UNIVERSITÉ DE LIMOGES
ÉCOLE DOCTORALE « Sciences et Ingénierie pour l'Information »
FACULTÉ DES SCIENCES ET TECHNIQUES

Thèse N° XX-2011

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE LIMOGES

Discipline / Spécialité : Informatique

présentée et soutenue par

Agnès Cristèle NOUBISSI

le 12 Décembre 2011

Mise à jour dynamique pour cartes à puce Java

Thèse dirigée par Jean-Louis LANET,
Co-encadrée par Julien IGUCHI-CARTIGNY

JURY

Rapporteurs :	
M. Daniel Hagimont	Professeur à l'INPT/ENSEEIHT
M. Gilles Grimaud	Professeur à l'Université de Lille
Examinateurs:	
M. Pierre Girard	Ingénieur de recherche Gemalto (HDR)
M. Jean-Louis Lanet	Professeur à l'Université de Limoges
M. Julien Iguchi-Cartigny	Maître de Conférences à l'Université de Limoges

Remerciements

Mes plus vifs remerciements s'adressent au Pr Jean-Louis Lanet, mon directeur de thèse, sans qui cette thèse ne serait pas ce qu'elle est. Tout au long de ces trois années, il a su orienter mes recherches aux bons moments, il a toujours été disponible pour toutes mes multiples sollicitations. Pour tout cela, pour sa confiance et la liberté qu'il m'a accordé durant tout mon travail de thèse, pour son soutien sans faille en fin de thèse, je le remercie vivement.

Je remercie également Dr Julien Iguchi-Cartigny, mon co-encadrant de thèse, pour ses conseils avisés, il a toujours fait preuve d'une grande écoute à mon égard, de beaucoup de patience et a toujours su me remotiver dans mes moments de doute.

Je remercie les rapporteurs de cette thèse Pr Gilles Grimaud et Pr Daniel Hagimont pour la rapidité avec laquelle ils ont lu mon manuscrit et l'intérêt qu'ils ont porté à mon travail. Merci également aux autres membres du jury qui ont accepté de juger ce travail notamment Pierre Girard.

Je remercie mon fiancé Ervé de m'avoir soutenu dans les moments les plus difficiles ces dernières années, surtout durant la fin de thèse.

Merci à tous les doctorants pour les moments conviviaux que nous avons passé ensemble. Je pense tout particulierement à Alex, Amaury, Boussad, Céline, Cyril, David, Emma, Guillaume, Julien, Mickael, Nadir, Nassima, Oana, Richard, Selim et Yves.

Un merci tout particulier à Amaury, Clement, David, Guillaume, Nassima et Jean-Baptiste d'avoir eu la patience de relire cette thèse.

A Hubert et Sylvie pour leur amitié.

Une pensée émue pour tous mes amis de longues dates qui ont toujours été là : Aude, Gisèle, Irène, Alain, Patrick, Raymond, Elvadas, Rodrigue, Simplice, Innocent, Désiré, Larissa, Suzy et beaucoup d'autres. J'en oublie certains mais cela ne veut pas dire que je ne pense pas à eux au moment où j'ecris ces mots.

Je remercie tout particulierement la famille Fakam, pour tout le soutien dont elle m'a fait part durant ces années.

Je tiens également à remercier la famille De Vanssay pour tout le temps qu'ils m'ont consacré, pour leur disponibilité, pour leur chaleur familiale et leur gentillesse à mon égard.

Je termine par une très forte pensée à ma famille et plus particulièrement à ma mère Nganyou Rose et mon père Gomsu Jean-Balland, pour leur soutien incommensurable. Merci à mes deux frères Joël, Jude et mes deux soeurs Ingrid et Karelle.

Résumé

Contrairement à la mise à jour traditionnelle, la mise à jour dynamique est la capacité de pouvoir modifier un système logiciel ou une application durant son exécution sans interruption de ce dernier et sans perte de l'état d'exécution du système. L'objectif est de permettre aux développeurs de pouvoir corriger les défauts (mettre à jour une partie fautive du système) ou de faire évoluer leurs applications (ajout, suppression de nouvelles fonctionnalités).

Les cartes à puce sont de petits ordinateurs dotées d'au moins un circuit intégré ou puce capable de contenir de l'information. Ce circuit intégré peut contenir un microprocesseur capable de traiter l'information. Parmi les types de cartes, on distingue la Java Card, basée sur la technologie Java. C'est une carte multi-applicative avec les applications qui s'exécutent au dessus de la machine virtuelle Java embarquée dans la carte. Ces cartes ont des durées de vie de plus en plus élevée. Il est difficile d'imaginer que les applications embarquées n'aient pas un besoin de mise à jour pour solutionner un bogue, mettre à jour une faille de sécurité, améliorer des fonctionnalités ou en supprimer, ceci de façon transparente à l'utilisateur.

De nombreux systèmes existants offrent des mécanismes de mise à jour dynamique. Cependant, ces systèmes sont généralement destinés à des environnements n'ayant pas de fortes contraintes de ressources (environnement de type poste de travail). De plus, les approches de solutions proposées ne sont pas toujours applicables au domaine des objets à faibles ressources tels que les cartes à puce.

Dans cette thèse, nous présentons EmbedDSU, un système de mise à jour dynamique pour les cartes à puce basées sur Java. Ce système est basé sur un ensemble de mécanismes à l'extérieur de la carte et à l'intérieur de la carte. Pour évaluer notre système, nous avons réalisé un prototype basé sur simpleRTJ[1], une machine virtuelle Java pour l'embarqué.

Mot-clefs : *Carte à puce*, *Java Card*, *Mise à jour dynamique*, *DSU*.

[1] simple Real Time Java

Abstract

Unlike the traditional update, the dynamic update is the ability to modify a software system or an application during its execution without stopping it and without loosing its execution state. The aim is to enable developers to be able to correct bugs, improve and delete some functionnalities.

Smart cards are small computers equipped with at least one integrated circuit or chip that contain information. This integrated circuit can contain a microprocessor capable of processing information. Java Card is a type of smart card based on Java technology. This is a multi-application card which allows application to run on the top of the embedded virtual machine.

These cards may have a long lifetime. It is hard to imagine that embedded application do not need an update to solve a bug, update security holes, improve or remove features, in transparent manner to the user.

Many existing systems provide mechanisms for dynamic update. However these systems are generally intended for environments that do not have strong resource constraints (like workstation environment). In addition, the existing approaches of solutions are not always applicable to the field of smart cards.

In this thesis, we present EmbedDSU, a dynamic update system for smart cards based on Java. This system consists of a set of mechanisms off-card and on-card. To evaluate our system, we have implemented a prototype based on simpleRTJ[2], a Java virtual machine for embedded systems.

Keywords : *smart card*, *Java Card*, *dynamic update*, *DSU*

[2]simple Real Time Java

Table des matières

Liste des tableaux

Table des figures

Première partie

Introduction

Chapitre 1

Introduction

Sommaire

La prolifération des systèmes embarqués, les avancées technologiques et l'apparition des nouvelles technologies autour du réseau sans fil ont permis un développement rapide des systèmes enfouis et embarqués. Ces systèmes suscitent un intérêt croissant dans l'industrie. Ceci est dû aux enjeux considérables qu'ils représentent par le potentiel des progrès technologiques qu'ils offrent dans des domaines variés tels que l'aéronautique, l'aérospatiale, l'automobile, le multimédia, les télécommunications, les maisons intelligentes, etc. En effet, on note une utilisation de plus en plus massive de ces systèmes basés sur des puces. Les cartes à puce en sont un exemple (carte bancaire, carte vitale, carte Navigo, etc.).

Pour faire face aux nouvelles fonctionnalités ou à l'amélioration des fonctionnalités existantes des logiciels embarqués, ces systèmes sur puces doivent être régulièrement mis à jour. La mise à jour traditionnelle pour ces systèmes consiste à écrire une nouvelle image du système et à le redémarrer. Cependant, ils existent des systèmes critiques où les applications ne peuvent pas être interrompues notamment les systèmes de contrôle de trafic aérien. Un arrêt ou un redémarrage d'un tel système implique la perte de l'état d'exécution pouvant entraîner de nombreuses conséquences. Il est donc nécessaire d'opter pour des solutions de mise à jour sans interruption de service et sans perte d'état : c'est la mise à jour dynamique.

1.1 Définition

Nous définissons la mise à jour dynamique, encore appelé *Dynamic Software Update* (DSU) ou *Hot Software Update* (HotSwUp), comme :

La possibilité de pouvoir modifier un système logiciel ou une application durant son exécution sans interruption de ce dernier et sans perte de l'état d'exécution du système.

En d'autre termes, – contrairement à la mise à jour statique ou traditionnelle – l'ajout, la suppression ou la modification de tout ou une partie du système (ou de l'application) est effectuée durant l'exécution et ceci de façon transparente pour l'utilisateur. Dans la littérature, d'autres concepts sont proches de celui de la mise à jour dynamique notamment l'extensibilité, l'adaptation dynamique et la reconfiguration dynamique.

L'extensibilité [BSP+95] se réfère à des mécanismes permettant d'ajouter ou d'étendre les fonctionnalités d'un système ou d'une application sans nécessairement effectuer une mise à jour des parties existantes.

L'adaptation dynamique [CFW10, Ham10] représente un changement du système en réponse à une modification du contexte dans lequel il se trouve. Une adaptation dynamique peut par exemple faire suite à l'ajout de ressources ou bien précéder le retrait de ressources à l'application. Dans ce cas, la modification de l'application doit lui permettre de prendre en compte ces changements.

La reconfiguration dynamique [Pol08] peut se définir comme l'ensemble des changements apportés à une application en cours d'exécution. Ces changements pouvant être :
(1) La modification de l'architecture de l'application (ajout, retrait des modules et de leurs liaisons) ;
(2) La modification de la distribution géographique de l'application ;
(3) La modification de la mise en œuvre d'un composant ;
(4) La modification des interfaces de services, etc.

Le problème de mise à jour dynamique, quant à lui, peut être considéré comme un cas particulier d'une problématique plus vaste qui est la reconfiguration dynamique, car elle permet tout en modifiant le système, de passer d'un état d'exécution (ou d'une configuration donnée) du système vers un autre.

1.2 Objectifs de la mise à jour dynamique dans les cartes à puce

La DSU est un sujet qui mobilise une grande communauté de chercheurs depuis des années. De nombreux travaux ont été réalisés dans le domaine, principalement pour des applications s'exécutant sur des postes de travail. L'idée est de permettre aux développeurs de corriger des bogues, d'ajouter, supprimer ou améliorer des fonctionnalités de leurs applications (ceci après déploiement), que celles-ci soient en cours d'exécution ou non.

La carte à puce peut aussi requérir le besoin de corriger des défauts dans une application préalablement chargée, ou peut nécessiter d'adapter le système à un nouvel environnement.

Correction des défauts : Après conception, vérification et validation de l'application dans le but d'éliminer toute source d'erreur, celle-ci est chargée dans la carte afin d'être utilisée. Toutefois, des défauts de l'application peuvent être constatés lors d'un chainage de commandes inattendues,

des erreurs de protocole cryptographique peuvent être décelées voire même des erreurs de conception. La capacité de mise à jour dynamique permet donc de corriger ces bogues sur ces applications après leurs déploiements sur la carte.

Nouvelles versions : Les besoins d'un système ou d'une application de la carte peuvent évoluer dans l'objectif de modifier son comportement ou de proposer de nouvelles fonctionnalités. L'objectif étant d'améliorer des fonctionnalités existantes de l'application et/ou développer de nouvelles fonctionnalités pour faire face à de nouvelles exigences. Dans ces cas, une fois la nouvelle version disponible, la mise à jour dynamique permet de passer de l'ancienne version à la nouvelle version de l'application sans perte d'état du système et de façon transparente pour l'utilisateur de la carte.

1.3 Objectifs de la thèse

Cette thèse est à placer dans le contexte des petits objets embarqués, notamment les cartes à puce, basées sur une machine virtuelle Java. L'objectif de cette thèse est de fournir une approche de mise à jour dynamique pour des cartes à puce Java. Cette approche dans le cadre idéal devrait avoir les propriétés suivantes :

- La flexibilité : pouvoir gérer l'ensemble des types de modifications qui peuvent être effectuées par un développeur et ceci à tous les niveaux de granularité de l'application ;
- Pas d'intervention humaine : Il s'agit de ne pas requérir une intervention humaine quelconque, après avoir reçu les entrées nécessaires pour effectuer la mise à jour ;
- Sûreté : l'approche doit garantir aux développeurs et aux utilisateurs que la nouvelle version est fiable, que le système reste cohérent, que la sémantique des programmes est préservée, qu'après le transfert d'état le nouvel état correspond bien à la nouvelle version et, surtout, que cette mise à jour n'apporte pas de nouvelles failles de sécurité ;
- Efficacité : il s'agit de garantir qu'il n'y a pas de perte de performances due au processus de mise à jour et/ou à l'exécution de la nouvelle version de l'application. Il est vrai qu'un système de mise à jour dynamique est une opération ou une fonctionnalité supplémentaire, cependant, celle-ci doit avoir un coût minimal.

L'objectif de cette thèse est donc de définir un mécanisme de mise à jour dynamique pour carte à puce Java avec les propriétés citées ci-dessus. Ce mécanisme doit pouvoir définir *le quoi, le quand, et le comment* de la mise à jour. Il doit donc fournir :

- Une approche générique et des algorithmes permettant de déterminer les modifications apportées entre deux versions d'une application Java ;
- Une méthode et des algorithmes permettant de déterminer le moment (point « sûr ») auquel la mise à jour de l'application en cours d'exécution peut être réalisée tout en préservant la cohérence du système ;
- Une méthode et des algorithmes pour effectuer la mise à jour proprement dite pour obtenir la nouvelle version de l'application et le nouveau contexte du système correspondant à partir

de l'ancienne version.

L'association de ces trois éléments et la définition d'une relation entre eux, définit le système de DSU que nous appellerons EmbedDSU. Nous développerons aussi dans cette thèse, des outils afin de valider par prototypage notre approche. En effet, disposer d'un prototype nous permet de quantifier l'approche proposée afin de s'assurer que nos choix sont compatibles avec les contraintes d'une carte à puce.

EmbedDSU, le système – de mise à jour dynamique pour carte à puce Java – proposé dans cette thèse est basé sur un ensemble de mécanismes à l'extérieur (*off-card*) et à l'intérieur de la carte (*on-card*) avec comme relation ou liaison entre les deux parties, un fichier exprimé dans un langage dédié (DSL[3]) conçu à cet effet.

1.4 Organisation du document

Cette thèse est organisée en trois parties :

Contexte de l'étude et état de l'art

Le chapitre 2 présente les notions, les définitions nécessaires pour comprendre la problématique de DSU dans le contexte Java et Java Card.
Le chapitre 3 présente les problèmes scientifiques de DSU, les solutions possibles, celle traitées dans la littérature et celle proposées dans le cadre d'EmbedDSU pour le cas des systèmes à fortes contraintes de ressources.

Contribution et évaluation

Le chapitre 4 décrit l'architecture d'EmbedDSU et présente les algorithmes proposés pour chaque problème scientifique de DSU.
Le chapitre 5 développe l'implémentation d'un prototype grâce à une machine virtuelle Java pour l'embarqué appelé simpleRTJ et une évaluation de celui-ci tout en expliquant les résultats obtenus sur la plateforme d'évaluation AT91 EB40A.

Perspectives et conclusion

Le chapitre 6 présente les perspectives des travaux de recherche notamment l'adaptation d'EmbedDSU pour la mise à jour dynamique dans les plateformes orientées services basées sur OSGi et l'étude d'une proposition d'un cadre formel pour la vérification de la validité du processus de DSU proposé.
Le chapitre 7 présente une conclusion de l'ensemble du document et une synthèse des contributions de cette thèse.

[3]Domain Specific Language

Deuxième partie

Contexte de l'étude et état de l'art

Chapitre 2

Contexte de l'étude

Sommaire

Cette thèse concerne la mise à jour dynamique des composants logiciels sur carte à puce Java, partant du constat que les cartes ont une durée de vie de plus en plus élevée. On peut citer par exemple le cas du passeport électronique (E-Passport) dont la durée de vie est de dix ans environ (cinq ans renouvelable). De même, une carte bancaire peut avoir une validité plus importante que celle fixée de manière logicielle par le système bancaire. En réalité, la validité d'une carte bancaire est limitée par le nombre de transactions de la carte (nombre d'opérations d'écriture dans l'EEPROM [4]).

Il est difficile d'imaginer qu'une carte à puce puisse résister à de nouvelles attaques (logiques ou cryptographiques) sur une longue durée. Dans le cas des E-Passports, si une faiblesse algorithmique est découverte, l'unique solution est de retourner tous les passeports à l'organisme émetteur et d'en obtenir des nouvelles. Ce qui représente un coût considérable.

Le patch dynamique peut servir non seulement pour des algorithmes cryptographiques (dans les modules systèmes) mais aussi pour des applications simples s'exécutant sur la carte.

[4] Electrically-erasable programmable read-only memory

En effet, il est aussi difficile d'imaginer que les applications embarquées n'aient pas un besoin de mise à jour pour solutionner un bogue, améliorer des fonctionnalités ou en supprimer.

Prenons l'exemple du bogue de carte bancaire de l'année 2010 où suite au passage de l'année 2009 à 2010, des millions de cartes sont devenues inutilisables. En effet, dès le 1er Janvier 2010, près de 30 millions d'allemands n'ont pas pu utiliser leurs cartes bancaires puisque les lecteurs de cartes, plus précisément les distributeurs automatiques de billets (*DAB*) et les terminaux points de vente n'étaient plus en mesure de lire les cartes.

Les DABs n'étaient pas la cause du problème, il s'agissait plutôt d'une erreur de programmation dans une application de la carte dû au codage en l'année 2010 en hexadécimal qui était lu par les distributeurs comme correspondant à l'année 2016. Ainsi, les cartes étaient considérées comme non valides par les lecteurs de cartes.

La carte ne disposant pas encore de mécanisme de mise à jour dynamique, une solution a été d'appliquer une procédure corrective afin d'éviter le remplacement des cartes concernées. Cette procédure consistait à appliquer un patch à tous les terminaux points de vente et tous les DABs, ce qui est resté très coûteux au niveau financier, temporel et en investissement humain.

Une solution de DSU, aurait été alors plus intéressante et très efficace.

Dans la suite, nous décrivons les concepts liés à la mise à jour dynamique Java et les problématiques liées à celle-ci dans le cas des applications Java en général et des applications cartes à puce Java en particulier.

2.1 Java

La technologie Java est caractérisée par le langage à proprement dit et d'une plateforme fournissant un environnement de développement et d'exécution des applications Java.

L'environnement de développement aussi appelé JDK [5] est indispensable à la création de programmes Java grâce aux APIs[6] dont il dispose.

L'environnement d'exécution Java (JRE[7]) se compose d'une machine virtuelle Java (JVM[8]), et d'une bibliothèque Java de classes de base permettant d'exécuter les programmes Java sur différentes plates-formes matérielles.

Les sections suivantes décrivent quelques éléments de l'environnement d'exécution notamment la machine virtuelle Java et la structure d'une application Java.

2.1.1 Machine virtuelle Java (JVM)

Par définition, une JVM est une machine abstraite définie par une spécification. La spécification d'une JVM peut être vue en termes de type de données, de système de chargement et d'exécution des applications.

[5] Java Development Kit
[6] Application Programming Interfaces
[7] Java Runtime Environment
[8] Java Virtual Machine

Type de données

La machine virtuelle manipule deux catégories de types : les types primitifs et les types de références. Afin de permettre la portabilité, la JVM et le langage Java spécifient la taille et le format des types primitifs. On peut citer par exemple comme types primitifs *byte*, *short* et *int* qui sont représentés respectivement sur 8, 16, 32 bits en mémoire mais sur 32 bits sur la pile Java. Les types de références représentent les références sur une instance de classe ou un tableau. La valeur *NULL* représente un pointeur vers aucune instance de classe ou vers aucun tableau.

Structures de données d'exécution

La mémoire de la machine virtuelle est organisée en structure de données pour stocker différents types d'informations tels que les objets créés, les valeurs de retour de méthodes, les variables locales de méthodes, les opérandes des méthodes, etc. Ces structures de données peuvent être le tas (Heap), la zone de méthode et les piles Java (pile de threads). On peut aussi avoir des piles de méthodes natives pour les codes natifs appelés par les applications Java.

1. **Le tas de la JVM**
   C'est est une zone partagée par tous les threads de la machine virtuelle. Elle permet de stocker toutes les instances ou de tableau. A chaque objet d'instance est associé une référence qui permet d'accéder aux valeurs de chaque champ de l'objet et aux méthodes associées à la classe de l'objet. Ces références peuvent être par la suite stockées dans la pile soit comme paramètre d'une méthode, soit comme opérande d'une instruction, ou être stockées dans un champ d'un autre objet dont la classe possède une relation de composition avec la classe de l'instance créée.

2. **Une frame**
   A chaque appel de méthode, une nouvelle frame est créée et empilée sur la pile Java du thread associé. Cette frame est dépilée à la fin de l'exécution de la méthode. Une frame permet de stocker toutes les informations nécessaires à l'exécution de la méthode telles que les variables locales, les paramètres et valeurs d'appels de la méthode, les résultats des calculs intermédiaires, etc.

   Une frame est donc composée d'un ensemble de structure de données telles que :
   - La table de variables locales : elle stocke les arguments de la méthode et les variables locales à la méthode. Les variables sont indicées à partir de 1 dans la table permettant ainsi un accès direct aux entrées de celle-ci. L'entrée 0 de cette table contient la référence du *this* (l'instance à partir de laquelle la méthode a été appelée).
   - La pile d'opérandes : elle contient les résultats intermédiaires résultants des calculs intermédiaires ou des valeurs de retour d'appels intermédiaires de méthodes.
   - Des informations complémentaires : il peut s'agir d'un registre contenant l'adresse de la prochaine instruction à exécuter appelé communément *Program Counter* (PC), d'une référence vers la table de constante de la classe de la méthode, d'une référence vers la table d'exception de la méthode, etc.

3. **La pile Java**
   A chaque thread exécuté est associé une pile Java. La pile Java d'un thread est constituée par

les appels de méthode. A chaque appel de méthode est associée une frame et cette dernière est empilée sur la pile. L'ensemble des frames relatives à l'ensemble des appels de méthodes d'un thread est appelé pile Java. On peut aussi noter le concept de pile de thread qui représente l'ensemble des piles Java de la machine virtuelle, appartenant donc à tous les threads s'exécutant dans la VM.

4. **La pile de méthodes natives**
   Une méthode native est une méthode écrite dans un autre langage que Java, généralement il peut s'agir du langage C ou de l'assembleur. L'interface JNI (*Java Native Interface*) se charge de faire communiquer le code Java et le code natif. Lorsqu'un thread ou une application Java invoque une méthode native, c'est la pile de méthodes natives qui est utilisée. La structure de cette pile dépend du système sous-jacent.

5. **La zone de méthode**
   C'est une zone partagée par tous les threads. Dans cette zone sont mémorisées les classes chargées dans la VM. Chaque classe contient les informations nécessaires à son utilisation. Ces informations peuvent être les tables de méthodes de la classe, table de champs et la table de constante. Ces informations sont décrites en détails plus loin dans la section 2.1.1.

Système de chargement et d'exécution

Chaque VM possède un système de chargement de classe dans la zone de méthode. Ce système est constitué d'un ensemble de chargeurs de classes (*classloader*). Ces chargeurs de classe peuvent être hiérarchiques ou pas. Il est à noter que le processus de chargement pour une machine virtuelle standard diffère du processus de chargement dans le domaine Java Card (voir section 2.1.3).

Système de chargement standard

L'exécution d'une classe nécessite tout d'abord son chargement dans la VM.

Au sein de la VM, une classe suit un cycle de vie précis de son chargement à son retrait. Ce cycle de vie peut être définit comme suit :
(1) Chargement de la classe proprement dit ;
(2) Liaison ;
(3) Initialisation ;
(4) Instanciation ;
(5) Retrait.

