of sensor networks". In : *Proceeedings of the Second European Workshop on Wireless Sensor Networks*, pp. 354–365, 2005.

[Kosh 05b] J. Koshy and R. Pandey. "VMSTAR : synthesizing scalable runtime environments for sensor networks". In : *SenSys '05 : Proceedings of the 3rd international conference on Embedded networked sensor systems*, pp. 243–254, ACM Press, New York, NY, USA, 2005.

[Le M 04] A.-F. Le Meur, J. Lawall, and C. Consel. "Specialization Scenarios : A Pragmatic Approach to Declaring Program Specialization". *Higher-Order and Symbolic Computation*, Vol. 17, No. 1, pp. 47–92, 2004.

[LeJO] "LeJOS". http ://lejos.sourceforge.net/.

[Lero 01] X. Leroy. "Java Bytecode Verification : An Overview". In : *CAV '01 : Proceedings of the 13th International Conference on Computer Aided Verification*, pp. 265–285, Springer-Verlag, London, UK, 2001.

[Levi] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, *et al.* "TinyOS : An operating system for wireless sensor networks". *Ambient Intelligence.*

[Lian 98] S. Liang and G. Bracha. "Dynamic class loading in the Java virtual machine". In : *Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98)*, pp. 36–44, 1998.

[Lind 99] T. Lindholm and F. Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[Marq 04] K. Marquet. *Modèles de mémoire à objets persistants pour Java embarqué.* Master's thesis, Université des Sciences et Technologies de Lille, 2004.

[Marq 05] K. Marquet, A. Courbot, and G. Grimaud. "Ahead of Time Deployment in ROM of a Java-OS". In : *Proc. 2nd International Conference on Embedded Software and System (ICESS 2005)*, pp. 63–70, Springer-Verlag, Berlin, Xi'an, China, 2005.

[Miki 01] M. H. Miki, M. Sakamoto, S. Miyamoto, Y. Takeuchi, T. Yoshida, and I. Shirakawa. "Evaluation of processor code efficiency for embedded systems". In : *ICS '01 : Proceedings of the 15th international conference on Supercomputing*, pp. 229–235, ACM Press, New York, NY, USA, 2001.

[Mull 97] G. Muller, B. Moura, F. Bellard, and C. Consel. "Harissa : a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code". In : *Third USENIX Conference on Object-Oriented Technologies (COOTS)*, USENIX, Portland, Oregon, June 1997.

[Myer 95] W. Myers. "Taligent's CommonPoint : the promise of objects". *Computer*, Vol. 28, No. 3, pp. 78–83, Mar 1995.

[Necu 96] G. C. Necula and P. Lee. "Safe kernel extensions without run-time checking". In : *OSDI '96 : Proceedings of the second USENIX symposium on Operating systems design and implementation*, pp. 229–243, ACM Press, New York, NY, USA, 1996.

[Necu 97] G. C. Necula. "Proof-carrying code". In : *POPL '97 : Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 106–119, ACM Press, New York, NY, USA, 1997.

[OSGi 03] OSGi Alliance. *OSGi Service Platform, Release 3.* IOS Press, Inc., 2003.

[Pals 91] J. Palsberg and M. I. Schwartzbach. "Object-oriented type inference". In : *OOPSLA '91 : Conference proceedings on Object-oriented programming systems, languages, and applications*, pp. 146–161, ACM Press, New York, NY, USA, 1991.

[Powe 05] B. Power and G. W. Hamilton. "Declassification : Transforming Java Programs to Remove Intermediate Classes.". In : *5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2005), 30 September - 1 October 2005, Budapest, Hungary*, pp. 183–192, 2005.

[Proe 97] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. "Toba : Java For Applications : A Way Ahead of Time (WAT) Compiler". In : *Third USENIX Conference on Object-Oriented Technologies (COOTS)*, University of Arizona, Portland, Oregon, June 1997.

[Rash 89] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, and R. Sanzi. "Mach : A Foundation for Open Systems". In : *Proceedings of the Second Workshop on Workstation Operating Systems (WWOS2)*, September 1989.

[Rays 02] D. Rayside and K. Kontogiannis. "Extracting Java library subsets for deployment on embedded systems". *Sci. Comput. Program.*, Vol. 45, No. 2-3, pp. 245–270, 2002.

[Rigg 96] R. Riggs, J. Waldo, A. Wollrath, and K. Bharat. "Pickling State in the Java System". *Computing Systems*, Vol. 9, No. 4, pp. 291–312, 1996.

[Ripp 04a] C. Rippert, A. Courbot, and G. Grimaud. "A Low-Footprint Class Loading Mechanism for Embedded Java Virtual Machines". In : *3rd ACM International Conference on the Principles and Practice of Programming in Java*, Las Vegas (USA), 2004.

[Ripp 04b] C. Rippert, D. Deville, and G. Grimaud. "Alternative schemes for low-footprint operating systems building". Tech. Rep., INRIA-Futurs, 2004.

[Schu 03a] U. P. Schultz, J. L. Lawall, and C. Consel. "Automatic program specialization for Java". *ACM Trans. Program. Lang. Syst.*, Vol. 25, No. 4, pp. 452–499, 2003.

[Schu 03b] U. P. Schultz, K. Burgaard, F. G. Christensen, and J. L. Knudsen. "Compiling java for low-end embedded systems". In : *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pp. 42–50, ACM Press, 2003.

[Selt 97] M. Seltzer, Y. Endo, C. Small, and K. Smith. "Issues in extensible operating systems". Tech. Rep. TR-18-97, Harvard University, 1997.

[Shap 89] M. Shapiro, Y. Gourbant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. "SOS : An Object-Oriented Operating System - Assessment and Perspectives.". *Computing Systems*, Vol. 2, No. 4, pp. 287–337, 1989.