1. **Le chargement**
   Le processus commence par le chargement de la classe grâce à un chargeur de classe. Il charge dynamiquement et initialise les classes et interfaces requises par la VM lors de l'exécution d'une application. Il hérite de la classe *java.lang.ClassLoader* et peut appartenir à un système de délégation du chargeur de classe. Il effectue généralement les opérations suivantes :
   - Vérification de l'existence de la classe dans la mémoire ;
   - Chargement du byte code relatif à la classe dans la zone de méthode ;
   - Initialisation de la classe dans la VM.

2. **La Liaison**
   Après le chargement du byte code des méthodes de la classe dans la mémoire, la liaison de la classe est effectuée. Ce processus est aussi appelé la résolution. Il s'agit de remplacer toutes les références symboliques de la table de constante (voir 2.1.3) en références mémoires.
3. **L'initialisation**
   Après la résolution de liens, la machine virtuelle alloue de l'espace dans le tas de la VM pour la création et l'initialisation de l'instance associée à la classe et l'exécution peut débuter.

Système d'exécution

Chaque VM possède aussi un système d'exécution. Ce système d'exécution peut consister en un interpréteur de byte code (résultat de la compilation des instructions Java). Cet interpréteur peut donc être chargé d'exécuter les byte codes des méthodes des classes chargées en mémoire. Cet interpréteur peut être matériel (on peut citer l'exemple de Jazelle [Lab09] qui permet à certains processeurs ARM[9] de faire exécuter directement le byte code Java par le matériel), logiciel ou mixte. L'interpréteur peut aussi être couplé avec un compilateur à la volée (JIT[10]), tout dépend de la plateforme et de l'implémentation du système d'exécution.

2.1.2 Structure d'une classe

Une classe Java est constituée de membres données (variables d'instances ou statiques), de membres méthodes et d'un ensemble d'attributs (interfaces, droit d'accès de la classe, etc.).

Paquetage ou package

Un paquetage est un mécanisme d'espace de nommage pour regrouper des classes, des interfaces, et d'autres sous paquetages. Le regroupement de classes ou d'interfaces en paquetage permet d'éviter les collisions entre classes ayant le même nom. Le partage de classe entre les différents paquetages est effectué grâce au mécanisme d'import.

Classe

Une classe est un type défini par le programmeur. Elle regroupe dans une seule entité les données contenues dans une instance et les opérations qui peuvent s'effectuer sur celles-ci.

Une classe peut hériter d'une autre classe pour redéfinir certaines méthodes. Plus exactement, l'héritage de classe est un mécanisme de sous-typage permettant de construire de nouvelles classes par spécialisation d'une classe existante. Ainsi il est possible d'obtenir des classes filles capable de redéfinir les membres données et membres méthodes de la classe mère et spécifier de nouvelles caractéristiques propre à elle-même.

Une classe peut aussi être abstraite si elle possède au moins une méthode abstraite. Une classe abstraite ne peut-être instancié, seules les classes filles peuvent l'être.

[9] Advanced RISC Machines
[10] Just In-time Translation

Une classe peut implémenter une ou plusieurs interfaces et permettre ainsi de réaliser une forme héritage multiple. Mais cet héritage multiple n'est disponible qu'à travers la signature des méthodes. En effet, une interface est un type défini permettant de spécifier les méthodes par leur signature ou leur prototype mais sans toutefois les implémenter.

Membre données

Ils représentent l'ensemble des variables définies dans la classe et celles héritées éventuellement de la classe mère. A chaque variable est associé un type permettant de déterminer l'ensemble des opérations pouvant être réalisées sur celle-ci. Cependant il existe plusieurs types de membres données :

- Les variables d'instances (encore appelées variables d'objet). Pour chaque variable d'une classe, il existe une valeur pour chaque instance.
- Les variables de classes contiennent une valeur commune à toutes les instances ou tous les objets d'une classe.
- Les constantes, quant à elles, sont des variables dont la valeur est non modifiable une fois l'initialisation de la classe terminée.

Membre méthodes

Les méthodes implémentent le corps métier de la classe. Elles peuvent être :

- des constructeurs permettant d'initialiser les membres données de la classe lors de son allocation ;
- des méthodes d'instances permettant de décrire le comportement d'une instance de la classe ;
- des méthodes de classes (méthodes statiques) permettant de décrire le comportement propre à la classe. Ces opérations s'appliquent uniquement sur les variables de classe.

Une méthode est généralement définie par sa signature de méthode et son code. Une signature de méthode contient les éléments suivants :

- la classe à laquelle la méthode appartient ;
- le paquetage de cette classe ;
- les paramètres d'entrées et leur type ;
- le type de retour de la méthode.

Une méthode Java, contrairement à une méthode native, a un code de méthode écrit en Java. Le code de la méthode est constitué de variables locales et d'un ensemble de blocs de code. Un bloc de code étant une suite d'instructions délimitée par une accolade ouvrante et une accolade fermante.

On peut avoir deux types de bloc de code :

- un bloc de code de classe définit en dehors de toute méthode mais précédé du mot clé *static* ou alors définit dans une méthode de classe ;
- un bloc de code d'instance.

Les modificateurs

Il s'agit d'autres attributs pouvant être associés aux membres données et membres méthodes d'une classe. On peut citer par exemple :

– *public*, *protected*, *private* : permettant de déclarer le type d'accès à une variable ou une méthode ;
– *static* permettant de définir les variables et méthodes de classe ;
– *final* : empêche la surcharge d'une méthode ou la modification d'une variable après initialisation.

2.1.3 Structure d'un fichier classe

Après compilation du fichier Java, on obtient un fichier de byte code Java appelé fichier classe (*classfile*) par opposition au fichier Java qui contient les instructions dans un langage de haut niveau.

Le fichier classe représente le format portable et exécutable d'une classe sur toute machine virtuelle Java indépendamment de la plateforme matérielle sur laquelle la VM s'exécute. Le format d'un *classfile* représente un fichier Java avec des métadonnées supplémentaires. Il est constitué de dix sections de taille variable, structuré comme illustré dans la figure 2.1.

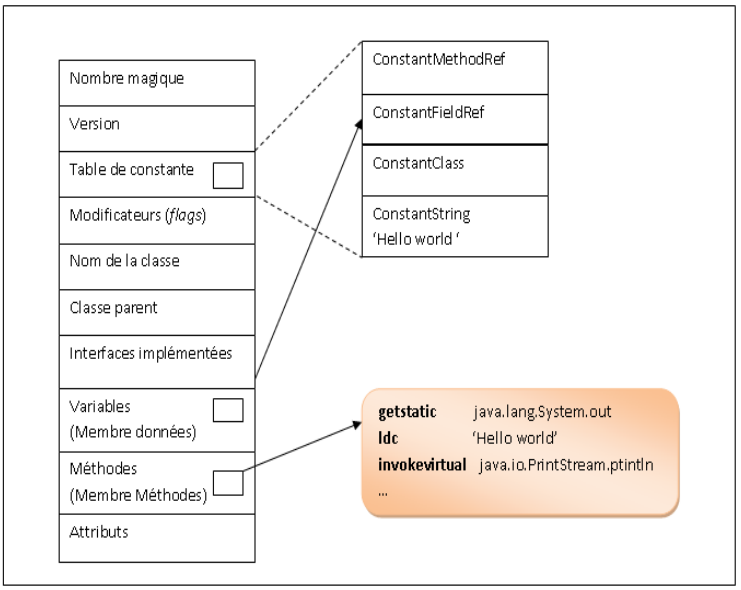

Fig. 2.1 – Format d'un fichier classe

Nombre magique
C'est un nombre écrit en début d'un fichier permettant d'indiquer le format du fichier et dans certain cas, il peut plutôt s'agir d'une signature de fichier. Dans le cas d'un fichier classe, ce nombre est représenté par une valeur en hexadécimale : ***0xcafebabe***. Ce nombre permet lors de la vérification du contenu de la classe de déterminer s'il s'agit d'un fichier classe ou pas.

Version
Elle indique le numéro de version de la VM, et permet ainsi à une VM de vérifier si elle est compatible avec le fichier classe à exécuter. Les valeurs de versions peuvent être 45 (jdk 1.1), 46 (jdk 1.2), 47 (jdk 1.3), 48 (jdk 1.5), 49 (j2se 5.0), 50 (j2se 5.0).

Table de constantes
La table de constante décrit toutes les constantes (chaîne de caractères) utilisées dans la classe. Il

peut s'agir d'un chiffre, nombre, nom des packages, nom des méthodes, nom des variables, nom des interfaces, types des variables, valeurs d'initialisation, signature de méthode, dimension de tableau, description des types complexes, etc.

Modificateur, nom de classe, classe parent
Dans le fichier classe, les drapeaux de la classe ou droits d'accès (*protected*, *public*, *abstract*) sont représentés à travers des masques de bits. Le nom de la classe est fournit en utilisant le caractère / comme séparateur pour décrire le chemin d'accès à la classe. La classe parent est précisée dans le cas d'héritage sinon la classe par défaut est la classe *Object* (*java.lang.Object*) dont la classe parent correspond à la valeur 0.

Interfaces implémentées
La mise en place des héritages multiples se fait à travers les interfaces. Une classe peut ainsi implémenter plusieurs interfaces. Toutes ces interfaces sont répertoriées dans la section interface du fichier classe, dont les valeurs sont des index vers des entrées de la table de constante de la classe.

Variables et méthodes
Les membres données et méthodes d'une classe sont décrits par deux tables. Et chaque description comprend les informations suivantes :
- les drapeaux ou droits d'accès ;
- l'identifiant : nom de la variable ou de la méthode, il s'agit précisément de l'index de l'entrée correspondante dans la table de constante. Cette entrée contient un pointeur vers le code de la méthode ;
- type : signature de la méthode ou type de la variable, pointeur vers la table de constante.

Attributs
Il s'agit des attributs de la classe : le paquetage dans lequel elle se trouve, ses modificateurs et droits d'accès, ses types (classe virtuelle, interface, classe fille, classe abstraite, etc.), sa taille, etc.

La section suivante présente Java Card, qui est une version de Java destinée à la programmation de cartes à puce. Toutes les cartes à puces sont caractérisées par les ressources très limitées disponibles pour l'exécution d'applications. Il a donc fallu développer un langage qui soit à la fois fiable, robuste et peu gourmand en ressources. La solution proposée par Sun Microsystems Inc en 1996 a été la technologie Java Card.

2.2 Java Card

Jusqu'à très récemment la programmation des cartes à puce était réalisée en assembleur et en C, et longuement vérifiée avant la phase de masquage. La phase de masquage est une phase durant laquelle le système d'exploitation de la carte est figé lors de la fabrication dans la ROM[11] de la carte sur le silicium. Le programme contenait à la fois le système opératoire et l'application.

Le processus de développement était donc long, ceci dû à la fois au langage de programmation et à la campagne de test associée au développement. Dans un tel schéma, toute évolution du code est difficile voire impossible. Ce processus de développement long et coûteux était inadapté à certains marchés nécessitant un temps de déploiement (*time-to-market*) réduit.

[11] Read Only Memory

Depuis quelques années, sont apparues de nouvelles cartes à puce pouvant répondre à ces marchés. Elles sont basées sur des systèmes ouverts et autorisent le chargement dynamique de code. Généralement ces cartes permettent le chargement de plusieurs applications sur la même carte. Les cartes basées sur la norme Java Card [Myc03a, Myc03b, Myc09a], MultOS [Mul09] ou bien DotNet Card [Car08a] font partie de cette nouvelle génération.

Ces systèmes opératoires utilisent des machines virtuelles afin d'améliorer la portabilité, la compacité du code et la sécurité. Les cartes ouvertes offrent une couche standard de services commune à une ou plusieurs applications. Par exemple, le standard Java Card permet aux développeurs de bénéficier des trois fonctionnalités suivantes :

- Le développeur d'applications ne tient plus nécessairement compte du composant électronique dans la carte. Son application est donc théoriquement portable sur n'importe quelle carte respectant le standard. Dans la pratique, à cause de l'hétérogénéité des plates-formes, il faut que le développeur n'utilise aucune librairie spécifique à la carte utilisée, et que la carte choisie dispose de suffisamment de ressources matérielles pour supporter l'application (taille mémoire suffisante pour installer l'application et pour qu'elle puisse fonctionner).
- Il est possible d'embarquer plusieurs applications dans une même carte. Ces cartes peuvent donc être qualifiées de cartes multi-applicatives. Les applications embarquées peuvent communiquer ensemble, ou être «complètement» indépendantes. Le partage des ressources communes est dans ce cas limité au minimum nécessaire à travers une interface de partage spécifique à Java Card.
- Une carte peut éventuellement rester ouverte durant tout son cycle de vie, c'est-à-dire qu'elle peut dans ce cas charger dynamiquement une nouvelle application, même après son déploiement. Cette dernière fonctionnalité peut être perçue comme une révolution en même temps qu'un challenge. La révolution tient au fait qu'il est possible, à tout moment, de mettre à jour les fonctions des applications, autorisant ainsi une maintenance non réalisée jusqu'alors. Le challenge tient au fait qu'il faut gérer la configuration des cartes qui sont, dans bon nombre de cas, personnalisées pour chaque utilisateur final de la carte.

La carte à puce est un outil de sécurité et prône en tant que tel une fermeture. Il faut, en plus de proposer de nouvelles fonctionnalités et de nouveaux services, continuer d'assurer la même sécurité qu'auparavant, c'est-à-dire assurer l'intégrité, la confidentialité et la disponibilité des données et des applications de la carte.

De nouveaux mécanismes de sécurité doivent être mis en place afin de garantir la sécurité de la carte et lui permettre ainsi de préserver ses qualités de sécurité et de fiabilité dans les futures applications. L'une des implémentations des cartes ouvertes est Java Card qui représente la quasi totalité des cartes ouvertes en production actuellement.

2.2.1 La spécification Java Card

La plateforme Java Card permet de bénéficier d'un sous-ensemble de la technologie Java afin de pouvoir développer des applications pour carte à puce ou pour des équipements à fortes contraintes de ressources (mémoire, temps processeur, puissance de calcul, etc.).

Dans la version minimale de Java Card 3.0 (*classic edition*), la machine virtuelle exécute des applets

chargés dans un format de fichier appelé CAP (*Converted APplet*). D'autre part, la communication avec le lecteur est effectuée grâce au protocole APDU[12] [ISO97a, ISO97b].

La version connectée de Java Card 3.0 [Myc09b,Myc09c,Myc09d], possède un serveur web embarqué avec la possibilité de charger les applications web Java Card grâce à des fichiers au format JAR[13] tout en utilisant le protocole HTTP[14] pour la communication.

Java Card peut être considérée comme une plateforme fournissant un environnement sécurisé pour carte à puce, interopérable et multi-applicatif qui possède les avantages du langage Java. Cette dernière version (*Connected Edition*) est celle utilisée comme support de cette thèse bien que le travail puisse être adapté sur les autres éditions. Dans la suite, nous décrirons l'édition connectée de Java Card 3.0 : l'architecture, le langage et les mécanismes de sécurité.

2.2.2 L'architecture

Une carte à puce de type Java Card est constituée principalement de trois éléments :

- L'API : elle décrit les paquetages permettant d'utiliser un sous-ensemble du langage Java, d'utiliser les modules fournis par l'environnement Java Card et de faire appel aux fonctions de sécurité accessibles par les applets.
- La machine virtuelle Java Card : elle est en charge de l'interprétation des byte codes, tout en offrant les mécanismes de sécurité.
- L'environnent d'exécution : il intègre les mécanismes comme le chargement de classe, la gestion de la mémoire, le cycle de vie des applications etc.

Quelle que soit l'implémentation utilisée, elle doit au minimum implémenter ces trois spécifications. Bien entendu, il est possible d'introduire des mécanismes complémentaires, ainsi que des APIs complémentaires pour faciliter le développement des applications embarquées.

D'un point de vue global, une Java Card peut être représentée comme un système comportant deux zones différentes.

- La première est figée : elle est constituée du système d'exploitation et des extensions éventuellement fournies par les fournisseurs de la carte. Elle contient le JCRE[15] et le système d'exploitation sous-jacent.
- La deuxième est modifiable. Elle est constituée des applets chargées dans la carte.
  Il est impossible actuellement de modifier une partie du système d'exploitation. Cependant l'arrivée de la technologie Flash pour la mémoire en remplacement de la ROM rend possible si des mécanismes existent pour modifier après fabrication le système d'exploitation, le JCRE et les différentes APIs.

[12] Application Protocol Data Unit
[13] Java ARchive
[14] Hyper Text Transfer Protocol
[15] Java Card Runtime Execution

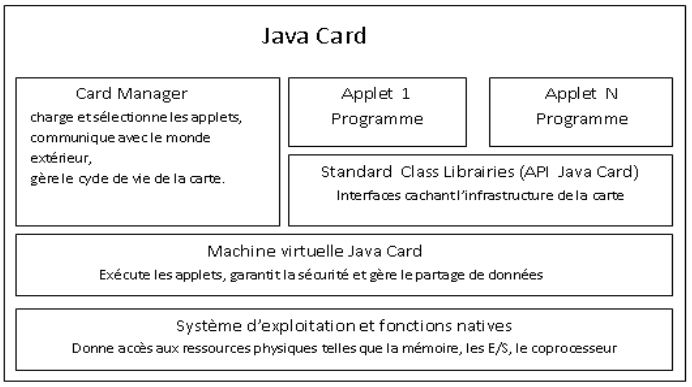

Fig. 2.2 – Architecture Java Card

L'application se situe généralement dans la zone modifiable, et ne communique qu'au travers du JCRE. En conséquence, la seule connaissance de l'interface du JCRE doit suffir pour permettre de développer des applets. L'édition connectée, contrairement à l'édition classique, introduit dans son architecture (voir) une nouvelle machine virtuelle ainsi qu'un nouvel environnement d'exécution qui supporte trois modèles d'exécution :

- Le modèle d'applet classique est basé sur la version précédente de Java Card avec un modèle de communication basé sur le protocole APDU.
- Le modèle d'applet étendu basé sur la version précédente de Java Card mais avec la gestion du multithreading, du format de fichier classe, la gestion des chaines de caractères, etc. Ce modèle est basé sur le protocole de communication APDU.
- Le modèle d'application web est basé sur la construction de servlets. Ce modèle est basé sur le protocole HTTPS pour la communication avec la carte.

2.2.3 Langage Java Card

Le langage Java Card est un sous ensemble de Java conçu pour des environnements aux ressources mémoires et processeur limitées notamment la carte à puce. Ce langage étant un sous ensemble de Java, il existe donc des caractéristiques Java supportées et non supportées. Parmi les caractéristiques supportées, se trouvent la gestion des paquetages, la création dynamique d'objets, les méthodes virtuelles, les interfaces, les exceptions, l'importation statique, les annotations, les types simples (booléen, *byte*, *short* et entier), les classes *Object* et *Throwable*. De plus, pour l'édition connectée, la gestion du multithreading, la destruction automatique des objets (*Garbage collector*) et le support natif du fichier classe avec l'édition des liens se faisant dans la carte ont été ajoutées.

En ce qui concerne Java Card 3.0 édition connectée, il existe évidement des caractéristiques Java non supportées, on peut citer :

- la non prise en charge des nombres à virgules flottante (type *float*, *double*) ;
- pas de chargeur de classe (*classloader*) définit par l'utilisateur ;
- pas de gestion de groupe de threads ou des threads démons ;
- pas de finalisation des instances de classes ;

– pas de gestion d'exception asynchrone (ceci dû à une limitation de la gestion des erreurs sur la plateforme).

En effet, en dehors des erreurs définies par la spécification, l'exécution du programme doit, soit s'arrêter, soit renvoyer un code d'erreur proche de la super classe de l'erreur.

2.2.4 Sécurité

Il existe plusieurs mécanismes de sécurité fournis par Java Card 3.0 qui peuvent être classifiés en deux catégories principales : la sécurité fournie par le langage Java et la sécurité fournie par la plateforme.

Le langage Java Card intègre tous les mécanismes de sécurité intrinsèque au langage de programmation Java tels que :

– Le typage : Java est fortement typé permettant ainsi d'éviter les codes frauduleux.
– Les pointeurs ou références : Java interdit toute opération arithmétique sur les pointeurs.
– La conversion de type : Java ne permet pas de conversion de type non prévue. En effet, toute conversion de type est bien encadrée avec les concepts de polymorphisme, d'encapsulation et d'héritage.

La sécurité sur Java Card 3.0 est aussi fournie par la plateforme. En effet, elle possède plusieurs mécanismes de sécurité, on peut citer entre autres :

1. Le vérifieur de byte code. Le processus de vérification a lieu durant le chargement de l'application et consiste en une interprétation abstraite du byte code (impliquant la vérification du type d'objets manipulés et le contenu de la pile). Avec la nouvelle version de la plateforme Java Card, cette vérification est implémentée grâce à la vérification de type basée sur l'attribut *StackMapTable* [Bod06]. Cet attribut de fichier contient zéro ou plusieurs états du type des variables locales et des données contenues dans la pile des opérandes aux points de jonctions (point de transfert des flots de contrôles). Chaque méthode possède son propre attribut *StackMapTable*. Cet attribut est généralement ajouté au fichier classe par le compilateur et est utilisé par le vérifieur lors du chargement de l'application afin de valider le code de l'application.
2. Le pare-feu (*firewall*). Il partitionne le système en espaces protégés appelés contexte de groupe. Un contexte de groupe est relatif à un paquetage Java. A chaque applet est associé un AID (*Applet IDentification*) permettant de récupérer le nom du package dans lequel il est défini. Ainsi deux applets appartiennent au même contexte de groupe si elles appartiennent au même package. Les objets sont associés au contexte de groupe de l'applet qui l'a crée. Le pare-feu sert donc à isoler les contextes de telle sorte qu'une méthode s'exécutant dans un contexte ne puisse pas accéder aux objets d'un autre contexte. Le partage d'objets entre contextes est effectué grâce à l'interface *Shareable* permettant d'obtenir des objets dit « *Shareable* » et donc partageables entre différents contextes.
3. La gestion des autorisations. Elle est effectuée grâce à un fichier définissant les permissions (fichier *policy*). Ce fichier ajouté lors du chargement de l'application permet de déterminer les autorisations pour l'application lors de l'exécution.

4. L'isolation de code et le contrôle d'accès. Il permet d'assurer que le code de l'application chargée n'interférera pas avec le code d'une application existante ou à venir.
5. L'authentification. A chaque utilisateur authentifié sur la carte est associé un ensemble de droits ou de rôles. Un rôle est relatif à l'accès aux ressources protégées telles que les services basés sur les SIO[16] ainsi qu'aux ressources webs. Sur la plateforme Java Card, on distingue deux types d'utilisateurs : le propriétaire de la carte et les autres (administrateur, etc.). L'authentification est effectuée grâce à des identificateurs pouvant être un mot de passe, un code PIN (*Personnal Identification Number*), une information biométrique, etc.
6. Les communications sécurisées. Ceci passe par l'utilisation d'un tunnel sécurisé entre le client et le serveur grâce au protocole TLS (*Transfer Layer Security*) pour limiter les écoutes et assurer la protection des données envoyées.
7. Les annotations de sécurité. La spécification permet d'annoter une classe ou une méthode afin de spécifier les classes et méthodes sensibles de l'application.

2.3 Mise à jour dynamique

Le problème de mise à jour dynamique est un sujet sur lequel de nombreuses recherches ont été effectuées. L'objectif général étant de pouvoir mettre à jour des applications en cours d'exécution sans stopper l'application et sans arrêter la plateforme sur laquelle elles s'exécutent. En effet, dans le but de fixer des défauts, d'améliorer et/ou de supprimer certaines fonctionnalités, les applications sont appelées à évoluer ou, plus exactement à être mises à jour. Il en est de même pour les applications pour cartes à puce.

Le problème de mise à jour dynamique consiste en quatre tâches principales :

1. Détermination du moment opportun durant lequel la mise à jour peut être réalisée en conservant l'intégrité et la cohérence du système ;
2. Migration du contexte de l'ancienne version vers la nouvelle version, ceci passe par
   - l'interruption temporaire d'exécution de l'ancienne version ;
   - l'exécution des fonctions de transformation pour l'obtention du nouveau contexte d'exécution correspondant à la nouvelle version ;
     - Mise à jour du code ;
     - Mise à jour des données ;
3. Restauration du contexte (*Roll-back*) si la mise à jour a échoué.
4. Poursuite de l'exécution en utilisant le nouveau composant et le nouveau contexte.

A côté de ces tâches, s'ajoute le problème de vérification, de validation du processus de mise à jour et le problème de sécurité.
(1) Validité des modifications effectuées ;
(2) Préservation des propriétés sémantiques de l'application ;
(3) Détection des nouvelles failles de sécurité apportées par le système de mise à jour ;
(4) Proposition de solution à ces nouvelles failles.

[16] Shareable Interface Objects

Un composant système ou une application peut être constitué d'un ensemble de classes Java et/ou natifs. Nos travaux s'intéressent aux composants systèmes et applications écrites en Java, et notre objectif est de remplacer un composant ou une application par un autre avec un niveau de granularité qui est la classe.

2.4 Mise à jour dynamique de programmes Java

Dans la plupart des systèmes existants, le niveau de granularité de mise à jour dynamique est la classe Java et la mise à jour est effectuée selon différents aspects de modification.

1. **L'aspect modification du code**
   La mise à jour du code dépend des modifications effectuées sur les fragments du code de la classe. Ces modifications de code peuvent être :
   - la modification des instructions d'une méthode,
   - la modification des champs d'une classe,
   - la modification de la liste des paramètres d'une méthode,
   - la suppression ou l'ajout d'une méthode, d'un champ, d'un constructeur, d'une exception, etc.

   Au cours du processus de mise à jour dynamique du code, certaines considérations peuvent être effectuées, par exemple une modification d'une instruction peut être considérée comme une suppression suivie d'un ajout de celle-ci.

2. **L'aspect modification des données**
   Il s'agit des aspects techniques nécessaires pour changer l'état de l'application en cours d'exécution vers l'état correspondant à la nouvelle version. Généralement, il est quasiment impossible de commencer la nouvelle version à partir de son état initial, par conséquent l'état en cours doit être transformé pour obtenir un état correspondant à la nouvelle version avec lequel le processus peut continuer sans incohérence du système.
   L'état du système peut être composé de :
   - Les threads courants en cours d'exécution ;
   - Les données des méthodes dans la pile de thread (variables locales, compteurs de programmes, etc) ;
   - Les instances dans le tas de la machine virtuelle ;
   - Et les autres structures de données de la VM impactées par la mise à jour.

   La mise à jour des données consiste donc à mettre à jour les instances dans le tas de la machine virtuelle, les frames dans la pile Java et les autres structures de données nécessaires de la VM. Elle concerne donc la transformation de l'état du système au niveau de tout ce qui ne touche pas le code en lui même.

3. **L'aspect activité**
   Il s'agit de pouvoir déterminer le moment opportun où la mise à jour peut être effectuée en conservant la cohérence du système, ceci dû particulièrement au fait que les méthodes en cours d'exécution doivent aussi être remplacées (mais à quel moment ?, et comment ?). Ces questions et les réponses à celles-ci sont développées en détail dans le chapitre suivant.

4. **L'aspect restauration de contexte (*Roll-back*)**
   En cas de non-atomicité de la mise à jour, ou en cas d'erreur durant la mise à jour, il est nécessaire de pouvoir annuler la mise à jour et surtout de pouvoir restaurer l'ancien contexte d'exécution. En fonction de l'avancée du processus de mise à jour, la restauration peut concerner :
   – Le code (méthodes, signature des champs, paramètres de méthodes, etc.)
   – Les données (les instances dans le tas de la VM, les références dans les frames présentes dans la pile Java, etc.)

2.5 Mise à jour dynamique de programmes Java Card

Dans Java Card 3.0, une application possède un cycle de vie bien définit. Initialement, un module d'application est chargé sur la carte pour être stocké en mémoire persistante. Les services partagés sont ensuite publiés pour devenir accessibles par les autres modules d'applications autorisés. Lorsqu'un service est demandé, une instance de l'application associée est créée. Si une demande explicite de destruction est émise, l'instance de l'application peut être supprimée. Enfin, une application peut être tout simplement déchargée de la carte.

On peut donc résumer le cycle de vie d'une application comme suit :
(1) Chargée ;
(2) Instanciée ou active ;
(3) Selectionnable ;
(4) Supprimée (suppression de l'instance) ;
(5) Déchargée (suppression de l'application).

Dans le système de mise à jour *post-issuance*, on remarque plusieurs défauts dû à ce cycle de vie. Premièrement, la mise à jour d'une application ou d'un module d'une application Java Card passe par un retrait puis un chargement de la nouvelle version de l'application. Cependant, le retrait suppose l'arrêt du service offert par cette application. Deuxièmement, si le service est partagé ou publié dans Java Card, cela suppose un arrêt des autres applications utilisant ce service.

La mise à jour *post-issuance* ne tient pas compte du côté dynamique de l'application. En plus, en ce qui concerne Java Card, la machine virtuelle Java ne s'arrête jamais puisque les objets persistants sont préservés entre les sessions de communication avec le lecteur de carte. Avec le processus de mise à jour post-issuance, la mise à jour d'un module de la machine virtuelle Java Card est impossible pour le moment. L'unique solution est de graver une nouvelle image de la machine virtuelle sur une nouvelle carte. Cette thèse permet donc de proposer une alternative à la création d'une nouvelle carte en proposant une approche de mise à jour dynamique.

La problématique de DSU sur les cartes à puce peut se résumer en quelques questions à savoir :

Comment réduire le temps de transfert et l'occupation mémoire des entrées nécessaires à la DSU ? Une fois, les entrées nécessaires à la DSU chargées dans la carte :

– Comment effectuer le transfert d'état d'exécution de l'ancienne version de l'application pour obtenir un nouvel état d'exécution correspondant à la nouvelle version ?

– A quel moment effectuer le transfert d'état d'exécution en conservant la cohérence du système ? Comment déterminer les moments opportuns du système pour la mise à jour ?

- Comment mettre à jour le code de l'application avec un coût minimal ?
- Comment effectuer le *roll-back* sur la carte si le processus de mise à jour échoue ?
- Comment s'assurer de la validité de cette DSU ?
- Est ce que le nouveau code et le nouvel état d'exécution de l'application correspondent respectivement au code et à l'état d'exécution de l'application si on avait chargé directement cette nouvelle application en passant par une mise à jour traditionnelle ?

Comment mettre en place un tel système en tenant compte des fortes contraintes de ressources des cartes à puce, en minimisant l'utilisation mémoire, en minimisant la consommation énergétique (cas des cartes SIMs[17] par exemple), et bien sûr en minimisant l'utilisation du temps processeur ? Les réponses à la plupart de ces questions ont donné naissance à un système appelé EmbedDSU de mise à jour dynamique pour cartes à puce Java que nous présentons dans cette thèse.

2.6 Conclusion

Actuellement, la mise à jour dynamique des applications dans les cartes à puce Java est effectuée par l'opération de *post-issuance*. Nous avons constaté que cette opération non seulement ne prenait pas en compte la dynamicité des applications mais aussi ne permettait pas la mise à jour des modules systèmes (par exemple un module de la machine virtuelle Java Card). De plus, si un bogue est détecté dans un module système, l'unique solution est de retourner toutes les cartes à l'émetteur pour en obtenir de nouvelles, ce qui est bien sûr très coûteux. Cette thèse propose une solution à ces problèmes en proposant une solution de mise à jour dynamique malgré les fortes contraintes de ressources de la carte. Le chapitre suivant présente les problèmes scientifiques liés à la mise à jour dynamique, les solutions proposées dans la littérature et une analyse de ces solutions dans le cadre de la carte à puce.

[17] Subscriber Identity Module

Chapitre 3

Etat de l'art

Sommaire

De nombreuses recherches ont abordé le problème de la mise à jour dynamique dans divers contextes partant des applications autonomes (*stand alone*) aux serveurs d'applications dans des systèmes distribués, répartis, etc. Cependant, ces systèmes de mise à jour dynamique ne sont généralement pas destinés au domaine de l'embarqué (ils ciblent plutôt les postes de travail).

Dans le domaine de l'embarqué, notamment celui de Java Card, aucun système de DSU n'existe pour le moment. Au regard des contraintes de ressources et de sécurité des cartes à puce, il n'est pas toujours possible d'adapter les approches des solutions existantes. Dès lors, il s'agira de proposer une solution de DSU innovante et adaptée aux cartes à puce Java. L'objectif de ce chapitre est de

recenser, de synthétiser et d'analyser plusieurs travaux liés au problème de la mise à jour dynamique en limitant l'analyse aux systèmes de DSU basés sur les langages Java et C/C++.

Ce chapitre présente premièrement les problèmes scientifiques liés à la mise à jour dynamique. Ensuite, il décrit les différentes solutions à ces problèmes et les différents systèmes de DSU présents dans la littérature. Enfin, il analyse la manière par laquelle ces solutions peuvent être appliquées au domaine de la carte à puce.

3.1 Problèmes scientifiques liés à la DSU

La mise à jour dynamique nous permet de passer d'un état d'exécution d'une application vers un nouvel état correspondant à la nouvelle version, ceci sans interrompre le fonctionnement (sans perte d'état). Il s'agit principalement de pouvoir mettre à jour le code, les données et les autres structures de données associées au contexte d'exécution de l'application et d'exécuter tout ceci de façon homogène tout en préservant la cohérence du système.

3.1.1 Mise à jour du code

Le problème de mise à jour du code réside principalement dans la modification du code binaire embarqué et accessoirement dans la gestion des versions du code de l'application à mettre à jour. La modification du code implique soit de réécrire ailleurs en mémoire et de modifier toutes les références à ce code, soit de modifier le code original avec une indirection vers le nouveau code.

3.1.2 Mise à jour des données

Il est difficile de parler des mises à jour dynamiques sans considérer le problème de transfert d'état des données. De nombreux problèmes sont à résoudre, notamment :

- la coexistence des données de l'ancienne et de la nouvelle version après la mise à jour ;
- la méthode de transfert des anciennes instances vers les nouvelles ;
- la préservation du système de type des objets après la mise à jour ;
- la validité de la représentation des données en mémoire correspondant à la nouvelle version de l'application, etc.

Pour garantir la cohérence de l'état d'exécution après la mise à jour des données, il est nécessaire de déterminer au préalable le moment opportun du système durant lequel ce transfert peut être effectué. Il s'agit donc de déterminer un point sûr du système.

3.1.3 Détermination des points sûrs

Cette section commence par définir les notions de méthodes actives, méthodes non actives, méthodes restreintes et méthodes non restreintes. Ces définitions sont relatives aux méthodes de la classe à mettre à jour et à leur état d'exécution durant la mise à jour.

Définitions

Une méthode est dite **active** si elle est en cours d'exécution dans la machine virtuelle, donc c'est une méthode possédant une frame dans une pile Java de la VM.

Une méthode est dite **non active** si elle n'est pas en cours d'exécution dans la VM donc si elle ne possède pas de frame dans la pile Java.

Une méthode est dite **restreinte** si elle est une méthode modifiée de la classe à mettre à jour et si elle est active dans la VM. Ainsi, c'est une méthode modifiée et en cours d'exécution dans la VM lors de la mise à jour.

Une méthode est dite **non restreinte** si elle n'est pas une méthode modifiée de la classe à mettre à jour. Elle peut être active ou pas dans la VM. Donc c'est une méthode non modifiée pouvant être en cours d'exécution ou pas lors de la mise à jour.

Un point dit sûr pour une classe donnée, est obtenu lorsqu'on n'a plus de méthodes restreintes dans la pile Java lors de la mise à jour de cette classe.

Problématique de point sûr

Appliquer la mise à jour à un moment inopportun peut laisser le système dans un état incohérent voir même peut provoquer une défaillance du système en dépit de la cohérence du système de type. Ceci peut arriver par exemple, si après la mise à jour, une méthode restreinte :

1. accède à un champ qui a été supprimé ou dont le type a été modifié ;

2. fait appel à une méthode dont la signature a été modifiée ou une méthode qui a été supprimée.

A coté de ces problèmes scientifiques, à savoir la mise à jour du code, la mise à jour des données et la recherche du point sûr, s'ajoute un autre problème qui est celui de la restauration du contexte. C'est un problème qui est très peu traité dans la littérature voir quasiment inexistant dans les systèmes de DSU actuels. Cependant, dans le monde de la carte, au vu des contraintes de sécurité et d'espace mémoire, il est nécessaire de prévoir un mécanisme de restauration du contexte d'exécution en cas d'échec ou de détection d'opération illicite lors du processus de DSU.

3.1.4 Préservation du système de type

Lors de la mise à jour, il est impératif de s'assurer que le système de type est préservé. Si la classe C à modifier hérite de B, il faut garantir que la mise à jour ne modifie pas l'arbre d'héritage. Si la classe C hérite d'une autre branche, potentiellement il devient possible de lire et écrire des fragments mémoire appartenant à d'autres objets voir d'autres contextes de sécurité ou pire de pouvoir modifier des éléments du tas statique : c'est-à-dire avoir la possibilité de générer du code auto-modifiable. La garantie de cohérence du système de type est primordiale pour garantir la sécurité de la mise à jour.

3.2 Différentes approches de solution

3.2.1 Mise à jour du code

Une des fonctionnalités principales d'un système de DSU est la mise à jour du code. Cette fonctionnalité est traitée différemment selon que l'on soit dans un système distribué ou dans un environnement autonome (textitstandalone) avec des applications en C/C++ ou en Java, etc. De nombreuses techniques existent pour la mise à jour du code.

Coexistence des versions

Il s'agit de faire coexister dans le système, les versions des applications mises à jour. Dans ce cas, il est nécessaire de bien gérer les numéros de version de l'application et les espaces de nommage. Pour cela, chaque appel est intercepté dans le but de déterminer la version à laquelle appartient l'objet correspondant et ainsi accéder à l'espace de nommage relatif de la version de l'application en cours d'exécution.

Fonction d'indirection

Les systèmes de mise à jour dynamique pour les applications écrites en langage C/C++ utilisent un système d'indirection pour chaque méthode ou fonction mise à jour comme illustré par la figure 3.1. Chaque appel de fonction passe par une table d'indirection dont les entrées ont été mises à jour afin de pointer sur la nouvelle version de la fonction. Ainsi, pour chaque appel de méthode relative à la nouvelle version de l'application, une recherche dans la table d'indirection est effectuée.

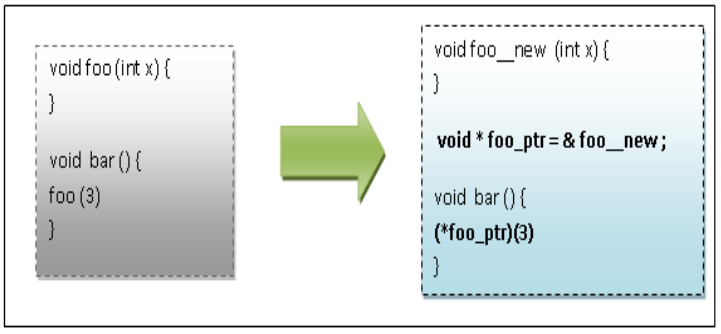

Fig. 3.1 – Exemple de fonction d'indirection

Technique du saut

La technique du saut consiste à remplacer les premières instructions de code d'une méthode modifiée par un appel vers la nouvelle version de la méthode. Ainsi, tous les appels de cette méthode pointent toujours vers l'ancien code qui lui, fait appel à la nouvelle version de la méthode.

Une variante de la technique du saut, est d'effectuer une réécriture dynamique du code et une recompilation des parties du code afin de rediriger les appels vers la nouvelle version du code.

3.2.2 Mise à jour des données

Coexistence des données

Il existe deux types de modèle de mise à jour des données :
(1) des modèles où les anciennes versions des données et/ou des méthodes peuvent coexister après la mise à jour. Dans ces cas, l'accès à une version d'un objet est défini par le type de la méthode qui y accède. Ainsi un nouvel objet sera accessible par le nouveau code et vice versa ;
(2) des modèles, où après la mise à jour, seules les données correspondantes à la nouvelle version sont présentes dans le système. Ainsi, celles correspondantes à l'ancienne version sont transformées pour qu'elles correspondent à la nouvelle version.

Cette transformation peut être « paresseuse » ou non.

Elle est dite paresseuse si les données sont transformées uniquement lors du premier accès à celles-ci après la mise à jour de l'application (soit durant l'exécution de la nouvelle version de l'application).

Elle est dite non paresseuse si toutes les données sont transformées au cours de la mise à jour.

Transformation des données

De nombreuses techniques existent pour obtenir les nouvelles données ou pour pouvoir les obtenir à partir des anciennes. Cependant la plupart des systèmes de DSU utilisent les fonctions de transfert afin de satisfaire la sémantique de la mise à jour.

Ils existent deux type de fonctions de transfert :
(1) les fonctions de transfert fournies par le programmeur, basées sur la sémantique de l'ancienne version de l'application ;
(2) les fonctions de transfert générées automatiquement.
Certains systèmes de DSU fournissent la fonctionnalité de génération de fonction de transfert lors de la mise à jour. Généralement, ces fonctions de transfert générées automatiquement sont limitées à l'initialisation des nouveaux objets. Les valeurs d'initialisation sont obtenues grâce à la spécification du système sur lequel s'exécute l'application. Par exemple, si le système est une machine virtuelle Java, l'on sait que par défaut, les entiers sont initialisés à zéro et les références à NULL. Ainsi, si les membres données d'une classe sont modifiés alors une fonction de transfert peut être générée de façon automatique selon le type du champ ajouté ou modifié. Ce qui permet lors de la création de l'objet, d'initialiser le segment mémoire correspondant au nouveau champ dans la nouvelle instance.

Les fonctions de transfert fournies par le programmeur permettent de spécifier la manière d'obtenir les nouvelles versions de données en fonction des anciennes. Elles peuvent opérer soit au niveau de la pile de threads, soit au niveau du tas de la machine virtuelle, ou au niveau des variables statiques. De nombreuses approches de transfert d'état existent se basant sur deux critères :

1. Comment ?
2. A quel moment ?

Comment ?

Une solution est de permettre aux programmeurs de définir des fonctions de transfert mais uniquement en ce qui concerne les données dont le type a été modifié entre les versions. En effet, en ce qui concerne les variables d'instances, la mise à jour est effectuée grâce aux fonctions de transfert générées automatiquement sauf pour les variables dont le type a changé. Dans ce dernier cas, la transformation de type est appliquée grâce à aux fonctions de transfert fournies par le programmeur.

Une autre approche est de créer un nouvel objet correspondant à la nouvelle version, d'initialiser les champs non modifiés avec les valeurs de l'ancien objet à travers une recopie et d'initialiser les champs ajoutés grâce aux fonctions de transfert générées automatiquement ou fournies par le programmeur.

Paresseux ou non ?

Mode paresseux

De nombreux systèmes utilisent le système de transfert d'état dit paresseux. Celui-ci s'applique après la mise à jour. Pour chaque accès à un objet :
(1) il vérifie si l'objet appartient à une classe modifiée, ou à une application modifiée ;
(2) Si c'est le cas, il vérifie si l'objet correspond à la nouvelle version. Dans le cas contraire, il applique les fonctions de transfert pour mettre à jour l'objet et l'exécution continue.

L'avantage est de ne pas appliquer les fonctions de transfert à tous les objets de la classe ou de l'application mise à jour, mais plutôt aux objets auxquels on a réellement accédé. Cependant, il est nécessaire d'ajouter un champ supplémentaire à l'objet permettant de déterminer le numéro de version de ce dernier. Tout ceci induit un surcoût en espace mémoire et en temps d'exécution durant l'exécution normale de l'application, et ceci à cause du coût de la vérification, pour chaque objet accédé après la mise à jour.

Mode non paresseux

Une alternative à cette solution est d'appliquer les fonctions de transfert à tous les objets à transformer, ceci durant la mise à jour de la classe ou de l'application concernée. Ce qui nécessite d'introspecter le tas et de parcourir l'ensemble des objets afin de déterminer ceux concernés par la transformation.

A quel moment ?

Après avoir répondu à la question comment ?, il est nécessaire de répondre aux questions quand ? à quel moment ? Il s'agit de déterminer les moments durant lesquels la mise à jour (du code et des données) peut être réalisée en conservant la cohérence du système. Ces moments sont appelés des points sûr du système.

3.2.3 Détermination des points sûrs

La technique de méthode restreinte

La détection d'un point sûr ne demande pas que toutes les méthodes actives de la classe finissent leur exécution mais seulement les méthodes restreintes. Donc il ne s'agit pas d'obtenir des piles d'exécution où toutes les méthodes de la classe à mettre à jour ne sont pas actives. Il s'agit plutôt de déterminer s'il n'existe pas de méthodes non restreintes dans les piles de threads.

Ils existe de nombreuses techniques de détection de point sûr autre que la technique de méthode restreinte.

La technique de méthode actives

Ces techniques proposent de mettre à jour des méthodes actives. Cependant, elles doivent prouver :

1. qu'elles ne font pas des appels à des méthodes dont la signature a changé ou à des méthodes supprimées ;

2. qu'elles ne font pas des accès à des champs supprimés ou à des champs d'objets dont le type a été changé.

Dans cette technique, une nouvelle définition de méthode restreinte et non restreinte est proposée :

1. Une méthode restreinte est une méthode modifiée, présente dans la pile de thread et dont la mise à jour ne peut être réalisée qu'après son exécution ;
2. Une méthode non-restreinte est une méthode modifiée, présente dans la pile de thread et dont la mise à jour peut être réalisée sans attendre la fin de son exécution.

Dans cette technique, obtenir un point sûr revient à déterminer le nombre de méthodes restreintes. Ce qui revient à attendre que les méthodes restreintes aient finis leur exécution.

Cette technique réduit le nombre de méthodes restreintes, ce qui permet d'obtenir rapidement un point sûr. Cependant elle ajoute des surcoûts de vérification (pour chaque méthode active de la classe à mettre à jour, parcours des instructions, vérification de chaque appel de méthode, vérification de chaque accès aux champs d'un objet de la classe, etc. sont nécessaires). Ce surcoût de vérification ne saurait être toléré dans la carte au regard des contraintes mémoire et processeur. On peut penser à obtenir ces informations de façon statique à l'extérieur de la carte et à les utiliser à l'intérieur lors de la mise à jour ce qui augmenterait la taille des données à envoyer en on-card et bien sûr ajouterait un surcoût lié à l'interprétation et à la vérification du type de méthode (restreinte ou non).

Pour les méthodes non restreintes, cette technique introduit aussi un nouveau problème, celui de déterminer le PC correspondant dans la nouvelle version. En effet, soit f, une méthode restreinte ayant initialement 6 instructions, la nouvelle méthode f possède 10 instructions ou moins de 6 instructions. Le problème est le suivant : comment déterminer l'instruction où il va falloir continuer l'exécution après la mise à jour du code de f ?

De nombreux travaux se sont penchés sur ce problème et proposent plusieurs solutions entre une solution qui consiste à faire une correspondance entre chaque PC de l'ancienne version en son correspondant dans la nouvelle version.

Dans cette solution, il est possible d'initialiser la frame de la nouvelle version de la méthode à partir de la frame de l'ancienne version de la méthode.

Cependant, la technique de recherche de point sûr basée sur les méthodes restreintes peuvent rencontrer un problème de boucle infinie. En effet, si une méthode restreinte est active et contient une boucle infinie, un point sûr n'est jamais atteint ceci dû au fait que la méthode restreinte ne terminera jamais son exécution. Dans la littérature, il existe au moins deux solutions proposées pour ce problème : la technique d'extraction de la boucle et la technique de reconstruction de la pile de thread.

Extraction de boucle

La technique d'extraction de boucle consiste à associer une autre méthode à l'ensemble d'instructions constituant la boucle. On obtient ainsi au moins deux méthodes f1 et f2 en extrayant les instructions de la boucle dans f1 pour en faire une nouvelle méthode f2. Ainsi si une modification a lieu avant ou après la boucle, cette modification a lieu dans f1 et peut être effectuée. Par contre, si la modification a lieu dans la boucle, c'est-à-dire dans f2, le point sûr ne pourrait jamais être atteint puisque f2 serait une méthode restreinte infinie.