[Shay 03] N. Shaylor, D. N. Simon, and W. R. Bush. "A java virtual machine architecture for very small devices". In : *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pp. 34–41, ACM Press, 2003.

[Simu 99] T. Simunic, L. Benini, and G. D. Micheli. "Energy-efficient design of battery-powered embedded systems". In : *ISLPED '99 : Proceedings of the 1999 international symposium on Low power electronics and design*, pp. 212–217, ACM Press, New York, NY, USA, 1999.

[Stal] R. M. Stallman. *Using and Porting the GNU Compiler Collection*. Free Software Foundation.

[Sun 04] Sun Microsystems. "Java Object Serialization Specification, version 1.5.0". http ://java.sun.com/j2se/1.5.0/docs/guide/serialization/, 2004.

[Sund 00] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. "Practical virtual method call resolution for Java". In : *OOPSLA '00 : Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 264–280, ACM Press, New York, NY, USA, 2000.

[Suri 01] N. Suri. "State Capture and Resource Control for Java : The Design and Implementation of the Aroma Virtual Machine.". In : *Java Virtual Machine Research and Technology Symposium*, USENIX, 2001.

[Tane 87] A. S. Tanenbaum. "A UNIX clone with source code for operating systems courses". *SIGOPS Oper. Syst. Rev.*, Vol. 21, No. 1, pp. 20–29, 1987.

[Tenn 00] D. Tennenhouse. "Proactive computing". *Commun. ACM*, Vol. 43, No. 5, pp. 43–50, 2000.

[Tiny] "TinyVM". http ://tinyvm.sourceforge.net/.

[Tip 03] F. Tip, P. F. Sweeney, and C. Laffra. "Extracting library-based Java applications". *Commun. ACM*, Vol. 46, No. 8, pp. 35–40, 2003.

[Tip 95] F. Tip. "A survey of program slicing techniques.". *J. Prog. Lang.*, Vol. 3, No. 3, 1995.

[Truy 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 2000 : Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, pp. 29–43, Springer-Verlag, London, UK, 2000.

[Wand 87] M. Wand. "A Simple Algorithm and Proof for Type Inference". *Fundamenta Infomaticae*, Vol. 10, pp. 115–122, 1987.

[Weis 81] M. Weiser. "Program slicing". In : *ICSE '81 : Proceedings of the 5th international conference on Software engineering*, pp. 439–449, IEEE Press, Piscataway, NJ, USA, 1981.

[Weis 93] M. Weiser. "Hot Topics : Ubiquitous Computing". *IEEE Computer*, Oct. 1993.

[Wong 99] D. Wong, N. Paciorek, and D. Moore. "Java-based mobile agents". *Commun. ACM*, Vol. 42, No. 3, 1999.

[Yerg 98] F. Yergeau. "UTF-8, a transformation format of ISO 10646". 1998.

[Yoko 92] Y. Yokote. "The Apertos reflective operating system : The concept and its implementation". In : A. Paepcke, Ed., *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 414–434, ACM Press, New York, NY, 1992.

[μCL] "μCLinux". http ://www.uclinux.org/.

Résumé

Java est une technologie attractive pour les équipements embarqués et contraints, de par ses propriétés de sûreté, de portabilité et de faible empreinte mémoire du code. Cependant, la taille imposante d'un environnement Java complet a obligé les producteurs d'équipements embarqués à utiliser des spécifications dégradées de Java aux fonctionnalités limitées, telles que J2ME ou Java Card. Ces spécialisations précoces de Java perdent la compatibilité au niveau applicatif avec l'édition standard, et ne peuvent ainsi s'adresser qu'à des cas d'utilisation particuliers.

Notre travail consiste à permettre l'utilisation de l'édition standard de Java sur les systèmes contraints, au travers d'une spécialisation tardive et agressive du système qui intervient après déploiement de ses applications. L'occurrence tardive de la spécialisation permet de mieux déterminer les conditions d'utilisation du système, et donc de le spécialiser « sur mesure » par rapport aux applications qu'il exécute.

Nos contributions sont les suivantes : dans un premier temps, nous définissons la notion de « romization », consistant à déployer un système hors-ligne avant de capturer et de transférer son image mémoire vers l'équipement sur lequel il doit s'exécuter. De cette définition, nous proposons une architecture de romization capable de capturer une image mémoire du système à n'importe quel moment de son exécution. Dans un second temps, nous traitons des moyens d'analyse et de spécialisation permettant de rendre cette image mémoire embarquable. L'évaluation effectuée montre que cette spécialisation tardive, appliquée à un environnement Java standard déployé, permet effectivement d'en obtenir une version minimaliste et embarquable sur un équipement contraint.

Abstract
Late specialization of embedded Java systems for small and restrained devices

Java is an attractive technology for embedded and constraint devices, thanks to its safety, portability, and low bytecode footprint properties. However, the important size of a Java environment obliged embedded devices producers to use degraded and features-limited specifications of Java, like J2ME and Java Card. These early specializations of Java lose applicative-level compatibility with the standard edition, and only address particular use cases.

Our work consists in allowing using standard Java on constraint embedded systems, through a late and agressive specialization that occurs after the deployment of applications on the system. The late occurrence of specialization allows to infer the usage conditions of the system more accurately, and to tailor it "on demand" according to the applications that run on it.

Our contributions are as follows: first, we define the "romization" notion, which consists in deploying a system off-line before capturing and transfering its memory image to its target device. From this definition, we propose a romization architecture that permits to capture a memory image of a system at any time of its execution. Then, we address analysis and specialization techniques allowing to produce an embeddable version of this memory image. Our evaluation shows that late specialization makes it possible to obtain an embeddable version out of a deployed standard Java environment.