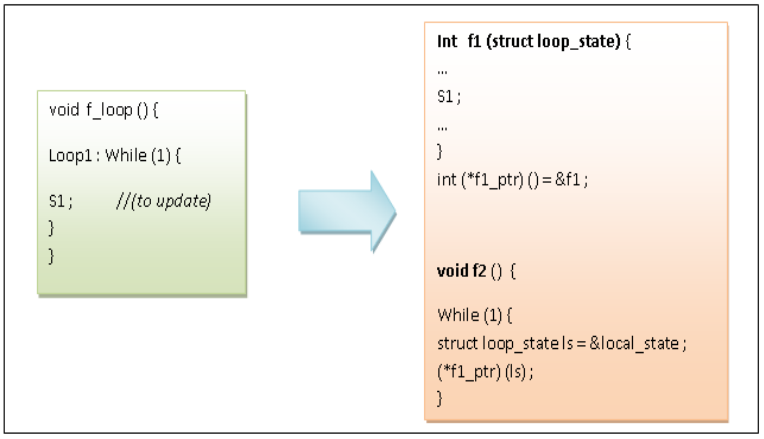

Fig. 3.2 – Exemple extraction de boucle

Reconstruction de piles

La technique de reconstruction de piles nécessite une extraction de l'état de la frame de méthode restreinte et un transfert de l'état de la frame vers une nouvelle version. Ceci se fait en transférant l'exécution vers la bonne instruction dans le code de la nouvelle méthode.

Cette méthode se base sur un autre type de fonction de transfert : les fonctions de transfert de pile de thread fournies par le programmeur. Ceci nécessite au programmeur de comprendre le fonctionnement de la machine virtuelle et de comprendre le fonctionnement de la pile de thread.

Toutefois, que ce soit dans la technique d'extraction de boucle ou la technique de reconstruction de pile, une solution au cas de boucle inter-méthodes n'a pas encore été proposée.

3.2.4 Préservation du système de type

Langage C/C++

Certaines techniques de transfert d'état sont généralement matures et performantes mais souffrent d'un certains nombres de problèmes dus principalement aux restrictions du langage, notamment le fait que le langage C ne préserve pas le système de type (dit *type-unsafe*). La préservation du système de type peut s'effectuer de façon statique à la compilation et/ou de façon dynamique durant l'exécution. Le langage C préserve le système de type dans des contextes limités. Par exemple, une erreur de compilation est générée lorsque l'on tente de convertir un pointeur sur un type de structure en un pointeur vers un autre type de structure, sauf si une conversion explicite est utilisée.

Le langage C fournit des conversions explicites vers des types indéfinis ou non spécifiés. Par exemple, l'allocation de la mémoire est effectuée grâce à la fonction *malloc* qui retourne un pointeur non typé, dont le code appelant doit convertir selon le type de pointeur approprié. Des expressions telles que *(struct bar *) malloc (sizeof (struct bar))* sont généralement rencontrées. Et ces zones mémoires peuvent changer de type durant l'exécution. Ce type d'opération est aussi classée parmi les,opérations non-sûres (ne préservant pas le système de type).

Par contre, le langage C++ préserve mieux le système de type lorsque l'on n'utilise pas les pointeurs sans type (*void*) ni les conversions explicites entre les pointeurs de deux types. Dans ces langages où les types dynamiques associés aux segments mémoires peuvent être totalement différents des types statiques déclarés dans l'application, le transfert d'état qui est un élément essentiel dans le processus de la mise à jour dynamique devient plus compliqué à réaliser. En effet, le transfert d'état nécessite de pouvoir capturer l'état courant et appliquer certaines opérations sur celui-ci pour obtenir le nouvel état correspondant à la nouvelle version de l'application. Ce problème de transfert d'état pour ce type de langage s'observe de façon très explicite dans la migration de thread pour l'équilibrage de charge et la tolérance aux pannes dans les calculs haute performances. Pour résoudre le problème de capture d'état et de transfert d'état, il est nécessaire, entre autres, de pouvoir déterminer pour chaque segment mémoire le type dynamique associé, ce qui parait impossible dans les langages C et C++.

Une solution est de pouvoir identifier les opérations sur les pointeurs et les appels aux fonctions des APIs pouvant causer un problème au niveau du système de type. Pour cela, une approche intra-procédurale peut être envisagée pour détecter les opérations sur les pointeurs ne préservant pas le système de type, puis systématiquement tracer leur propagation. Avec la caractéristique dynamique des applications, il n'est pas toujours possible de prédire l'exécution effective et le chemin d'appel de fonction. Toutes les possibilités ne sont donc pas traitables, d'où la nécessité d'un algorithme heuristique.

Cet algorithme s'applique en deux phases :

1. Au moment de la compilation, un module d'analyse statique de code scanne, identifie toute opération dite non-sûr, et détermine la propagation tout en insérant dans le code des annotations pour spécifier les instructions délicates.

2. Durant l'exécution, un contrôle dynamique est effectué grâce à ces annotations insérées dans le code pour capturer les événements de type non-sûr. Ceci permet de détecter les applications non-sûres et de déterminer si la capture d'état peut être effectuée ou pas.

Une autre solution est de ne pas faire de transfert ou de capture d'état. C'est cette solution qui est envisagée dans certains systèmes de mise à jour dynamique pour les langages C et C++. Les langages C/C++ en plus du fait qu'ils sont non-sûrs ne possèdent pas de ramasse-miettes compliquant ainsi la tache des systèmes de mise à jour dynamique associés. L'absence de machine virtuelle est aussi un désavantage. Par exemple une JIT (*Just In-time Compilation*) basée sur une machine virtuelle peut compiler et recompiler les classes ajoutées ou modifiées sans imposer de coût supplémentaire. La mise à jour du code dans les systèmes de DSU pour langage C/C++, celle-ci passe par l'insertion statique d'indirections ou l'insertion dynamique des sauts pour rediriger les appels des méthodes ou fonctions.

Langage Java

Java est un langage fortement typé, il n'existe pas de possibilité de réaliser un transtypage non souhaité. En effet, le compilateur vérifie lors d'un transtypage que l'opération est valide, s'il ne peut le déterminer statiquement alors une instruction de vérification de type est automatiquement insérée dans le code. Le DSU doit donc garantir que cette propriété de Java est maintenue. Deux éléments peuvent être sujets à erreur, le langage d'expression de la DIFF et le mécanisme dans la carte de mise à jour. Le mécanisme d'extraction de la DIFF doit vérifier qu'une classe C dans la version initiale est bien définie comme héritant de la même classe dans la nouvelle version. Le mécanisme de mise à jour dans la carte doit garantir de ne pas modifier la classe mère lors de la modification.

Dans la suite, nous décrirons plus en détail chaque système de DSU, que ce soit pour des applications C/C++, Java ou pour des systèmes d'exploitation. Ce sont des systèmes de DSU qui ne sont pas utilisés dans le domaine de l'embarqué. En effet, en ce qui concerne l'embarqué notamment les carte à puce Java, il n'existe pas encore de système de DSU dans l'état de l'art.

3.3 État de l'art

Au cours des trente dernières années, une variété d'approches a été proposée pour la mise à jour dynamique des applications. Un grand nombre de systèmes de DSU basés sur un compilateur, une machine virtuelle ou une bibliothèque logicielle ont été développés pour les langages C [FS91, BH00, Hic01, ABBS05], C++ [HG98, BHS+05], Java [MPG+00, ORH02, BLS+03, DE03], et les langages fonctionnels comme ML [DGW97, Dug01] et Erlang [AVWW96].

La majorité de ces systèmes ne supporte pas tous les types de modifications pouvant être effectués dans la pratique. Par exemple, la modification de définition de type ou de prototype de fonction ne peut pas être effectué [HG98, ORH02, DE03, BHS+05, ABBS05], ou bien de telles modifications peuvent être réalisées mais uniquement pour les types abstraits [DGW97, BKA+05].

Dans de nombreux cas, la mise à jour de code actif (présent dans la pile d'exécution durant le processus de mise à jour) n'est pas possible [FS91, Gup94, DGW97, MPG+00, BHS+05], et les

données stockées dans des variables locales ne peuvent pas être mises à jour [FS91, Gup94, HG98, Hic01].

Certaines approches sont intentionnellement moins complètes, celles-ci ciblent les plateformes de développement [GR89, Arc] ou de l'instrumentation dynamique [BH00].

Les sections suivantes présentent quelques travaux pour le langage C, C++ et Java.

3.3.1 DSU dans le domaine C/C++

Il existe plusieurs systèmes de mise à jour dynamique des applications C/C++ qui ciblent particulièrement les systèmes d'exploitation et les applications serveurs.

Ginseng

Ginseng [NHSO06] est un système de mise à jour dynamique pour les applications serveurs en C. Ce système de DSU a été développé dans le cadre des travaux de recherche de l'équipe PLUM[18] du département informatique de l'Université du Maryland aux États-unis. C'est un système qui peut s'exécuter sous Linux et peut actuellement être utilisé pour réaliser une analyse statique du code[19].

C'est un système de DSU flexible qui supporte plusieurs types de mises à jour telles que la signature des méthodes, la définition des structures, les variables globales et les instructions des méthodes.

Mise à jour du code. Pour la mise à jour du code, Ginseng utilise une technique standard de DSU, il s'agit de la technique d'indirection des appels de méthodes pour accéder aux nouvelles versions. Ginseng est composé de trois modules : un compilateur, un générateur de patch, et un module de mise à jour proprement dite. Deux fonctions sont associées au compilateur. Il compile les applications pour qu'elles puissent être dynamiquement mises à jour c'est-à-dire de telle sorte que le code existant puisse être redirigé vers les nouvelles fonctions présentes dans le patch dynamique. Pour les types de données, le code est compilé tout en indiquant les types obsolètes et les fonctions de transfert à appeler lors de la mise à jour.

Mise à jour des données. Ginseng utilise la solution de transfert d'état. Il génère les fonctions de transfert pour les types de donnée à mettre à jour et durant l'exécution, il alloue ou ajoute les espaces supplémentaire requis pour la nouvelle version des objets. Il utilise une technique « paresseuse » pour la mise à jour des objets. En effet, les fonctions de transfert sont appelées uniquement lors du premier accès à l'objet durant l'exécution de l'application mise à jour. Ginseng ne supporte pas la mise à jour des méthodes actives sur la pile d'exécution. Pour garantir le système de type, Ginseng interdit les instructions ou les appels de fonctions de type non-sûr.

Détection de point sûr. Il analyse le code pour s'assurer qu'il n'y ait pas d'opérations dites non-sûr (voir section 3.2.4) et que le système de type sera préservé même après la mise à jour. Deux fonctions sont aussi associées au générateur de patch. Il identifie les modifications effectuées entre les deux versions (au niveau des variables globales, des fonctions, des types, etc.). Ensuite, pour chaque modification détectée, il génère les fonctions de transfert nécessaires à la mise à jour. Le

[18] Programming Languages at University of Maryland
[19] Site web Ginseng : `http://www.cs.umd.edu/projects/PL/dsu/`

module de mise à jour dynamique s'applique à la version de l'application obtenue par le compilateur de Ginseng pour le chargement de celle-ci et l'édition des liens dynamiques après détection d'un point dit sûr du système.

Ksplice

Ksplice [AK09, Arn09] est un système de mise à jour dynamique pour le noyau Linux. Le développement de ce système a débuté en 2007 comme projet de recherche à l'Institut de Technologie de Massachussetts aux États-unis.

Ce système permet de mettre le noyau Linux à jour sans avoir à redémarrer le système d'exploitation. Actuellement, on peut télécharger[20] et installer une version pour les distributions Ubuntu 9.04 et 9.10 sur des architectures X86 et X86-64.

Mise à jour du code. Pour la mise à jour du code, Ksplice prend en entrée un correctif du code source pour le fonctionnement actuel du noyau et génère le patch binaire qui est appliqué au noyau pour obtenir la nouvelle. Comme la plupart des systèmes de DSU pour modules C, Ksplice utilise la technique d'indirection pour accéder à la nouvelle version d'une méthode, ceci grâce à l'ajout des opérations/instructions de saut durant la mise à jour. Pour cela, Ksplice, remplace les premières instructions de l'ancienne version de la méthode par une instruction de saut vers la nouvelle méthode.

Mise à jour des données. Ksplice ne supporte que les modifications apportées aux instructions de méthodes et ne supporte pas les modifications de signature de variables, de signature de méthodes, etc. Ksplice ne supporte pas non plus la mise à jour des méthodes actives sur la pile d'exécution. Cependant, le niveau de flexibilité est suffisant pour corriger la plupart des bogues de sécurité dans les modules du noyau Linux.

La conception de Ksplice est basée sur trois aspects permettant respectivement de générer le code de remplacement, de résoudre les symboles/adresses dans le code de remplacement et de vérifier que le système de type est bien préservé après la mise à jour. Pour Ksplice, certaines hypothèses sur le système d'exploitation sont faites notamment :

1. les modules du noyau peuvent être chargées dynamiquement ;
2. le format ELF[21] utilisé généralement dans les noyaux Linux, BSD[22] et Solaris, est utilisé.

La solution requérant un accès au code source du module à mettre à jour et les patchs pour les modules propriétaires devraient être générés par son fournisseur ou par toute autre partie ayant accès au code source du module.

Ksplice génère le code de remplacement en trois étapes. Il commence par identifier les modifications effectuées entre l'ancienne et la nouvelle version. Pour cela il compare le fichier ELF binaire de l'ancienne version à celui de la nouvelle, ce qui permet d'identifier les modifications et de générer le code de remplacement approprié.

[20] Site web Ksplice : `http://www.ksplice.com/`
[21] Executable and Linkable Object
[22] Berkeley Software Distribution

Cependant, Ksplice doit générer des codes de remplacement qui ne font aucune hypothèse sur les adresses mémoire des fonctions et structures de données lors de la mise à jour dynamique des modules associés. En outre, ces codes de remplacement peuvent faire référence à des fonctions et des structures de données existantes dans le noyau en cours d'exécution. Pour cela, Ksplice utilise des options de compilations fournis par la plupart des compilateurs C y compris le compilateur GNUC C et le compilateur C d'Intel, il s'agit des options *–ffunction-sections* et *–fdata-sections*.

L'activation des ces options force le compilateur à générer les *offsets* pour les fonctions et les structures de données, et donc il en résulte un code plus général ne faisant pas de suppositions sur les adresses des fonctions et structures de données en mémoire. Le code objet résultant contient des instructions machine plus générales avec des *offsets* qui seront modifiés plus tard durant la phase de liaison (phase de *link*). Après avoir obtenu les codes de remplacement, Ksplice vérifie le code du noyau en cours en le comparant au code de l'ancienne version utilisée pour obtenir les codes de remplacement (appelé pré-code). Si une différence est constatée entre le pré-code du noyau et celui en cours d'exécution, alors le code de remplacement est erroné. Pendant ce processus de vérification, Ksplice peut aussi obtenir les informations sur les adresses mémoire du code et les structures de données des modules du noyau en cours d'exécution. Avec ces informations, Ksplice peut calculer les bonnes adresses, ce qui permet de mettre à jour les *offsets* dans les codes de remplacement.

POLUS

POLUS [CYC+07,CHYZ11] supporte la mise à jour dynamique pour des systèmes multithreads. Il a été devéloppé dans l'équipe calcul parallèle dans le cadre des travaux de recherche du groupe Système à l'Université Fudan en Chine[23].

Mise à jour du code. Pour la mise à jour du code, POLUS utilise un module de générateur de patch pour fournir le correctif (*patch*) relatif à l'ancienne et la nouvelle version, ce patch est compilé par un compilateur standard. Il utilise aussi un injecteur de patch qui a pour but d'insérer le patch dans le système en cours d'exécution. Le patch contient le code nécessaire pour maintenir la cohérence entre les threads qui manipulent les données partagées.

Mise à jour des données. POLUS est un système de DSU qui permet la coexistence de l'ancien et du nouvel état et utilise des fonctions de synchronisation sur les états à chaque fois qu'un accès en écriture est effectué dans l'un des états d'exécution (ancien ou nouveau).

Détection du point sûr. POLUS permet les mises à jour immédiates sans attente d'un point sûr du système, ceci en permettant la coexistence des anciennes et nouvelles versions après la mise à jour. Et il permet la mise à jour des signatures des méthodes et leurs implémentations.

Upstare

C'est un système de DSU qui a été développé dans le cadre des travaux de recherche sur la tolérance aux fautes[24] à l'Université de l'état d'Arizona aux États-unis

Upstare [MR07,MB09,Mak09] supporte la mise à jour dynamique des applications C multithreads. Son architecture est similaire aux systèmes de DSU présentés précédemment. Il est composé de

[23] `http://ppi.fudan.edu.cn/polus`
[24] `http://www.mkgnu.net/upstare`

plusieurs modules tels que le compilateur, le générateur de correctif, et un module de mise à jour proprement dit.

Mise à jour du code. Upstare compile en instrumentant une application en y ajoutant notamment des informations nécessaire à son processus de mise à jour. Il utilise la technique d'indirection pour les appels de fonctions.

Mise à jour des données. Upstare est l'unique système qui supporte la reconstruction de la pile durant la mise à jour dynamique. Il est capable d'extraire l'état de la pile et de le reconstruire afin qu'elle corresponde à celle de la nouvelle version de la fonction. Cette reconstruction de la pile peut être effectuée même si la méthode ou la fonction est active dans la pile d'exécution. La reconstruction de la pile assure que toutes les instances de fonction sur la pile d'appel sont mises à jour avant la poursuite de l'exécution. Un autre module d'Upstare, précise pour une fonction donnée, les correspondances entre l'ancien et le nouveau code permettant ainsi d'obtenir les points de reprise d'exécution après la mise à jour du code.

Détection de point sûr. Il instrumente les points d'entrées et de sorties des fonctions, les boucles, ceci permettant de déterminer les points dit « sûrs » du système. Upstare, après détection d'un point sûr, suspend tous les threads associés aux applications et applique une mise à jour atomique à l'aide d'un thread dit « thread coordinateur ». Upstare ne peut pas effectuer la mise à jour dynamique si un thread est bloqué en attente d'une ressource ou à cause d'un appel système. Cette restriction est particulièrement problématique pour les applications serveurs qui sont généralement multithreads et qui sont le plus souvent en attente de requêtes.

K42

K42 est un système d'exploitation d'usage général en cours de développement chez IBM [25]. Les premiers articles sur le projet K42 voient le jour à partir de 1997 [KS97, AFK+97, HAW+01, AHS+02]. Un des objectifs principal du projet est de développer une plateforme qui peut devenir une base de système d'exploitation et un noyau portable sur un large éventail de périphériques. Il fonctionne actuellement sur les systèmes *PowerPC*.

K42 est structuré comme un ensemble d'objets où chaque objet exporte une interface en déclarant une classe virtuelle de base. Ce système basé sur la programmation orientée objet supporte la mise à jour dynamique [BKA+05, BHS+05, Bau07]. Elle est effectuée avec un niveau de granularité qui est, une instance.

Mise à jour du code. L'ancien code est remplacé par le nouveau code de façon globale. Uniquement le transfert d'état des données est réalisé.

Mise à jour des données. Dans K42, chaque objet présent en mémoire possède une référence dans une entrée de la table des objets et chaque accès à un objet passe par la table des objets. Ainsi après la mise à jour des instances proprement dite, la mise à jour des entrées de la table d'objets est aussi effectuée. Et les accès aux nouvelles versions des objets sont effectués grâces aux nouvelles références contenues dans la table des objets.

[25] `https://researcher.ibm.com/researcher/view_project.php?id=2078`

K42 utilise la technique de « *clusterisation* » pour les instances. Ce mécanisme permet à un objet de contrôler sa propre distribution entre les processeurs. Dans K42, chaque objet est invoqué à l'aide d'indirection à travers une table de translation d'objets (OTT[26]). L'OTT est stockée dans la mémoire spécifique au processeur. Un objet peut accéder de façon transparente à ces autres copies ou exemplaires dans les mémoires des autres processeurs. Dans k42, La mise à jour dynamique des instances se base sur le mécanisme d'OTT. Ce processus comprend plusieurs phases :

1. Avant la mise à jour, les threads accèdent aux objets à travers l'OTT. Dans ce cas, il n'y a aucun coût supplémentaire autre que celui de l'indirection de pointeur vers la table OTT. Pour une instance donnée, K42 utilise un objet médiateur en mettant à jour l'entrée dans la table d'OTT correspondant à l'instance. Tous les threads en cours d'exécution accédant à l'instance passent par le médiateur qui redirige vers l'ancienne version de l'objet. L'objet médiateur met en place un compteur qui est décrémenté à chaque fois qu'un thread ayant accès à l'objet termine son exécution.
2. Une fois que tous les threads accédant à l'objet à mettre à jour ont terminé leur exécution, le médiateur de l'objet bloque les appels des nouveaux threads et l'instance peut donc être mise à jour à travers les fonctions de transfert.
3. Une fois l'objet mise à jour, le médiateur met à jour l'entrée correspondante dans la table OTT, ce qui permet aux threads d'acceder directement à la nouvelle version de l'instance. Le médiateur d'objet s'auto-détruit et détruit l'ancien objet et la mise à jour est terminée.

Dans K42, on remarque ainsi que les points « sûrs » sont déterminés par rapport à chaque objet à mettre à jour, et lorsqu'un point sûr est obtenu, les fonctions de transfert sont appelées pour la mise à jour.

Détection de point sûr. Pour effectuer la mise à jour, K42 restreint l'accès aux objets mis à jour, attend que toutes les requêtes en cours soient traitées, que tous les threads aient fini leur exécution, ensuite applique les fonctions de transfert pour obtenir les nouvelles versions des instances.

3.3.2 DSU dans le domaine Java

Dans le domaine Java, de nombreuses approches de DSU ont été proposées. Ces approches se divisent principalement en deux catégories :

1. Celles basées sur l'utilisation d'une machine virtuelle, notamment en modifiant celle-ci pour ajouter les fonctionnalités de mise à jour dynamique.
2. Celles s'appuyant sur d'autres techniques notamment les chargeurs de classes, les classes proxy, l'utilisation d'un compilateur spécial, etc. Le principal inconvénient de ces approches non axées sur une machine virtuelle est la flexibilité et les coûts de mises à jour élevés.

Systèmes de DSU basés sur une JVM

DVM & JDRUMS

DVM[27] a été mis en place dans le cadre du projet évolution dynamique dans les systèmes distribués orientés objets[28] à l'Université de Californie, Davis, aux États-unis.

[26] Object Translation Table
[27] Dynamic Virtual Machine
[28] DVM, `http://www.cs.ucdavis.edu/~pandey/Research/research.htm`

JDRUMS a été devéloppé dans le cadre du sous-projet DYNACOMP[29] du projet RISE regroupant deux équipes de chercheurs respectivement dans les Universités de Linköping et Växjö en Suède.

Mise à jour du code. JDRUMS [AR00, RA00] implémente la mise à jour dynamique à travers l'utilisation des tables d'indirections pour remplacer les anciennes versions des classes par les nouvelles en modifiant les adresses des classes.

DVM [MPG+00] permet la mise à jour dynamique en utilisant un chargeur de classes dynamique. Le chargeur de classes dynamique est une extension du chargeur de classes par défaut de la machine virtuelle Java. DVM propose le chargeur de classe afin de permettre aux développeurs de recharger une classe active grâce à la méthode *reloadClass* et de la remplacer par une nouvelle version grâce à la méthode *replaceClass*. Le chargeur de classe dynamique a été ajouté dans une machine virtuelle standard ce qui a donné lieu à la machine virtuelle avec chargeur de classe dynamique (DVM).

Mise à jour des données. DVM utilise un algorithme similaire à l'algorithme de ramasse-miettes « *Mark and Sweep* » pour rechercher et mettre à jour les instances des classes dynamiques. Cet algorithme ne permet pas des fonctions de transfert d'état fournies par le développeur, ainsi il initialise les nouveaux champs à NULL ou à 0 (zéro) selon le type.

Recherche de point sûr. DVM utilise une méthode agressive en ce qui concerne les méthodes actives sur la pile d'exécution. En effet, si une méthode à mettre à jour est présente dans la pile d'exécution d'un thread, alors ce thread est détruit et une exception est levée. Dans les systèmes multithreads, DVM semble inapproprié.

JVolve

Ce système a été développé dans le cadre du projet Jikes RVM[30] au département informatique de l'Université du Texas aux États-unis.

JVolve [SHM09] est un système de DSU basé sur la machine virtuelle appelée Jikes RVM. Ce système supporte les modifications de classe, des champs, des méthodes et des signatures.

Mise à jour du code. JVolve étend Jikes RVM pour y ajouter les fonctionnalités de mise à jour dynamique. Il possède un module de préparation de la mise à jour appelé UPT (*Update Preparation Tool*). Ce module permet de déterminer entre l'ancienne version et la nouvelle version de l'application :

1. les fichiers classes modifiés ;
2. les fichiers classes supprimés et ceux ajoutés.

Par la suite, seuls les fichiers des classes modifiées et ajoutées sont pris en compte durant la mise à jour.

Mise à jour des données. Grâce aux fonctions de transfert fournies par le programmeur ou générées par défaut, JVolve met à jour les instances des classes présentes dans le tas de la machine virtuelle. JVolve permet la mise à jour des signatures des classes, des méthodes et la mise à jour des byte codes des méthodes.

[29] `http://www.ida.liu.se/~pelab/rise1/subprojects/dynacomp.shtml`
[30] Jike RVM, `http://jikesrvm.org/`

Recherche de point sûr. Le module d'UPT a aussi pour fonction, d'identifier les points sûr du système en vérifiant qu'il n'y a plus de méthodes actives relatives à une méthode modifiée d'une classe à mettre à jour. Une fois, un point sûr détecté, l'application est stoppée, le JIT[31] recompile les nouvelles versions et les anciennes classes sont remplacées par les nouvelles.

Cependant, JVolve ne fonctionne pour l'instant que sur Jikes RVM et ne peut fonctionner sur une machine virtuelle par défaut.

Système de DSU sans support JVM

DUSC

Ce système de DSU [ORH02] a été développé dans le cadre de travaux de recherche pour le projet « test de l'évolution du logiciel »[32] à l'Institut Universitaire de Géorgie aux États-unis.

DUSC[33] permet de mettre à jour dynamiquement un programme Java qui s'exécute en remplaçant, en ajoutant et/ou en supprimant les classes.

Mise à jour du code. DUSC permet la mise à jour dynamique grâce à un système de classes proxy. La classe originale est remplacée par une classe proxy contenant une interface équivalente à la classe d'origine et un pointeur vers l'implémentation actuelle de celle-ci. La classe proxy est chargée de transférer tous les appels vers l'implémentation actuelle de la classe et s'assure aussi qu'il n'y a plus de méthode active de la classe sur la pile d'exécution avant la mise à jour des instances correspondantes. Après le chargement de la nouvelle version de la classe, le proxy est mis à jour afin de rediriger les futurs appels vers la nouvelle version.

Mise à jour des données. DUSC présente quelques limitations, notamment il ne permet pas la mise à jour des signatures des méthodes et les interfaces des classes. En plus, à chaque classe doit être associée une classe proxy lors de la mise à jour, ce qui cause des coûts supplémentaires dans le processus de DSU. Il n'effectue pas de recherche de point sûr.

HotSwap

HotSwap [KT08, Kim09] est un système de mise à jour dynamique basé sur la réécriture de byte code (*byte code rewriting*). Il a été développé au cours des travaux de recherche dans le laboratoire d'informatique configurable[34] à l'Université de Virginia Tech aux États-unis.

Ce système de DSU permet de mettre à jour dynamiquement les applications serveurs qui s'exécutent sur une machine virtuelle Java. Ceci est réalisé par un remplacement de classe (ancienne par la nouvelle) dans la VM. Il ne supporte ni l'ajout de nouveaux champs ou méthodes, ni la modification de leurs signatures.

Mise à jour du code. Il possède un GUI[35] permettant au développeur de charger les nouvelles versions de classe avec le code nécessaire pour la mise à jour au travers un *socket* de connexion. Pour

[31] Just-in-time Translation
[32] Testing of Evolving Software
[33] Dynamic Updating through Swapping of Classes
[34] Configurable Computing Lab, `http://www.ccm.ece.vt.edu/`
[35] Graphical User Interface

mettre à jour les classes, la machine virtuelle Java fait un appel de la méthode *RedefineClasses* () qui a pour objectif de remplacer les byte codes des méthodes par ceux des méthodes correspondantes dans la nouvelle version de la classe.

Mise à jour des données. HotSwap permet uniquement la mise à jour des byte codes des méthodes. Il ne supporte pas la mise à jour des instances (donc pas besoin de fonctions de transfert) et ne recherche pas les points sûr du système.

Iguana/J

Ce système de DSU a été développé dans le cadre des travaux de recherche de l'équipe Systèmes Distribués du département informatique à l'Université Trinity College de Dublin. L'implémentation actuelle de Iguana/J fonctionne avec la JVM basée sur la JDK 1.3 et uniquement sur les plateformes Windows 32 bits.

Iguana/J [RC00, RC02] supporte l'adaptation dynamique des applications Java à travers la modification de la machine virtuelle.

Mise à jour du code. Iguana/J permet la mise à jour dynamique à travers l'adaptation des classes. Chaque classe peut être vue sous deux niveaux : le niveau de base correspondant à la version courante de la classe, et le méta-niveau correspondant à la nouvelle version de la classe après son adaptation. Après la mise à jour, chaque appel de méthode de la classe de niveau de base est intercepté et redirigé vers celle du méta-niveau.

Iguana/J utilise la programmation orienté aspect (AOP[36]) pour les classes de méta-niveau. Ces dernières contiennent du code additionnel permettant d'injecter des codes dans les différentes méthodes de la classe originale. La correspondance entre les classes de méta-niveau et celles du niveau de base peut s'effectuer de façon statique à travers un fichier ou de façon dynamique à travers la redirection des appels.

Mise à jour des données. Iguana/J est un système de mise à jour dynamique orienté aspect qui permet uniquement la mise à jour du byte code des méthodes. Il ne supporte pas la mise à jour des signatures des méthodes et des classes, et ne supporte pas non plus la mise à jour des instances.

3.4 Analyse et positionnement

Il s'agit d'effectuer une analyse des diverses solutions existantes pour chaque problème scientifique de DSU et de voir si l'une peut être applicable ou adaptable pour EmbedDSU dans le contexte d'objets à fortes contraintes de ressources tels que les cartes à puce.

3.4.1 Mise à jour du code

La coexistence des versions n'est pas applicable dans le monde de la carte à puce. En effet le stockage des classes est réalisé dans le tas statique qui est la zone mémoire utilisée et facturée au client. De plus, cette technique impose un surcout lors de l'exécution non compatible avec les

[36] Aspect Oriented Programming

contraintes de la carte. Il est évident que pour des contraintes de restauration, les deux versions seront présentes pendant un laps de temps. Il est nécessaire de minimiser son utilisation par le processus de mise à jour.

L'utilisation de fonctions d'indirection pose aussi le problème du coût fixe à chaque appel et d'un point unique d'attaque. En effet un attaquant découvrant cette table peut rediriger toutes les méthodes vers sa propre définition. La solution utilisée actuellement dans les cartes utilisant des fonctions d'indirection est de réaliser une opération sur les adresses. Les adresses stockées dans la table sont généralement masquées à l'aide d'une fonction XOR afin de réduire les capacités de l'attaquant. Ceci ajoute évidement un surcout lors de l'exécution.

La dernière solution est la technique du saut qui présente l'avantage de ne pas avoir à gérer les références (édition de lien) pour le reste du code. Elle nécessite par contre un surcout important de stockage et rend difficile la vérification des bornes des méthodes. En Java, les sauts sont toujours relatifs il faut donc que les fragments de code soient consécutifs.

La seule possibilité pour la carte est donc d'utiliser une réécriture par copie / modification suivie d'une édition dynamique de lien. Cette solution minimise le surcout en temps d'exécution et l'occupation mémoire au détriment du coût de l'exécution de la mise à jour. Le scénario d'utilisation de la mise à jour de la carte prend en compte le fait que cette dernière se fait à une borne spécifique et non pas lors d'une transaction. Donc la durée de la mise à jour n'est pas un critère important alors que le temps d'exécution ainsi que l'occupation mémoire le sont.

3.4.2 Mise à jour des données

La mise à jour pose deux problèmes la modification et l'initialisation des champs d'une part et l'édition dynamique de lien d'autre part. Nous trouvons la même problématique d'optimisation quant aux données. Le choix entre une mise à jour par atteignabilité (paresseuse) ou par transformation immédiate procède des mêmes critères : optimisation des temps d'exécution et de l'espace mémoire. Il faut évidemment faire un choix de type non paresseux lors de la mise à jour : création des nouveaux objets et leur initialisation, puis réaliser l'édition de lien par recherche dans le code, les différentes frames et les objets des anciennes références et leur mise à jour.

Le premier problème de la mise à jour des données concerne la création des nouvelles structures et leur initialisation. Lorsqu'on va créer de nouveaux champs, Java exécutera immédiatement lors du chargement le constructeur par défaut (*clinit*) pour les champs statiques, et le constructeur (*init*) pour chaque instance mais ensuite il faudra réinitialiser avec soit les valeurs anciennes, soit les valeurs définies par les fonctions de transfert. L'opération d'initialisation n'est connue que par le développeur de l'application qui peut spécifier une opération devant être réalisée lors de la mise à jour sur chaque champ. Il faut donc que le langage utilisé pour décrire les opérations de mise à jour soit sémantiquement riche pour décrire ces fonctions.

3.4.3 Détermination des points sûrs

La détermination des points sûrs est une opération délicate pouvant théoriquement ne jamais se réaliser. En effet, une machine virtuelle Java Card ne s'arrête jamais à la différence d'une machine virtuelle Java classique. La solution simple est de faire passer la machine virtuelle en un mode de

mise à jour dans laquelle l'administrateur peut effectuer toutes les modifications à sa guise. Ce cas d'utilisation résout le problème mais limite la mise à jour sur une borne dans un environnement de confiance. Dans la mesure où nous envisageons d'autres scénarios plus complexes de mise à jour dans lesquels nous devons rechercher l'existence de méthodes restreintes. Le problème de la boucle infinie ne se pose pas dans le monde de la carte à puce car pour être déposée sur une carte une application doit vérifier certaines propriétés en particulier l'absence de boucles infinies.

3.4.4 Comparaison entre les systèmes de DSU

Le tableau 3.1 récapitule l'ensemble des systèmes de DSU abordés tout en présentant les différences qui en ressortent. La comparaison est axée autour de trois composantes :

1. Les modifications supportées. Il s'agit des modifications couramment rencontrées telles que la modification de la signature d'une méthode, la signature des champs, les instructions de méthodes, etc.
2. Les fonctions de transfert. Ici, on fait ressortir les systèmes qui peuvent générer automatiquement des fonctions de transfert, qui peuvent traiter des fonctions de transfert fournies en entrée par les développeurs et aussi la manière dont les fonctions de transfert sont utilisées (en mode paresseux ou non).
3. L'activation de la mise à jour. Il s'agit du moment où le processus de mise à jour est démarré, déterminer les systèmes de DSU qui recherche les points sûr de ceux qui ne le font pas (mise à jour immédiate) et de déterminer aussi les systèmes capable d'annuler une mise à jour et de restaurer le contexte d'exécution.

		DSU C/C++					DSU Java avec support JVM				DSU Java sans support JVM		
		Ginseng	Ksplice	Polus	Upstare	K42	JDrums	DVM	JVolve	EmbedDSU	DUSC	HotSwap	Iguana/J
Modifications supportées	Signature des méthodes	O	N	O	O	O	O	O	O	**O**	O	O	O
	Signature des variables	O	N	O	O	O	O	O	O	**O**	N	N	N
	Instructions des méthodes	O	O	O	O	O	O	O	O	**O**	O	O	O
	Méthodes actives	O	N	N	O	N	N	N	N	**N**	N	N	N
	Support du multitâche	O	O	O	O	O	N	N	O	**O**	N	N	N
Fonction de transfert (FT)	FT par défaut	O	N	N	O	N	O	O	O	**O**	N	N	N
	FT fournie en entrée	O	N	O	O	O	O	N	O	**O**	N	N	N
	FT actif(A)/Paresseux(P)	P	A	P	A	P	A	A	A	**A**	x	x	x
	Coexistence de données	O	N	O	O	O	N	N	N	**N**	O	N	O
Activation de la mise à jour	Recherche point sûr	O	N	N	O	O	O	O	O	O	N	N	N
	Mise à jour immédiate	O	O	O	N	N	O	O	O	**O**	O	O	O
	Processus de *roll-back*	N	O	N	N	O	N	N	O	**O**	N	N	N

TAB. 3.1 – Comparaison des systèmes de DSU

Légende
- N : Non
- O : Oui
- x : Non applicable

3.5 Conclusion

De nombreuses solutions existent pour chaque problème scientifique mais celles ci ne sont pas toujours applicable dans le monde de la carte. Dès lors, il s'agit de proposer une solution globale traitant de l'ensemble des problématiques définies dans ce chapitre pour la mise à jour dynamique sur carte tout en respectant des contraintes liées à la mémoire et au temps d'exécution. Dans la suite, nous présenterons EmbedDSU, notre système de DSU pour carte Java : son architecture, son implémentation, son évaluation et les perspectives de recherche sur ce système.

Troisième partie

Contributions et évaluations

Chapitre 4

EmbedDSU

Sommaire

Nous avons vu que le processus de mise à jour des applications sur la plateforme Java Card ne prenait pas en compte le côté dynamique de celle-ci. En effet, la *post-issuance* consiste à supprimer l'ancienne version, ensuite à charger et à démarrer la nouvelle. Ce processus présente des limites notamment pour la mise à jour des algorithmes de la machine virtuelle Java Card, qui ne s'arrête jamais et dont les objets sont sauvegardés et restaurés lors des sessions de communications avec le lecteur.

Nous présentons dans ce chapitre, EmbedDSU, un système de mise à jour dynamique, qui peut représenter une alternative au processus de *post-issuance*.

Nous commençons par présenter l'architecture générale d'EmbedDSU. Ensuite, nous décrivons les algorithmes mis en place pour chaque problème scientifique de la DSU notamment la mise à jour du code, des données et la recherche du point sûr. De plus, EmbedDSU prévoit un mécanisme de *roll-back* grâce au module de restauration de contexte, l'algorithme de ce dernier est aussi présenté.

4.1 Présentation de l'architecture d'EmbedDSU

EmbedDSU réalise la mise à jour dynamique avec une granularité d'une classe. L'architecture d'EmbedDSU se subdivise en deux parties : une partie off-card et une partie on-card comme illustrée par la figure 4.1.

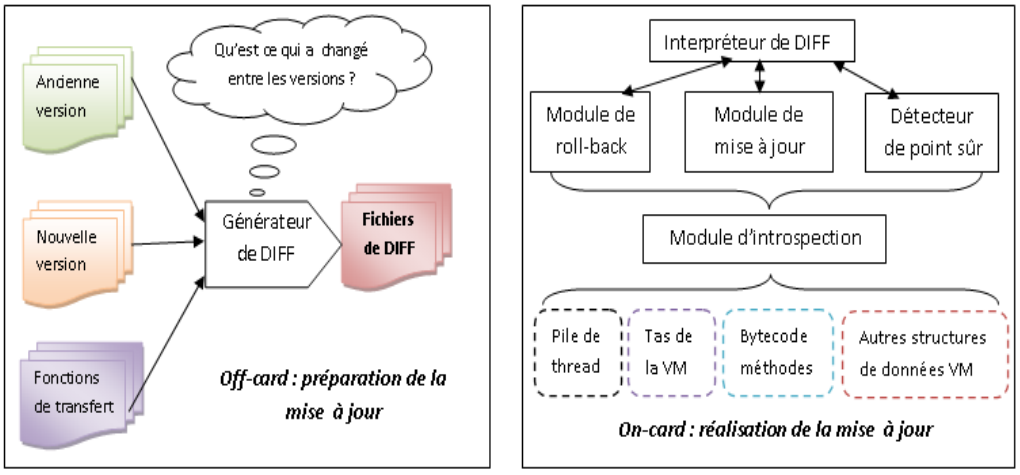

Fig. 4.1 – Architecture générale d'EmbedDSU

La partie off-card s'occupe de la préparation de la mise à jour qui est effectuée grâce à un module appelé générateur de DIFF. La partie on-card s'occupe de la mise à jour effective réalisée grâce à un ensemble de modules développés et intégrés dans la machine virtuelle. Ces modules communiquent entre eux pour garantir le bon fonctionnement du processus de DSU. Le processus général de mise à jour dynamique d'EmbedDSU peut être présenté par la figure 4.2.

4.1.1 Architecture off-card

L'architecture off-card d'EmbedDSU se décompose en deux sous-modules. Le module de génération des différences et le module de traitement des fonctions de transfert. L'architecture off-card peut être illustrée par la figure 4.3.

L'objectif en off-card est de déterminer les différences constatées entre les deux versions de la classe (ancienne et nouvelle version) et d'analyser les fonctions de transfert fournies par le développeur en vue d'extraire les informations à envoyer à la carte pour être utilisé durant la mise à jour. Les informations obtenues sont stockées dans un fichier appelé fichier de DIFF qui sera ensuite transféré à la carte.

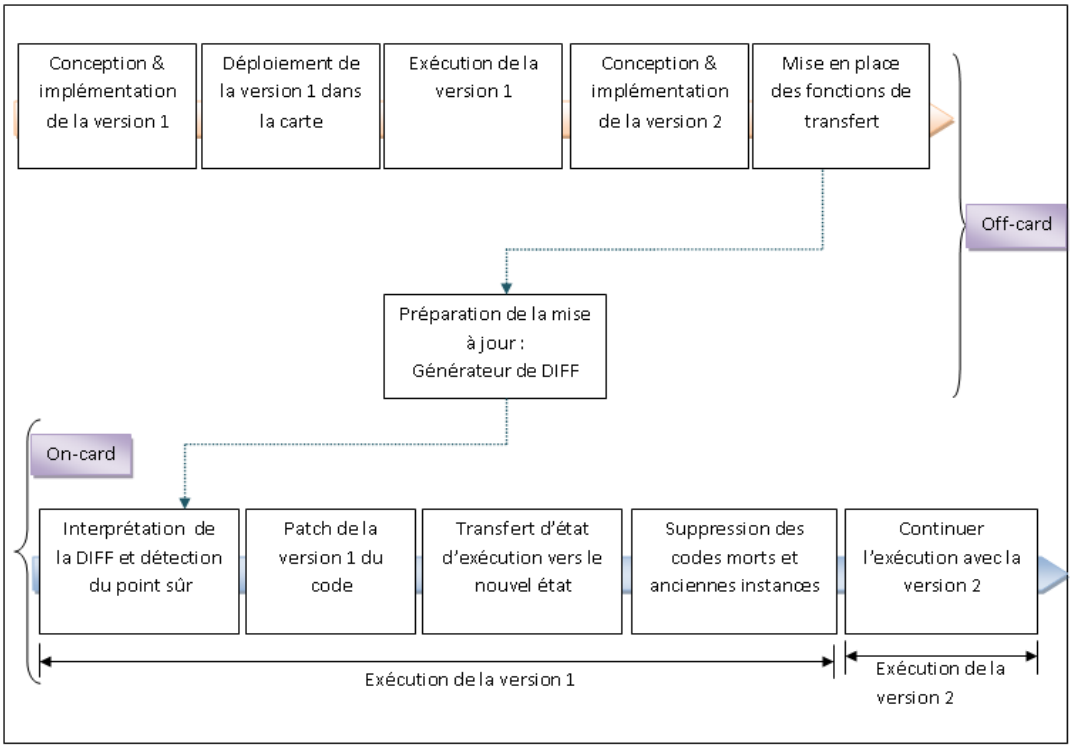

Fig. 4.2 – Processus général d'EmbedDSU

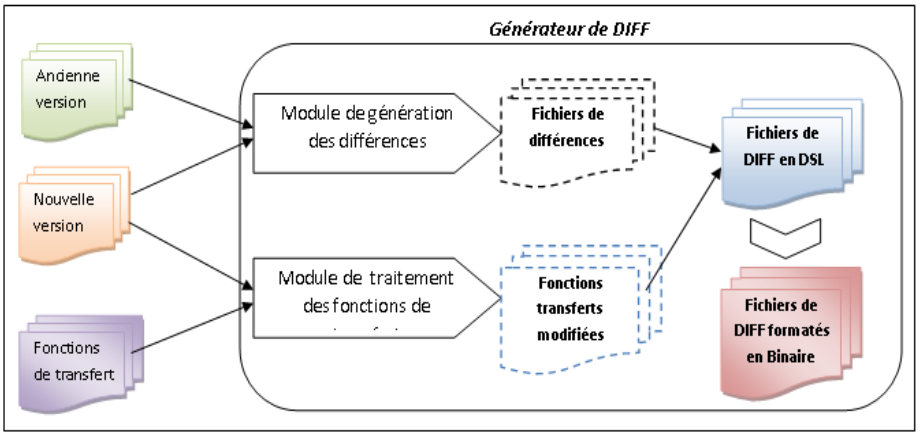

Fig. 4.3 – Architecture off-card d'EmbedDSU

Le module de génération de DIFF est donc chargé d'exprimer les différences entre les versions. Le module de traitement des fonction de transfert a pour rôle d'analyser les fonctions de transfert fournies par le programmeur et d'adapter les instructions de ces fonctions pour qu'elles puissent être utilisées en on-card pour la mise à jour.

Une fois ces différences et instructions de transfert obtenues, tout ceci est exprimé dans un langage

dédié (DSL[37]) mis en place à cet effet, on obtient ainsi un fichier intermédiaire.

Ce fichier intermédiaire est ensuite interprété pour générer le fichier de DIFF en binaire dans un format préalablement défini. Ensuite, ce fichier de DIFF binaire est transféré à la carte pour la mise à jour.

L'objectif du fichier binaire est d'obtenir un format compact des informations à envoyer dans la carte ce qui permet de réduire le temps d'envoi et l'empreinte mémoire occupée par le fichier de DIFF dans la carte.

Le fichier de DIFF binaire contient deux parties principales :

- Première partie : les modifications syntaxiques entre les deux versions permettant de spécifier à la partie on-card comment appliquer la mise à jour. Cette partie contient les informations sur les méthodes à supprimer, sur les champs à ajouter, sur les entrées de la table de constantes modifiées, etc.
- Deuxième partie : les instructions relatives aux fonctions de transfert fournies par le programmeur. En on-card, celles-ci sont utilisées pour transformer les instances dans le tas de la machine virtuelle afin d'en obtenir de nouvelles versions d'instances correspondant à la nouvelle version de la classe.

La suite de ce chapitre décrit plus en détail le processus on-card d'EmbedDSU et les algorithmes associés à la résolution des problèmes scientifiques de DSU dans le cadre d'EmbedDSU.

4.1.2 Architecture on-card

En on-card, afin de fournir les fonctionnalités de DSU, EmbedDSU étend la machine virtuelle Java en y ajoutant des modules relatifs à l'interprétation du fichier de DIFF, à la mise à jour du code, à mise à jour des frames, à la mise à jour des instances, à la détection du point sûr et à la restauration de contexte.

L'ensemble de ces modules communiquent entre eux comme illustré dans la figure ci-dessous 4.4.

Le module d'interprétation de la DIFF

Ce module est chargé d'analyser et d'interpréter le fichier de DIFF afin d'extraire les informations nécessaires pour initialiser les structures de données à utiliser lors de la mise à jour proprement dite (code, données, recherche point sûr) et lors de la restauration si échec. Ce module consiste à l'interprétation du fichier de DIFF et à l'initialisation des structures de données de mise à jour.

L'interprétation s'appuie sur la description du format du fichier DIFF binaire ce qui permet de déterminer où trouver les informations sur les champs, sur les méthodes, sur les fonctions de transfert, etc.

L'initialisation des structures de données de mise à jour est effectuée à partir des informations extraites dans le fichier de DIFF telles que les champs ajoutés ou supprimés, les méthodes ajoutées ou supprimées, les instructions ajoutées ou supprimées, etc. et celles obtenues en parcourant les structures de données de la machine virtuelle.

[37] Domain Specific Language

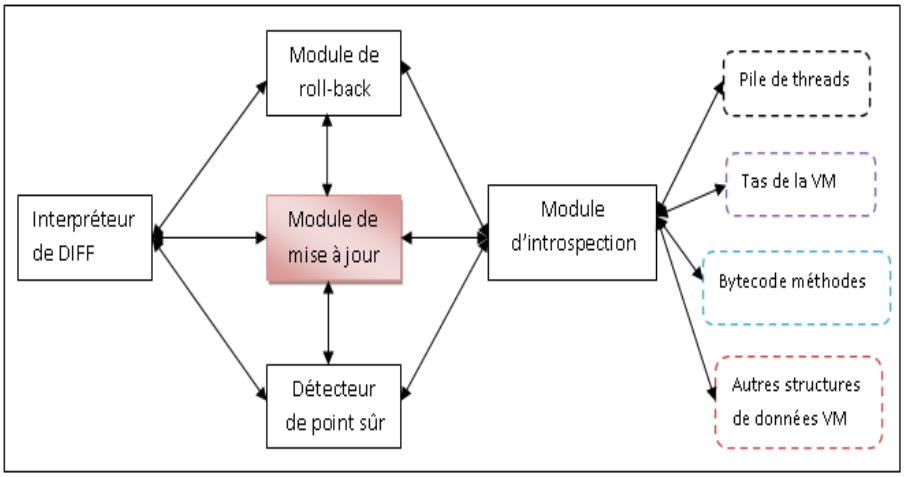

Fig. 4.4 – Communication entre les différents modules *on-card* d'EmbedDSU

Le module d'introspection

Ce module permet de parcourir les structures de données de la VM afin de fournir les informations sur celles-ci. Il s'agit par exemple :

- De déterminer les instances appartenant à la classe à mettre à jour et présentes dans le tas de la VM ;
- De déterminer l'ensemble de frames associées aux méthodes restreintes ;
- De fournir la référence d'une frame appartenant à une méthode donnée si celle-ci est active dans la pile de thread, etc.

Le module de détection de point sûr

En se basant sur les informations contenues dans les structures de données de mise à jour, ce module communique avec le module d'introspection pour déterminer s'il existe des méthodes restreintes dans la pile de thread. L'objectif étant de déterminer le moment opportun où la mise à jour peut être effectuée. Ce module est donc chargé de rechercher un point sûr du système. Une fois, un point sûr trouvé, le processus de mise à jour proprement dit peut commencer.

Le module de mise à jour

Ce module se base sur les informations fournies par le module d'introspection et celles fournies par le module de détection de point « sûr ». Il possède plusieurs fonctionnalités relatives à la mise à jour : les fonctionnalités de mise à jour du code, de mise à jour des instances, de mise à jour des informations dans les frames, de mise à jour des classes dépendantes[38], et la fonctionnalité de transfert d'état. Ce module contient donc plusieurs sous-modules relatifs à chacune de ses fonctionnalités.

[38] Une classe A est dite dépendante à la classe B si celle-ci fait appel à une méthode ou si elle accède à un champ de la classe B.

1. **Mise à jour du code**
   Ce sous-module se base sur les structures de données de mise à jour pour déterminer les méthodes qui ont été ajoutées, supprimées ou modifiées. A partir de ces informations, et les byte codes contenus dans le fichier de DIFF, il procède à la mise à jour du code par un système de recopie en modification que nous verrons plus en détail dans la partie 4.2.1.
2. **Mise à jour des instances**
   Ce sous-module s'appuie sur le module d'introspection en vue d'obtenir toutes les instances de la classe à mettre à jour et les références sur les structures de la VM affectées par la mise à jour de ces instances. Il s'appuie aussi sur les informations des structures de données de mise à jour pour déterminer les champs à ajouter, à supprimer ou à modifier. La mise à jour des instances est ensuite réalisée grâce au processus de recopie en modification.
3. **Mise à jour des frames**
   Ce sous-module permet de mettre à jour les informations dans les frames. Il s'agit notamment :
   (1) des références aux anciennes instances de la classe pour qu'elles pointent sur les nouvelles versions des instances ;
   (2) des adresses des méthodes non restreintes pour qu'elles pointent sur les nouvelles adresses de ces dernières.
   Ce module se base sur le module d'introspection pour déterminer les références et adresses de méthodes appartenant respectivement aux références des anciennes instances et anciennes adresses des méthodes de la classe en cours de mise à jour.
4. **Mise à jour des classes dépendantes**
   Ce sous-module se base sur le module d'instrospection pour déterminer les classes possédant des appels ou accédant aux champs de la classe en cours de mise à jour. Pour ces classes dépendantes, l'objectif du module est de mettre à jour les références vers la description du champ et les adresses de méthodes pour qu'ils correspondent à ceux de la nouvelle version de la classe.
5. **Fonctionnalité de transfert d'état**
   La mise à jour des instances passe aussi par l'initialisation correcte des champs ajoutés ou modifiés. Nous définissons deux types d'initialisation :
   – L'initialisation statique réalisée à partir d'une constante fournie en off-card ou à partir de la valeur par défaut définie dans la spécification de la machine virtuelle,
   – L'initialisation dynamique effectuée à partir d'une valeur correspondante à un résultat obtenu après évaluation d'une expression ou après l'exécution dynamique d'une méthode lors de la mise à jour.
   Ce sous-module se base sur le fichier de DIFF pour déterminer le type d'initialisation choisi par le programmeur et effectue le transfert d'état des instances.

Le module de roll-back

En cas de non atomicité de la mise à jour (arrachage de la carte par exemple), d'erreur, ou en cas d'opération illicite, ce module se base sur le module d'introspection et sur les informations fournies par les structures de données de mise à jour. Il permet de restaurer la version précédente du code, des instances, de la pile de thread et des structures de données de la VM qui ont été modifiées en fonction de l'état d'avancée du processus de mise à jour.

En effet, dans EmbedDSU, tant que la mise à jour n'est pas terminée, dans les structures de données de mise à jour, sont sauvegardées les références aux anciennes versions des instances et l'adresse de l'ancienne version de la classe en cours de mise à jour.

Ainsi, en cas d'échec, ces informations sont récupérées pour remplacer les valeurs des nouvelles références et nouvelle adresse dans les structures de données de la VM. Après le remplacement, le ramasseur de miettes est appelé pour libérer les espaces mémoires préalablement allouer aux nouvelles versions. Ce qui permet, à la fin de la restauration du contexte, d'utiliser les anciens objets et l'ancien code de la classe.

4.2 EmbedDSU et problèmes scientifiques de la DSU

Il s'agit de présenter les solutions adoptées dans le cadre d'EmbedDSU pour chaque problème scientifique de DSU, ceci grâce aux algorithmes associés. Dans la section suivante, nous définissons le concept de graphe de flot de contrôle (CFG[39]) utilisé par le générateur de DIFF en off-card, ensuite nous présentons la technique de recopie en modification.

4.2.1 Définitions

Présentation du concept de CFG

En informatique, un graphe de flot de contrôle est un graphe représentant l'ensemble des chemins que peut emprunter l'interpréteur lors de l'exécution d'un programme. Un flot de contrôle désigne l'ordre dans lequel sont exécutées les instructions ou fonction d'un programme. Il peut être modifié ou interrompu par divers types d'instruction pouvant entrainer la séparation des flots en zéro, un ou plusieurs chemins. Ces types d'instructions, dans le cas de Java en général et de Java Card en particulier, peut se subdiviser en plusieurs catégories :

1. Branchement inconditionnel ou saut : c'est le cas lorsque l'exécution se poursuit à une instruction différente.
2. Branchement conditionnel : c'est le cas lorsqu'il y a une condition qui doit être satisfaite afin qu'un groupe d'instruction puisse être exécuté. Dans ce cas, il y a au maximum, une alternative d'exécution.
3. Branchement conditionnel à choix multiples : ce sont des branchements conditionnels avec plus d'une alternative d'exécution.
4. Boucles : correspondent à l'exécution zéro ou plusieurs fois d'un groupe d'instructions tant qu'une condition n'est pas satisfaite ;
5. Arrêt inconditionnel : constitué des instructions capables d'arrêter l'exécution d'un programme notamment les retours de fonction ou les levées d'exception.

L'enchainement d'instructions successives ne contenant aucune rupture du flot de contrôle et aucune cible d'instruction de branchement, constitue un bloc élémentaire. Ce bloc possède un point d'entrée et un point de sortie, ainsi si un bloc est exécuté, chacune des instructions le composant sera exécuté sans interruption. Le point d'entrée du bloc peut être atteint de plusieurs autres blocs. Le point

[39] Control Flow Graph

de sortie du bloc pourra être toute instruction capable de rompre le flot de contrôle. Un CFG se compose d'un ensemble de sommets constitués par les blocs élémentaires et d'un ensemble d'arrêtes représentant les relations entre un point de sortie d'un bloc et un point d'entrée d'un autre bloc. Un exemple de CFG présentant les différents cas de figure peut être présenté par le schéma 4.5.

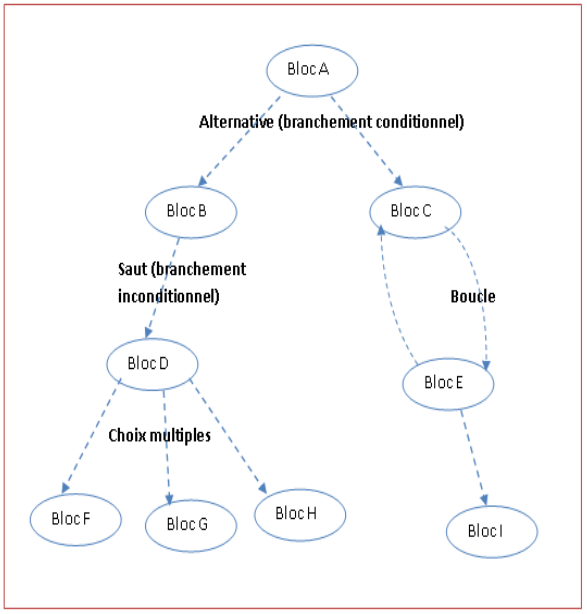

Fig. 4.5 – Un graphe de flot de contrôle

Le générateur de DIFF se base sur un CFG modifié. Ce dernier se construit de la même façon qu'un CFG. Cependant les blocs élémentaires sont constitués d'une seule et unique instruction. Une instruction est vue comme un ensemble de byte code Java.

La mise en place du CFG modifié est basée sur un algorithme capable de déterminer pour une méthode donnée, l'ensemble des byte codes correspondant à une instruction, appelés *patterns* d'instruction. Chaque *pattern* constitue donc un bloc élémentaire pour le graphe de flot de contrôle modifié. Et les arêtes sont mise en place selon le type de bloc élémentaire. Selon que l'instruction composant le bloc soit un saut inconditionnel, un branchement conditionnel, un choix multiple ou une instruction simple.

Méthode de recopie en modification

C'est une méthode qui consiste, lors de la recopie du code ou d'une instance – de l'ancien espace mémoire vers le nouveau –, à modifier ce dernier à partir des informations obtenues en off-card pour obtenir le code ou l'instance correspondant à la nouvelle version.

Plus précisément, lors de la mise à jour du code ou des instances, il s'agit pour les parties (du code ou des instances) qui n'ont pas été modifiées, de recopier celles-ci à partir de l'original et pour les parties modifiées, de se baser sur les informations obtenues à partir du fichier de DIFF pour déterminer l'action à réaliser.

L'utilisation de la méthode de recopie en modification présente un avantage réel notamment lors de la restauration du système. En effet, avec cette méthode, tant que la mise à jour n'est pas terminée, il existe toujours une copie de l'ancienne version du code et une copie des anciennes versions des instances.

4.2.2 Off-Card : Algorithme de génération du fichier de DIFF

Il commence par créer une structure arborescente liée à chaque classe. Ceci permet d'extraire et de ressortir pour chaque classe :

- La table de constantes ;
- Les droits d'accès ;
- Les interfaces ;
- Les champs de la classe ;
- Les méthodes de la classe.

Pour chaque méthode, il crée le CFG modifié associé. Une fois effectué, le générateur de DIFF effectue les comparaisons à tous les niveaux.

1. Au niveau des tables de constantes. Il s'agit de déterminer les entrées de la table de constantes qui ont été affectées par la mise à jour. Soit de déterminer les entrées de la table, qui ont été ajoutées, supprimées et de les sauvegarder dans le fichier de DIFF.

2. Au niveau des droits d'accès. Il s'agit de déterminer si les droits d'accès de la classe ont été modifiés. Par exemple, une classe qui passe du droit d'accès *default* à *private* ou *public*. Les modifications détectées sont exprimées et rajoutées dans le fichier de DIFF.

3. Au niveau des champs de classe. Il s'agit de déterminer les champs ajoutés, supprimés ou modifiés entre les deux versions. Pour chaque champ modifié, il faut déterminer le ou les types de modifications effectuées (modification du type, du droit d'accès, modification de l'index, modification de la valeur initiale, etc.).

4. Au niveau des méthodes de la classe. Le générateur de DIFF, à partir des arborescences des deux versions de la classe, détermine les méthodes ajoutées, supprimées et modifiées. Pour chaque méthode modifiée, il détermine les modifications au niveau de la signature des méthodes, des variables locales.

5. Ensuite, il compare les deux CFGs associés pour déterminer les instructions ajoutées et supprimées et rajoute toutes ces informations dans le fichier de DIFF. La règle appliquée au niveau des attributs de la classe (champs, méthodes, interfaces, etc.) est la suivante : les attributs apparaissant uniquement dans l'ancienne version sont des attributs supprimés et ceux apparaissant uniquement dans les nouvelles versions sont des attributs rajoutés.

L'algorithme générateur de DIFF peut être présenté comme suit :

Algorithm 2 Générateur du fichier de DIFF

ENTRÉES: Ancienne version de la classe C, Nouvelle version de la classe C'
SORTIES: Fichier de DIFF constitué de résultat1,2,3,4,5,6,7

1: Construire les structures arborescentes de A et A' des fichiers classes de C et C', à l'aide de BCEL.
2: À partir des structures arborescentes A et A'
3: Comparer les tables de constantes et générer résultat1
4: Comparer les droits d'accès et générer résultat2
5: Comparer les interfaces et générer résultat3
6: Comparer les champs et générer résultat4
7: Comparer les méthodes et générer résultat5
8: **pour** chaque paire de méthodes modifiées <m, m'> **faire**
9: Identifier les modifications au niveau des signatures et types de retours et générer résultat6
10: Créer les CFG G et G' associés,
11: Identifier les modifications au niveau des instructions et générer résultat7
12: Identifier les modifications au niveau des exceptions et générer résultat8
13: **fin pour**
14: **retourner** *resultat*1, *resultat*2, *resultat*3, *resultat*4, *resultat*5, *resultat*6, *resultat*7

Le contenu du fichier de DIFF :

résultat1 :	entrées de la table de constantes ajoutées ou supprimées,
résultat2 :	droits d'accès ajoutés, supprimés ou modifiés,
résultat3 :	interface ajoutés ou supprimés,
résultat4 :	champs ajoutés ou supprimés (le type, index dans la table de constante, les valeurs initiales, droit d'accès)
résultat5 :	méthodes ajoutées ou supprimées,
résultat6 :	pour les méthodes modifiées (la signature, type de retour et les variables locales)
résultat7 :	pour les méthodes modifiées, liste des instructions modifiées (ajoutées, supprimées)
résultat8 :	les exceptions ajoutées, supprimées, ou modifiées.

4.2.3 Mise à jour du code

Approche

L'approche proposée se base sur les informations obtenues à partir du fichier de DIFF pour mettre à jour le byte code des méthodes de la classe et les métadonnées de la classe. Elle consiste en une recopie en modification de la classe.

La mise à jour du code consiste non seulement à mettre à jour la classe concernée, mais aussi à mettre à jour les classes dépendantes. En effet, une classe A peut faire appel à une méthode modifiée de la classe B mise à jour, auquel cas il est nécessaire de remplacer dans la classe A, l'ancienne référence de la classe B par la nouvelle référence obtenue après la mise à jour.

On peut donc définir les étapes de mise à jour :

- Mise à jour du code de la classe par recopie en modification ;
- Mise à jour des classes dépendantes.

Algorithmes

Nous présentons dans un premier temps, l'algorithme de recopie en modification du code. Il est basé sur les informations contenues dans les structures de données de mise à jour et sur les byte codes des instructions ajoutées (s'il y en a) contenues dans le fichier de DIFF stocké en on-card. L'algorithme suivant présente la recopie en modification du code d'une méthode donnée.

Algorithm 4 Recopie en modification du code

Entrées: Ancienne adresse de la méthode, taille de l'ancienne méthode, fichier de DIFF, structure de donnée de mise à jour
Sorties:

```
1: pour un compteur de 1 à la taille de l'ancienne méthode faire
2:   Vérifier les informations dans les informations de données de mise à jour
3:   si le type de modification est l'ajout alors
4:     recopier les instructions fournies dans le fichier de DIFF
5:   sinon
6:     si le type de modification est la suppression alors
7:       aucune recopie n'est faite
8:     sinon
         recopier l'instruction de l'ancienne version du code et mettre à jour
9:     fin si
10:  fin si
11: fin pour
```

La mise à jour de la classe se fait grâce à la recopie en modification des byte codes des méthodes de la classe. Cependant une classe possède des métadonnées notamment la table de constante dont il est nécessaire de mettre à jour. La table de constante peut aussi être mise à jour par recopie en modification grâce aux informations fournies par les structures de données de mise à jour.

Algorithm 6 Mise à jour de la table de constante

Entrées: Table de constante de la classe à mettre à jour
Sorties: Nouvelle version de la table de constante

```
1: pour chaque entrée de la table de constante faire
2:   Vérifier les informations fournies par le fichier de DIFF
3:   si le type de modification pour cette entrée est l'ajout alors
4:     recopier les informations fournies dans le fichier de DIFF
5:   sinon
6:     si le type de modification pour cette entrée est la suppression alors
7:       aucune recopie n'est faite
8:     sinon
         recopier l'entrée de l'ancienne version du code et mettre à jour
9:     fin si
10:  fin si
11: fin pour
```

L'algorithme de mise à jour de la classe est basé sur les deux algorithmes précédents. Après recopie en modification des métadonnées, des entêtes et byte codes des méthodes, cet algorithme retourne l'adresse de la nouvelle version de la classe.

Algorithm 8 Mise à jour du code de la classe

Entrées: Ancienne adresse de la classe, fichier de DIFF, structure de données de mise à jour
Sorties: Nouvelle adresse de la classe

1: A partir des informations de la DIFF
2: Mettre à jour les entrées de la table de constante et les autres métadonnées
3: **pour** chaque méthode de la classe **faire**
4: recopie en modification de l'entête de la méthode,
5: recopie en modification du byte code de la méthode
6: **fin pour**
7: **retourner** *Nouvelleadressedelaclasse*

Après la mise à jour de la classe et l'obtention de la nouvelle adresse de celle-ci. Il est nécessaire de mettre à jour les références aux méthodes et champs de celle-ci dans les classes dépendantes.

Algorithm 10 Mise à jour des classes dépendantes

Entrées: Application à mettre à jour, Adresse de la nouvelle version de la classe A
Sorties:

1: **pour** chaque classe B de l'application **faire**
2: **si** une référence à une méthode ou à un champ de la classe A est trouvée **alors**
3: Mettre à jour la référence de la méthode ou du champ pour qu'elle corresponde à celle de la nouvelle version dans A.
4: **fin si**
5: **fin pour**

4.2.4 Mise à jour des données

Il s'agit de la modification de toutes les données impactées par la mise à jour, relative au contexte d'exécution de la classe à mettre à jour.

Dans notre approche, le contexte d'exécution d'une classe représente l'ensemble des structures et informations utilisées par cette classe durant son exécution. Il s'agit de :

- l'ensemble des objets de la classe dans le tas de la machine virtuelle,
- de l'ensemble des frames des méthodes de la classe dans la pile de threads,
- des informations relatives au code de la classe et aux byte codes de ses méthodes,
- et des autres structures de données de la machine virtuelle.

Toutefois, la mise à jour des données concerne précisément les instances dans le tas et les frames dans la pile de thread.

Solution proposée

Mise à jour des instances

Notre approche est basée sur la technique de transfert d'état à partir des fonctions de transfert générées automatiquement et/ou celle fournies par le programmeur. EmbedDSU utilise un système de transfert d'état dit «non paresseux» et applique ainsi les fonctions de transfert à tous les objets à mettre à jour, avant de continuer l'exécution de la nouvelle version de la classe.

Rechercher les objets à mettre à jour nécessite d'introspecter le tas de la machine virtuelle et de parcourir l'ensemble des objets afin de déterminer ceux concernés par la transformation. Pour cela, EmbedDSU s'appuie sur l'algorithme du collecteur de mémoire (GC[40]).

Le processus de mise à jour des instances peut être subdivisé en plusieurs étapes :

- La recherche des instances de la classe dans le tas de la machine virtuelle ;
- Le transfert d'état de l'ancienne version des instances pour qu'elles soient cohérentes avec la nouvelle version de la classe ;
- La mise à jour des frames dans la pile de threads.

La recherche des instances

L'algorithme de recherche des instances est basé sur un algorithme de ramasse-miettes. L'objectif est de rechercher toutes les instances vivantes de la classe à mettre à jour, ceci à partir de l'ensemble de racine de persistance.

Une racine de persistance représente tout objet ou structure de donnée initiale à partir de laquelle on peut retrouver un ensemble de références d'objets présents sur le tas de la VM. Une racine de persistance peut être :

- Une pile de frame (une pile d'opérandes, une pile de variables locales) ;
- Les variables statiques ;
- Les variables de classe ou instances de classe pouvant contenir des références vers d'autres objets dans le tas.

Il s'agit de trouver à partir de l'ensemble de racines de persistances, tous les objets d'instances de la classe à modifier pouvant être atteints. Chaque objet trouvé est directement modifié et un drapeau de mise à jour est placé sur ce dernier.

Transfert d'état

Notre approche consiste en une recopie en modification des instances trouvées. Pour chaque instance trouvée, les opérations effectuées sont les suivantes :

- La création de la nouvelle instance ;
- L'initialisation de la nouvelle instance pour qu'elle soit cohérente avec la nouvelle version de la classe.

Création de la nouvelle instance. Il s'agit de créer un nouvel objet correspondant à la nouvelle version à partir des informations obtenues grâce au fichier de DIFF. Ceci est effectué par une recopie en modification des instances.

[40] Garbage Collector

Algorithm 12 Recopie en modification d'une instance

Entrées: Ancienne référence de l'instance, tas de la VM, structures de données de mise à jour
Sorties: Nouvelle référence de l'instance

```
1: Récupérer l'instance correspondant à la référence fournie
2: pour chaque champ de l'instance faire
3:   si le champ est non modifié ou si uniquement la valeur du champ a changé alors
4:     recopier en modification le champ concerné
5:   sinon
6:     si le champ est ajouté ou si le type du champ a changé alors
7:       Ajouter un champ du type correspondant à la nouvelle instance en cours de création
8:     sinon
9:       si le champ est supprimé alors
10:        aucune action de recopie n'est faite
11:      fin si
12:    fin si
13:  fin si
14: fin pour
```

Après la création de la nouvelle instance correspondant à la nouvelle version de la classe, l'initialisation des champs est effectuée. Cette initialisation concerne les champs dont les valeurs ont été modifiées et les champs ajoutés.

L'initialisation est réalisée grâce aux fonctions de transfert. Cette initialisation peut être effectuée soit :

- par les valeurs par défaut fournies par la spécification de la machine virtuelle ;
- par une constante ;
- ou par le résultat de l'exécution d'une méthode ou de l'évaluation d'une expression.

A chaque type d'initialisation est associé un type de fonction de transfert. On peut donc avoir :

- Les fonctions de transfert par défaut. Ce sont des fonctions générées automatiquement pour l'initialisation statique (valeurs par défaut ou constante fourni dans le fichier de DIFF) ;
- Les fonctions de transfert fournies par le programmeur pour l'initialisation dynamique (expression, méthode).

L'algorithme d'initialisation consiste à déterminer tout d'abord le type d'initialisation grâce au contenu du fichier de DIFF, ensuite de réaliser l'action associée.

La mise à jour de certains champs des instances peut nécessiter l'exécution des deux types de fonctions de transfert. On peut prendre un cas simple où des champs ont été ajoutés. Dans ce cas, EmbedDSU exécute automatiquement la fonction de transfert par défaut et si une fonction de transfert est fournie par le programmeur alors celle-ci est aussi exécutée. Cependant, ils existent aussi des cas où aucune fonction de transfert n'est exécutée par exemple si aucun champ n'a été modifié, ou s'il y a eu uniquement suppression de certains champs et que le programmeur n'a fourni aucune fonction de transfert.

Algorithm 14 Transfert d'état d'une instance

Entrées: Référence nouvelle version d'une instance, structures de données de mise à jour
Sorties:
1: **pour** chaque champ de l'instance **faire**
2: **si** le champ est ajouté ou si uniquement le type du champ a été modifié **alors**
3: Initialiser à partir de la fonction de transfert générée automatiquement
4: **fin si**
5: **fin pour**
6: Exécuter la fonction de transfert fournie par le programmeur sur l'instance, s'il en existe

L'algorithme de mise à jour des instances découle directement des algorithmes précédents. Cependant, une phase de recherche d'instance est effectuée grâce à la modification de l'algorithme du ramasse-miettes.

Algorithm 16 Mise à jour des instances

Entrées: Anciens objets, tas de la VM, fichier de DIFF
Sorties: Nouveaux objets
1: Parcourir le tas de la VM à la recherche des instances de la classe à mettre à jour
2: **pour** chaque instance trouvée **faire**
3: recopie en modification de l'instance
4: transfert d'état de l'instance
5: **fin pour**

Mise à jour des frames

Après la mise à jour des instances, la mise à jour des frames est incontournable. En effet, dans la pile d'opérandes et la table de variables locales des frames, peuvent se trouver des références aux objets ou instances précédemment mis à jour. Il convient donc de mettre à jour toutes ces références pour qu'elles pointent sur les nouveaux objets correspondant à la nouvelle version de la classe.

Algorithm 18 Mise à jour des frames

Entrées: Anciennes références des objets, Nouvelles références, pile Java, structures de données de mise à jour
Sorties:
1: Parcourir la pile Java
2: **pour** chaque pile de thread **faire**
3: Parcourir la pile de thread à la recherche des frames contenant des anciennes références des objets mis à jour
4: **pour** chaque frame trouvée **faire**
5: parcourir les entrées du frame
6: **pour** chaque ancienne référence trouvée **faire**
7: Remplacer celle-ci par la nouvelle référence correspondante.
8: **fin pour**
9: **fin pour**
10: Positionner le drapeau de mise à jour à la pile de thread traité, permettant en cas de roll-back de localiser directement l'ensemble des piles de threads affectées par la mise à jour des références
11: **fin pour**

4.2.5 Recherche de point sûr

Solution adoptée

La détection du point sûr est basée sur la technique des méthodes restreintes. Ainsi, un point sûr est obtenu lorsque aucune frame dans la pile Java n'appartient à une méthode modifiée de la classe à mettre à jour. La méthode proposée ne résout pas le problème de boucle infinie. En effet, si une méthode restreinte est active et contient une boucle infinie, un point sûr n'est jamais atteint ceci dû au fait que la méthode restreinte ne terminera jamais son exécution.

La solution proposée est le comptage de référence, qui consiste à compter le nombre de frames associées aux méthodes restreintes, présentes dans la pile Java. De plus, il est nécessaire de définir un seuil d'attente durant la recherche du point sûr. Le seuil d'attente dans EmbedDSU représente le nombre maximum de méthode pouvant être exécuter durant la recherche du point sûr. Ce seuil peut être défini par le programmeur pour chaque classe à mettre à jour.

Algorithm 20 Recherche de point sûr

Entrées: Références des méthodes modifiées, Pile de Java, CompteurReference, Seuil
Sorties:

```
1: Parcourir la pile Java
2: pour chaque pile de thread faire
3:    Parcourir la pile de thread à la recherche des frames associés aux méthodes restreintes
4:    pour chaque frame trouvée faire
5:       si la frame est associée à une méthode restreinte alors
6:          incrémenter CompteurReference
7:       fin si
8:    fin pour
9: fin pour
10: pour chaque frame terminant son exécution faire
11:    si la frame est associée à une méthode restreinte alors
12:       Décrémenter CompteurReference
13:       Décrémenter Seuil
14:       si CompteurReference atteint zéro alors
15:          un point sûr est trouvé
16:       sinon
17:          si Seuil atteint zéro alors
18:             arrêter la recherche du point sûr et annuler la mise à jour
19:          fin si
20:       fin si
21:    fin si
22: fin pour
```

4.3 Mises à jour supportées par EmbedDSU

EmbedDSU supporte plusieurs catégories de mise à jour tant au niveau de la signature de la classe qu'au contenu des méthodes. Ces catégories de modification peuvent être subdivisées en quatre groupes :

Modification signature de la classe	Ajout et suppression des méthodes Modification de la signature des méthodes Ajout et suppression de champs Modification de la signature d'un champ (type et droit d'accès) Modification d'une valeur d'initialisation d'un champ.
Modification de la signature des méthodes	Ajout et suppression d'un paramètre Modification du type d'un paramètre Modification du type de retour d'une méthode Modification des droits d'accès (*private*, *public*, *static*, etc.) d'une méthode
Modification du contenu des méthodes	Ajout et suppression d'une variable locale Modification de la signature d'une variable locale Ajout, suppression et modification d'une ou plusieurs instructions
Fonctions de transfert	Initialisation statique Initialisation dynamique

Tab. 4.1 – Modifications supportées par EmbedDSU

4.4 Mise à jour non supportées par EmbedDSU

EmbedDSU ne supporte pas :
- La modification des interfaces d'une classe,
- La modification de la hiérarchie de la classe,
- La modification des exceptions.

En ce qui concerne la modification de la hiérarchie, changer les instructions d'importations modifie les classes et les interfaces référencées dans le code de la classe, et peut avoir plusieurs types d'implications qui peuvent être :

- Le type d'un champ ;
- La chaine d'héritage qui change, etc.

Cependant, on n'a pas besoin de type de modification lié spécialement à l'import puisque durant la compilation, ceci est utilisé pour vérifier les méthodes et champs référencés.

En ce qui concerne la modification des interfaces, il suffit d'étendre le processus d'EmbedDSU pour prendre en compte ce type de modification.

4.5 Conclusion

EmbedDSU est constitué d'un ensemble de mécanismes off-card et on-card. On constate qu'une fois les informations de mise à jour fournies – notamment le fichier de DIFF –, la mise à jour est réalisée par la suite en on-card sans aucune intervention humaine. EmbedDSU fournit un processus de restauration qui s'appuie essentiellement sur l'avantage de la méthode de recopie en modification. En effet, les anciennes références aux objets, les anciennes adresses du code et le code de la classe ne sont pas impactés par la mise à jour, ce qui permet en cas de non-atomicité de la mise à jour ou en cas d'opération illicite de restaurer l'ancien contexte.

Dès que la mise à jour est terminée, avant de continuer l'exécution, la libération de la mémoire est effectuée grâce au ramasse-miettes et l'espace associé à l'ancienne version de la classe est dés-alloué.

Dans le chapitre suivant, nous présentons un prototype d'EmbedDSU et son évaluation sur la plateforme d'évaluation AT91 EB40A. Ce prototype est basé sur la machine virtuelle SimpleRTJ, qui est une VM Java pour l'embarqué.

Chapitre 5

Prototype et Évaluation

Sommaire

Les chapitres précédents ont expliqué le principe et l'architecture de notre approche. Nous avons réalisé, dans cette thèse, une implémentation en C et Java basé sur la machine virtuelle simpleRTJ pour démontrer la faisabilité et la validité de cette approche. Ce chapitre présente les éléments de cette implémentation.

Dans la suite, nous présentons les outils développés dans chacune des parties de notre proposition (off-card et on-card). Nous commençons par une description détaillée de simpleRTJ, la machine virtuelle sur laquelle le système EmbedDSU est basé. Puis nous décrivons l'implémentation pour chaque élément de l'architecture d'EmbedDSU.

5.1 Présentation de simpleRTJ

SimpleRTJ est une machine virtuelle Java optimisée, conçue pour des petits objets embarqués basés sur des architectures 8/16/32 bits processeur avec une faible empreinte mémoire (18-24Kb).

Elle supporte le multitâche, les ramasse-miettes et les mémoires linéaires (64Kb/16Mb) et *banked* (64kb). Cependant, elle utilise une technique de pré-résolution à l'extérieur de la carte (en mode *off-line*) des dépendances permettant ainsi d'éviter le chargement dynamique et la nécessité d'embarquer toutes les bibliothèques sur la plateforme (et donc de réduire le coût d'exécution et d'occupation mémoire à l'intérieur de la carte).

5.1.1 Architecture générale

En tant que machine virtuelle, simpleRTJ assure la gestion de la mémoire et des threads du système.

Gestion de la mémoire
La gestion de la mémoire volatile est effectuée par un système d'allocation et de dés-allocation de l'espace mémoire. L'allocation d'espace mémoire peut être effectuée dans le tas, dans la pile de thread, au niveau de la table de référence et de la table de threads. La taille de l'espace mémoire allouée pour un objet, une frame ou une entrée dans l'une des tables correspond respectivement à la taille maximale d'un objet de la classe associée, de la taille maximale d'une frame dans le système, de la taille de la structure de donnée correspondant à une entrée de la table de référence ou de thread.

La dés-allocation se fait grâce au collecteur de mémoire ou au ramasse-miettes (GC [41]) qui a donc pour rôle de libérer l'espace en mémoire pour une frame, un objet ou tout simplement pour une entrée dans la table de threads ou de références. La technique du GC simpleRTJ est basée sur celle du MSC (*Mark-Sweep-Compact*), toutefois le compactage peut se réaliser de façon séparée au MS (*Mark-Sweep*). Le procédé global est le suivant :

- S'il n'y a plus d'espace mémoire pour une donnée alors on libère de l'espace appropriée.
- Si l'espace trouvée ou libérée ne peut être utilisable par la donnée, alors on compacte la mémoire (défragmentation de la zone mémoire).
- La défragmentation est aussi effectuée lorsque la taille de mémoire libre atteint un certain seuil critique.

Fonctionnement du GC
L'algorithme général du GC de simpleRTJ consiste à parcourir le tas de la VM et de marquer par une couleur, les objets référencés. Ils peuvent être :

- Les champs statiques des classes,
- Les champs des instances des classes dans le tas,
- Ou des éléments de tableaux.

A la fin du processus de marquage, les objets qui n'ont pas été marqués (blanc) peuvent être considérés comme des objets "poubelles" donc à supprimer. Au début du processus du GC, on suppose que tous les objets soient marqués en blanc (ceci est effectué lors de l'allocation d'un espace mémoire pour un objet ou un tableau) sauf les objets d'instances du système ou de la VM sont marqués en noir (objets toujours vivants). Ensuite, le GC parcourt la pile de thread, les champs statiques, les tableaux, les champs des objets à la recherche des références à un objet dans le tas. Si un objet est marqué en blanc alors il n'a pas encore été visité, Une fois, une référence

[41] Garbage collection

trouvée, si l'objet correspondant contient des références vers d'autres objets dans le tas, alors il est marqué en gris (à revisiter) sinon il est marqué en noir (objet vivant). Et le processus recommence pour les objets référencés par les objets marqués en gris. A la fin, tous les objets marqués en blanc sont à collecter et les objets gris et noir sont marqués en blanc pour le prochain ramassage de miettes (GC).

Gestion des threads
Lors de l'exécution d'une application java, un thread principal est créé, celui qui exécute la méthode main de l'application. Par la suite, plusieurs autres threads peuvent être crées, liés à l'application en cours ou à d'autres applications. A chaque thread Java est associée une pile d'exécution permettant de stocker pour chaque appel de méthode, les informations relatives à l'exécution de la méthode (frame). Ainsi, une pile d'un thread à un instant est un ensemble de frame représentant l'ensemble d'appels imbriqués des méthodes de diverses classes d'un instant donné à l'instant choisi. La pile d'un thread lié à une application représente donc un ensemble de frame hybride (frames associées aux méthodes de différentes classes).

5.1.2 Machine virtuelle simpleRTJ : Spécificités

Chaque VM possède un système de chargement de classe et un système d'exécution représentant respectivement le mécanisme permettant de charger les classes et les interfaces à partir du nom fourni et le mécanisme responsable de l'exécution du code des méthodes des classes chargées.

Processus chargement
Le processus relatif à simpleRTJ n'est pas tout à fait le même que celui d'une VM standard. En effet :

1. Une application n'est pas constituée d'un ensemble de packages, elle est plutôt représentée par un seul et unique fichier obtenu par compilation statique et préalablement pré-linkée, contenant toutes les classes de l'application ainsi que toutes les autres classes et interfaces nécessaires à son exécution. Ce fichier appelé fichier binaire d'extension *bin* est ensuite chargé dans la machine virtuelle pour son exécution. Sa structure est présentée en détail à l'annexe C.
2. Il n'y a pas de vérificateur de byte code intégré, donc la vérification du byte code chargée n'est pas effectuée, par contre le contrôle de type est effectué lors de l'exécution des byte code,
3. Une fois, le fichier binaire chargé, lors de l'appel d'une classe donnée, celle ci n'est pas chargée dans la zone de méthode (inexistante dans simpleRTJ). L'instanciation d'objet de la classe passe par la table de constante (récupération de l'adresse de la classe) et la table de champs de la classe (pour déterminer le nombre de champs, le type, le flag associé, etc.).
4. Il existe un MMU[42] permettant d'effectuer la résolution d'adresse à travers la table de constante de la classe considérée. Son rôle étant de récupérer l'adresse relative et d'obtenir l'adresse absolue par calcul basée sur l'adresse de début de localisation de la classe en mémoire persistante.

[42] Memory management Unit

Processus d'exécution

Format d'instruction

Les instructions de simpleRTJ représentent un sous ensemble de Java ISA [43]. Ils peuvent être composés de deux ou trois octets mais la majorité des instructions est codée sur un octet. Le format des instructions traitées est représenté sous l'une de ces formes :

1. code
2. code index
3. code index1 index2
4. code donnée
5. code donnée1 donnée2

Code	représente le byte code de l'instruction
Index	représente l'index dans la table de constante.
Donnée	représente une valeur

Système d'exécution dans simpleRTJ

simpleRTJ possède un système d'exécution, responsable de l'exécution du byte code des méthodes des classes chargées. Contrairement à la VM standard où le système d'exécution peut être constitué d'un ensemble de mécanismes tels que l'interpréteur de byte code, un compilateur à la volée, une exécution native, etc., celui de simpleRTJ ne possède qu'un élément : l'interpréteur de byte code. Initialement, une instance de la VM est lancée et prêt à exécuter les classes pseudo-chargées.

L'exécution d'une application commence à partir du point d'entrée de l'application (méthode main d'une classe de celle-ci) pour la création du thread initial de l'application. Ensuite, les appels de méthodes sont éventuellement effectués donnant lieu à la création de frame dans la pile de frame du thread correspondant à l'application. Exécuter une application passe donc par l'interprétation des byte codes de méthodes appelées, appartenant à des classes éventuellement différentes, qui représentent une interprétation croisée (*cross-interpretation*) des méthodes des classes. Le processus d'interprétation de byte code des méthodes passe par la recherche d'opérandes en passant par la table de constante, la table de variables locales ou encore la pile d'opérandes, tout dépend du type d'instruction à exécuter.

L'exécution des instructions dans la VM se présente comme un schéma assez complexe, passant par les quatre éléments suivants :

1. table de constante,
2. pile d'opérande et variable locales d'une frame donnée du thread de l'application correspondante,
3. tas de la machine virtuelle,
4. et vérification de type.

L'exécution dépend du type d'instruction. Certaines instructions peuvent avoir un impact sur la pile d'opérande (*dup*, *pop*, *swap*), nécessiter une indirection vers la table de constante (*invokevirtual* index, *new* index), ou opérer sur les variables locales de la frame courante (*iload*, *istore*, etc). simpleRTJ maintient le concept de redondance d'instruction que l'on a dans la Java standard, en

[43] Instruction Set Architecture

ce sens où l'on peut avoir plusieurs instructions qui peuvent avoir le même comportement sur la pile et sur les valeurs. En plus, il assure un contrôle de type à travers la table des champs associé à chaque classe dans le fichier binaire. Par exemple, pour des instructions manipulant un champ d'un objet d'une classe (*putfield*, *getfield*), simpleRTJ vérifie la cohérence du type du champ avec celui fourni par la sémantique de l'opcode de l'instruction, ceci grâce à la table de champs de la classe par contre, il ne gère pas une pile de méthodes natives, en effet il ne possède pas de JNI[44].

simpleRTJ diffère de la machine virtuelle Standard principalement au niveau de l'architecture des composants, et au niveau du processus de chargement. Dans la suite, nous allons comparer l'architecture et le processus de chargement d'une VM standard à ceux de simpleRTJ. La figure 5.1 présente l'architecture.

simpleRTJ présente une structure plus légère que celle de la VM Standard, en effet, étant destinée aux objets à faible empreintes mémoires et à contraintes processeurs, elle se présente comme une VM débarrassée d'un certain nombre de structures de données telles que la pile de méthodes natives (car ne possédant pas de JNI), de la zone de méthode (puisque utilisant directement le fichier binaire chargé dans la mémoire persistante) et de certaines fonctionnalités. Cependant, il possède de structures de données complémentaires telles que la table de thread, la table de références, etc., permettant de réduire la complexité dans la gestion des threads et des instances d'objets.

Structure de données
Type de données
simpleRTJ manipule deux catégories de types :

1. les types primitifs signés et non signés (uint8, uint32, etc.), le type booléen étant manipulé comme un type entier,
2. et le type référence permettant de stocker la référence à une instance d'une classe, à un tableau.

On peut noter que supporte aussi les types composés tels que les tableaux de type primitifs ou les tableaux de référence.

Stockage des données
simpleRTJ utilise deux grands espaces de stockage : une mémoire volatile dans laquelle l'exécution est effectuée, et une mémoire persistance où sont stockés le code de la VM, le code de démarrage, les applications, le système de fichiers, etc. Les espaces de stockage permettent donc de stocker d'une part les structures de données d'exécution et d'autre part les données persistantes nécessaires lors l'exécution.

La structure générale peut être comme indiqué dans la figure 5.1

[44] Java Native Interface

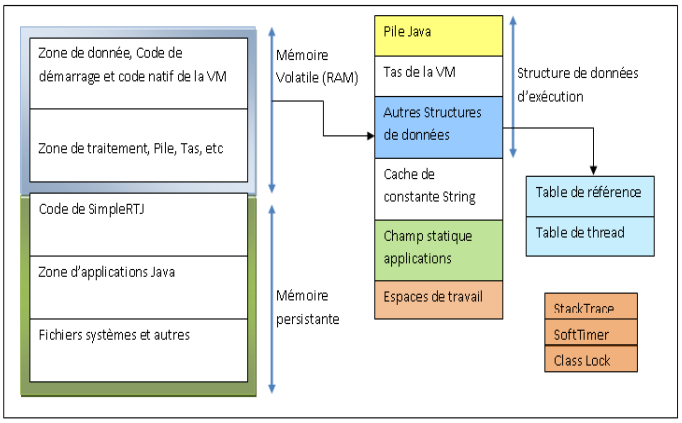

Fig. 5.1 – Structure générale de simpleRTJ

Représentation de la VM en mémoire volatile
Elle est structurée en deux zones : la zone de données propre à la VM : code de démarrage, code natif, etc. et la zone de traitement (utilisée lors de l'exécution).

La zone de traitement est constituée d'un ensemble de blocs parmi lesquels se trouve le tas, la pile Java, et des emplacements pour d'autres structures de données relatives au fonctionnement interne de la VM décrits dans les paragraphes suivants.

La pile Java. Cette zone contient les piles de frame de tous les threads en cours d'exécution dans la VM. En effet, à chaque thread est associée une pile de thread, et à la création du thread, cette pile est représentée par l'ajout d'un nouvel élément dans une structure de données appropriée appelée table de thread. A chaque appel de méthode, un nouveau frame est créé et empilé sur la pile Java du thread correspondant. Dans simpleRTJ, la pile Java est représentée par une liste doublement chainée de frames. Ainsi, les frames d'un thread sont chainées entre elles grâces aux adresses contenues dans les champs suivant et précédent de la structure de donnée associée à la frame. Elle est illustrée par la figure 5.2.

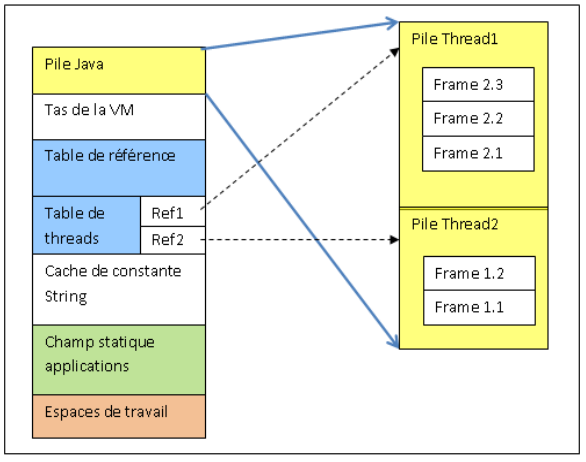

Fig. 5.2 – Pile Java

Chaque frame (figure 5.3) est constituée d'un ensemble de bloc associé aux arguments, aux variables locales, et aux opérandes des instructions. Seules les données de types primitifs et les références aux objets y sont stockées.

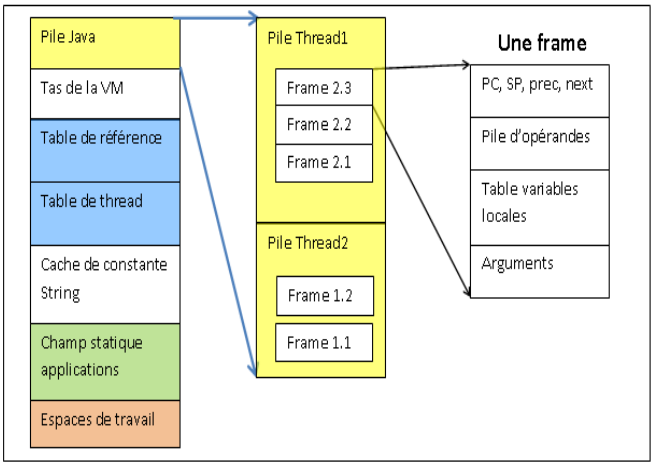

Fig. 5.3 – Vue sur une frame

Le tas . Chaque instance de la VM possède un tas. C'est une zone partagée par tous les threads créés dans une instance de la VM. Il contient l'ensemble des instances d'objets crées par les threads. Il peut contenir à la fois des objets statiques, objets dynamiques et les tableaux. Ces objets sont crées durant l'exécution ; et à chaque instanciation, une référence est générée et stockée dans une structure de donnée appropriée appelée table de référence. Ces objets deviennent par la suite accessibles uniquement par l'intermédiaire de ces références. Et à partir de la table de référence, on peut déterminer les classes associées à ces objets. Il est illustré par la figure suivante 5.4.

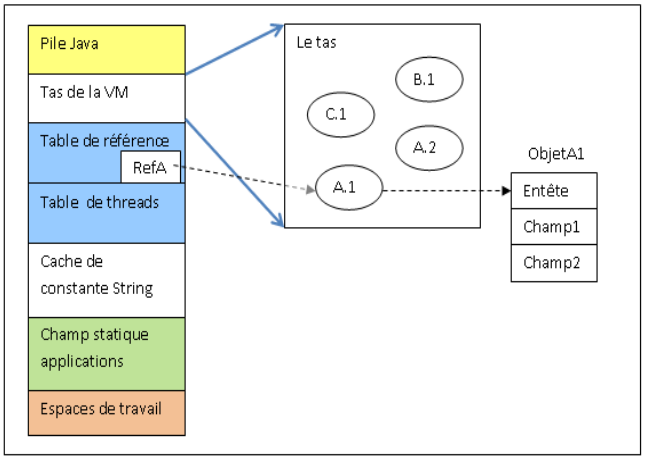

Fig. 5.4 – Vue sur le tas de la VM

Les autres structures de données de la VM

simpleRTJ s'appuie sur un ensemble de structure de données pour gérer principalement le contenu du tas et de la pile de thread. On peut avoir notamment :

1. La table de référence : permettant de stocker toutes les références aux objets crées dans le tas de la VM. Et partant, dans les frames de la pile de thread et dans les tableaux d'objets, uniquement ces références sont utilisées pour indexer l'objet dans le tas.
2. La table de thread : elle permet de stocker les informations relatives à un thread donné notamment, le numéro du thread, le frame de tête, la référence à son objet, ses drapeaux, les locks posés sur celui ci, etc.

Dans simpleRTJ, un thread est représenté par une liste doublement chainée d'un ensemble de frame.

– Champs statique des applications : espace où sont stockés les champs statiques des applications,
– Cache des constantes de chaines,
– Une zone de sauvegarde des registres manipulés par la VM.

Les registres sont utilisés par la VM lors de l'exécution des instructions du byte code (compteur de programme, le pointeur de pile, etc.)

Représentation de la VM en mémoire persistante Il s'agit principalement du code de simpleRTJ, des vecteurs d'interruptions, du code des applications (définition des classes, des interfaces, des méthodes, des constructeurs, etc.).

5.2 Prototype

5.2.1 EmbedDSU : Implémentation off-card

Il s'agit de déterminer l'ensemble des éléments à enregistrer dans le fichier de DIFF, devant permettre - une fois, transféré à l'intérieur de la carte – d'effectuer le transfert d'état du contexte d'exécution de l'ancienne version pour obtenir le nouveau contexte correspondant à la nouvelle version et continuer l'exécution. Pour cela, il est nécessaire de comprendre la structure d'une classe Java, d'un fichier classe (*classfile*) et présenter l'impact d'une modification sur le fichier binaire Java (*class file*).

Générateur de DIFF

Le générateur de DIFF est basé sur un CFG modifié.

L'algorithme générateur de DIFF se base ensuite sur les CFGs construits correspondant à l'ancienne et la nouvelle version de la classe. Les deux CFGs sont comparés et un fichier de DIFF est généré selon l'algorithme décrit dans la section 4.2.1

Le contenu du fichier de DIFF est constitué de modifications détectées sur :

– les table de constantes, notamment les entrées ajoutées, supprimées ou modifiées,
– les droits d'accès de la classe,
– les interfaces de la classe,
– les membres données (les champs) de la classe (au niveau du type, des droits d'accès, des valeurs initiales et des index dans la table de constante),

– les membres méthodes de la classe (au niveau des droits d'accès, de la signature, des variables locales et des instructions),
– les exceptions (modifiées, ajoutées ou supprimées)

Ce fichier de DIFF est exprimé dans un langage définit à cet effet, dont la grammaire est présentée en annexe D

Fichier de DIFF et fonction de transfert
Le générateur de DIFF possède un module permettant de gérer les fonctions de transfert. Une fois, les modifications déterminées au niveau des classes, le générateur de DIFF interprète les fonctions de transfert et rajoute les instructions nécessaires pour le transfert d'état dans le fichier de DIFF.

EmbedDSU gère deux types de fonctions de transfert :
– Les fonctions de transfert fournies par le programmeur ;
– Et les fonctions de transfert par défaut qui sont générées automatiquement en on-card.

Les fonctions de transfert fournies par le programmeur ne sont pas copiées dans le fichier de DIFF. En effet, ces fonctions de transfert sont analysées et adaptées avant d'être ajoutées dans le fichier de DIFF.

Processus de modification des fonctions de transfert
Dans EmbedDSU, pour chaque classe modifiée, le programmeur fournit éventuellement une classe associée correspondant à la fonction de transfert. Cette classe comporte une méthode statique contenant les instructions de transfert. A l'intérieur de la carte, l'objectif est de faire exécuter cette fonction de transfert comme une méthode de nouvelle version de la classe. Cette méthode est capable de prendre en entrée des instances de la nouvelle version et de les modifier afin d'obtenir celles correspondants à la nouvelle version. Pour cela, à l'extérieur de la carte, il est nécessaire de modifier les paramètres des instructions des fonctions de transfert fournis par le programmeur pour qu'elles correspondent à celles d'une méthode de la nouvelle classe. Il s'agit :

1. De modifier les index des tables de constantes des fonctions de transfert en leur valeur correspondantes dans le table de constante de la nouvelle classe,
2. De modifier les index des méthodes et des champs des fonctions de transfert pour obtenir ceux correspondant aux entrées de la table de méthode et la table de champs de la nouvelle version de la classe. Une fois, les instructions nécessaire modifiées, celle-ci sont enregistrées dans le fichier de DIFF à transférer dans la carte.

Transformation DIFF DSL vers DIFF binaire

L'idée est de réduire la taille de la DIFF à envoyer en on-card. Pour cela, un outil a été mis en place. Cet outil permet d'interpréter le fichier de DIFF en DSL afin de générer un fichier de DIFF binaire. Ce fichier de DIFF binaire est représenté dans un format prédéfini. A la suite, un outil de vérification du fichier de DIFF binaire a été aussi mis en place. L'objectif est de s'assurer que (1) le format de DIFF binaire a été bien respecté ; (2) et que les instructions présentes dans le fichier de DIFF sont valides et sont supportées par la Java Card. Le format du fichier de DIFF est présenté de façon plus détaillée dans l'annexe A. Une fois le fichier de DIFF binaire obtenu, EmbedDSU prévoit un module de vérification de ce fichier avant son transfert on-card dont le processus est détaillée dans l'annexe B

Une fois, le fichier de DIFF binaire obtenu et vérifié, celui-ci est transmit dans la carte pour effectuer la mise à jour proprement dite. Dans la suite, nous étayerons l'implémentation d'EmbedDSU partie on-card.

5.2.2 EmbedDSU : Implémentation on-card

Il s'agit de présenter l'implémentation du processus de mise à jour à l'intérieur de la carte. Ce processus de mise à jour permet de réaliser le transfert d'état d'exécution de l'ancienne classe ou l'ancienne application vers un nouvel état correspondant à la nouvelle classe. Dans la suite, nous présenterons ce qui constitue un état d'exécution d'une classe et ensuite nous expliquerons le processus de transfert d'état.

Processus général
L'implémentation à l'intérieur de la carte peut être résumée comme suit :

1. Modification de simpleRTJ
   (a) Introspection des structures de données de la VM,
   (b) Détection des points de mise à jour.
2. Implémentation du module interprétation
   (a) Authentification
   (b) Interprétation de la DIFF
3. Implémentation du module de mise à jour
   (a) Mise à jour du code,
   (b) Transfert de l'état d'exécution d'une classe vers le nouvel état,
   (c) Mise à jour des références dans les autres classes de l'application,
4. Assurer le *roll-back* en cas d'échec ou de non atomicité,
5. Continuer l'exécution de la classe après la mise à jour.

Les sections suivantes présentent une description de l'implémentation des modules on-card.

Module de recherche de point sûr

EmbedDSU détecte un point sûr lorsqu'il n'y a plus de méthode restreinte dans la pile de thread. Nous avons modifié la machine virtuelle pour qu'elle puisse fonctionner sous trois modes :

1. le mode normal (avant et après la mise à jour),
2. le mode d'attente d'un point sûr ;
3. et le mode de mise à jour.

La détection d'un point sûr consiste à détecter le moment opportun durant lequel la mise à jour peut être effectuée en préservant un minimum de cohérence du système. Un point sûr est obtenu lorsqu'aucun frame dans la pile Java n'appartient à une méthode modifiée dans la classe à mettre à jour. Il s'agit de parcourir la pile de thread, et de rechercher l'ensemble de frame appartenant aux méthodes modifiées de la classe à mettre à jour.

1. Si cet ensemble est vide, alors un point sûr a été trouvé, sinon attendre. Attendre d'atteindre un point sûr consiste à se placer dans un des modes de la VM qui est le mode d'attente, de placer un compteur de frame de méthode modifiée initialisé au nombre d'élément de l'ensemble précédemment obtenu. Dès que l'exécution d'une méthode modifiée est terminée, ce compteur est décrémenté, quand il atteint la valeur nulle, le point sûr est atteint et l'on peut dès lors passer au mode de mise à jour.
2. Si au cours du mode attente, un thread fait un appel à une méthode modifiée alors :
   - Si ce thread contient déjà dans sa pile de frame, une méthode modifiée, pour éviter un inter blocage, ce thread continue son exécution,
   - Si ce thread ne contient pas déjà dans sa pile de thread, une méthode modifiée alors il est bloqué, ce qui permet de réduire le temps d'attente et donc d'utiliser la nouvelle version de la méthode, une fois la mise à jour effectuée

Mise à jour du code

Une classe appartient à une application, elle même constituée d'un ensemble de classe et des fichiers de métadonnées. Sous simpleRTJ, une application est représentée par un fichier appelé *binFile* contenant l'ensemble de classe de l'application obtenu par compilation statique et des métadonnées. Grâce à une édition de liens préalablement réalisée, les adresses apparaissant dans le *binFile* représente des adresses relatives et donc des déplacements par rapport à l'adresse initial (0x0). Structure d'un *binFile* : Le schéma suivant 5.5 résume la structure d'une application pour simpleRTJ.

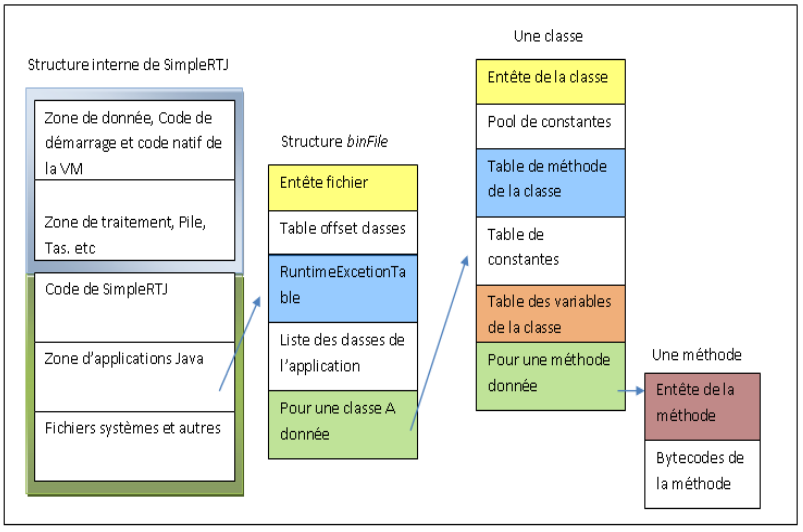

Fig. 5.5 – Structure de données d'une application sous simpleRTJ

Le format de fichier d'une application simpleRTJ est constitué d'un ensemble de sections de taille variable, dans un ordre prédéfini suivant

1. Entête du fichier

2. Table des offsets de classe
3. Table des offsets des classes d'exceptions
4. Les classes, et pour chaque classe
   - Entête de classe
   - Table de constante
   - Table de méthode
   - Table de constante
   - Table de champs
   - Les méthodes et pour chaque méthode
     (a) Entête de méthodes
     (b) Bytecode

Ce format de fichier est présenté de façon détaillée dans C.

Processus de mise à jour
La mise à jour du code de la classe passe par l'initialisation des structures de données nécessaires à la mise à jour à partir du fichier de DIFF, la relocation de la classe à mettre à jour, ensuite s'ensuit la mise à jour des dépendances à la classe dans le *binFile*.

Initialisation des structures de données
A partir du fichier de DIFF, il s'agit de ressortir l'ensemble d'information nécessaires à la mise à jour de la classe, donc à la mise à jour de l'entête de la classe, de la table des champs, de la table des méthodes, des entêtes des méthodes, des byte code des méthodes et nécessaire à la mise à jour des dépendances.

```
static   bool initData_Structures () {
. . .
         init_infos_field_table() ;
         init_infos_methods_table() ;
         init_new_length_methods() ;
         init_infos_constantPool_table();
}
```

relocation et mise à jour de la classe
La mise à jour du code de la classe consiste en la recopie en modification des informations. La relocation d'une classe va consister en la recopie en modification de l'entête de la classe, de la table de constante, de la table de méthode, de la table des champs, et des méthodes elle-même, ceci vers un autre espace mémoire.

Mise à jour des classes dépendantes
Après relocation de la classe, il est nécessaire de mettre à jour les informations relatives à cette classe dans les autres classes de l'application. Il peut s'agir de :

1. Pool de constante des autres classes.
   En effet, étant donné que toutes les classes possèdent une table de constantes contenant les adresses relatives de l'ensemble de variables et méthodes utilisés, ces dernières peuvent être

des méthodes et variables de la classe relocalisée. Dès lors, il est nécessaire de mettre à jour l'ensemble des tables de constantes des autres classes de l'application.

La mise à jour des dépendances consiste à partir de la table des offsets des classes, pour chaque classe, de parcourir la table de constante et pour chaque entrée, vérifier si l'adresse obtenue peut être une adresse de la zone de la classe mise à jour (relocalisée), si oui, alors mettre à jour l'entrée en modifiant l'adresse pour qu'elle pointe sur l'adresse correspondante dans la zone de la nouvelle classe. Cependant, ces méthodes ou variables de la classe à mettre à jour, présents dans les tables de constantes des autres classes, peuvent être supprimées dans la nouvelle version de la classe. Il est nécessaire dans ces cas, de mettre aussi à jour toute la classe associée, afin de conserver la cohérence du système.

2. Mettre à jour les métadonnées de l'application. Il peut s'agir de :

   – L'entête de fichier du *binFile* : par exemple si la taille de l'application a changé dû à la modification des tailles des méthodes de la classe mise à jour, ou si l'adresse du main a été modifiée ;
   – La table des offsets de classe de l'application, pour mettre à jour l'adresse de l'ancienne classe pour qu'elle pointe sur la nouvelle classe

3. Mise à jour des PCs dans les frames.
   Après la mise à jour du *binFile*, il est nécessaire de mettre à jour les compteurs de programmes (PC[45]) dans la pile Java de la VM. En effet, les méthodes non modifiées de la classe à mettre à jour, peuvent être présent sur la pile d'exécution. Et dont il est nécessaire de mettre à jour le PC du frame pour qu'il pointe sur l'instruction correspondante dans la nouvelle version de la classe.

[45] Program counter

```
static bool update_code() {
. . .
// Initialisation des structures de données
initDatataStructures() ;
. . . // Mise à jour de l'entête du binFile
update_HeaderFile() ;
. . .
// Mise à jour de la table des classes
update_ClassOffsetTable() ;
. . .
// Mise à jour de l'entête de la classe
update_ClassHeader ( ) ;
. . .
// Mise à jour de la table des constantes de la classe
update_ClassCP ( ) ;
. . .
// Mise à jour de la table de méthode update_MethodTable ( ) ;
. . .
// Mise à jour de la table des valeurs de constantes
update_constantTable ( ) ;
. . .
// Mise à jour de la table des champs
update_FieldTable ( ) ;
}
```

Après la mise à jour du code, s'ensuit la mise à jour des instances de la classe dans le tas et des frames dans la pile Java de la VM.

Mise à jour des instances

Le processus général de mise à jour d'instances consiste à parcourir le tas de la VM à la recherche de toutes les instances de la classe à mettre à jour et pour chaque instance, recopié en modifiant pour obtenir une instance cohérente à la nouvelle version de la classe.

Ce processus peut être subdivisé en plusieurs étapes :

- La recherche des instances de la classe dans le tas de la machine virtuelle,
- La création de la nouvelle instance,
- L'initialisation de la nouvelle instance pour qu'elle soit cohérente avec la nouvelle version de la classe.

1. **Recherche des instances**
   La fonction de recherche se présente comme une variante de l'algorithme de ramasse-miettes. En effet, elle parcourt le tas de la VM à la recherche de toutes les instances vivantes d'une classe à mettre à jour. Elle est basée sur l'ensemble des racines de persistances (pile, variables statiques, variables de classes, etc).

La fonction utilise une technique dite *mark and update*. Au fur et à mesure qu'on atteint les objets de la classe, ils sont directement mis à jour et un drapeau est ajouté à la structure de donnée de l'objet modifié, ce qui permet de ne plus retraiter ces instances lorsqu'ils seront atteints à partir d'une autre racine de persistance.

2. **Modification des instances**
   La mise à jour des instances trouvées passe par une allocation mémoire de l'espace nécessaire pour le nouvel objet. La taille de cet espace mémoire est calculée en fonction du nombre de champs ajoutées, ou supprimées. Une nouvelle référence de l'objet est obtenue, ce qui permet de mettre à jour, les références dans la pile de thread et les autres structures de données de la VM. Après allocation de l'espace mémoire, s'ensuit l'initialisation.
3. Elle s'effectue en deux étapes :
   - les valeurs des champs non modifiés sont recopiées dans le nouvel objet,
   - par contre, pour les champs ajoutés, on peut avoir deux types de situation :
     (a) Soit la valeur initiale est fournie et n'est rien d'autre qu'un entier à recopier dans le champ approprié de l'objet créé,
     (b) Soit la valeur initiale n'est pas fournie et cette valeur doit être initialisée par une fonction ou une méthode de la classe ou d'une autre classe ce qui est réalise à l'aide des fonctions de transfert.
4. Dans simpleRTJ, après chaque allocation mémoire, une référence est créée et la valeur de cette référence est sauvegardée dans la table de référence de la VM. Ainsi, tout objet d'instance créé possède une référence dans la table de référence.
5. Afin de pouvoir effectuer, un *roll-back* simple en cas d'échec, il est nécessaire d'avoir à la fois l'ancienne référence de l'objet et la nouvelle référence du nouvel objet créé. Pour cela, dans un premier temps, l'ajout d'un champ supplémentaire à la structure de donnée relative aux entrées de la table de références s'est avéré nécessaire. Ce dernier champ contient la référence à l'ancien objet de la classe.

Mise à jour des frames

Après la mise à jour des instances et de la table de référence, la mise à jour des frames est incontournable. En effet, dans la pile d'opérande et la table de variable locale des frames dans la pile Java de la VM, peut se trouver des références aux objets ou instances précédemment mis à jour. Il convient donc de mettre à jour toutes ces références pour qu'elles pointent sur les nouveaux objets correspondant à la nouvelle classe. Pour cela, l'algorithme est le suivant : parcourir la table de thread, pour chaque thread, parcourir l'ensemble de ses frames et pour chaque frame, parcourir sa table de variables locales et sa pile d'opérandes, si on trouve une référence appartenant à la classe à mettre à jour, alors rechercher cette ancienne référence dans la table de référence et mettre à jour l'entrée dans le frame associé. Pour chaque thread mis à jour, positionner le drapeau de mise à jour, ce qui permet en cas de *roll-back* de localiser directement l'ensemble des threads affectés par la mise à jour des références.

Mise à jour des structures de données

Pour la mise à jour des structures de données, il s'agit de la mise à jour :

1. de la table des locks sur les classes, En effet, après le processus de mise à jour, les threads bloqués durant l'attente doivent être débloqués enfin de continuer leur exécution avec les nouvelles versions des méthodes ;
2. des références dans la table des champs statiques de l'application. Les champs statiques contenant des valeurs de références vers les instances de classe mises à jour, doivent pointer sur les nouvelles instances d'objets ;
3. des références dans la table de référence ;
4. et des drapeaux de mise à jour dans la table de thread.

Après la mise à jour du code, des instances et des structures de données de la VM, le GC est appelé afin de libérer la mémoire et ensuite la machine virtuelle passe en mode standard pour continuer l'exécution avec la nouvelle version du code et les nouvelles versions des objets.

Le prototype, une fois mis en place, un ensemble d'évaluations ont été faites afin de vérifier l'adaptation de l'approche à un contexte de l'embarqué, donc à un environnement à forte contraintes de ressources.

5.3 Évaluation

5.3.1 Plateforme d'évaluation

L'évaluation s'est déroulée tout d'abord sur un poste de travail ordinaire ensuite sur la carte d'évaluation d'*Embest* AT91EB40A. EmbedDSU est basée sur une machine virtuelle Java simpleRTJ.

La plateforme Embest
Il s'agit de la carte d'évaluation AT91 EB40A sur laquelle la machine virtuelle étendu a été portée. Cette carte est équipée d'un processeur AT91 R4008 de type ARM7TDMI. Elle a une vitesse d'horloge interne par défaut de 32,76 *MHZ*, pouvant être au choix divisée par 2, par 4 ou par 8 en fonctions de besoins en terme de ressources processeurs. Elle dispose d'une mémoire de 256 *kbits* de RAM et de 2 *Mb* de Flash. Elle peut fonctionner au choix avec un adressage mémoire de 16 ou 32 bits. Elle offre aussi la possibilité d'ajuster la vitesse de fonctionnement de mémoire afin de simuler les mémoires lentes. Ses caractéristiques de mémoire et de processeur sont assez proches d'une carte à puce.

5.3.2 Méthode d'évaluation

Les raisons pour lesquelles l'évaluation a été réalisée sur une carte de test autre que la carte à puce Java Card, sont les suivantes :

- la Java Card n'offre pas d'accès à son système d'exploitation (VM) ;
- les Java Cards disponibles sur le marché ne possèdent pas un système de flashage ;
- et il n'est pas toujours facile de trouver une collaboration avec un encarteur, en plus le coût de fabrication n'est pas toujours accessible.

Nous avons donc opté pour une évaluation sur un plateforme de test très proche des caractéristiques d'une carte à puce. Cette évaluation consiste à déterminer :

1. le coût du fichier de DIFF comparé à celui du fichier de la nouvelle version de la classe ;
2. le coût de l'occupation mémoire EEPROM de chaque module du système EmbedDSU ;
3. le surcoût en mémoire RAM lors de la mise à jour ;
4. les performances lors du processus de mise à jour (mise à jour du code, mise à jour des instances, recherche de point sûr).

5.3.3 Coût du fichier de DIFF

La mise en place du fichier de DIFF binaire permet de réduire considérablement la taille du fichier à transférer dans la carte et ainsi de réduire le temps de transfert du patch de mise à jour. En effet, un système de DSU sans fichier de DIFF, prend généralement en entrée :
- le fichier de la nouvelle version de la classe ;
- et les fichiers de fonctions de transfert.

Il peut arriver des cas où les modifications concernent uniquement quelques instructions dans une méthode donnée ou alors la modification du type d'un champ. Par exemple, pour le bogue des cartes de l'année 2010, la mise à jour ne concernait que quelques instructions (voir une instruction de test). Dans ces cas, le transfert du fichier de la nouvelle version de la classe et du fichier de fonction de transfert représente un coût considérable. Sachant que dans le monde de la carte, l'utilisateur dispose souvent de quelques secondes pour effectuer son opération (retraits au DAB[46], badges, etc) et que le transfert des informations pour la mise à jour, dans le meilleur des cas, devrait s'effectuer durant l'opération. Même si la communication sur les cartes basées sur Java Card 3.0 édition connecté peut se faire à l'aide du protocole HTTP, la majorité des Java Card, actuellement utilise une communication basée sur le protocole APDU.

En plus du temps de transfert, l'envoie du fichier de la nouvelle version de la classe et du fichier de fonction de transfert peut s'avérer extrêmement coûteuse dans certains cas, en terme d'occupation mémoire, ceci en fonction du type de modification (Ajout de méthode, ajout de champs, ajout d'instructions dans une méthode donnée, suppression d'instructions, etc.).

La mise en place du fichier de DIFF permet de réduire considérablement le coût de transfert et le coût de l'empreinte mémoire à utiliser pour stocker les informations de patch. En effet, le fichier de DIFF permet une réduction de 30% à 57% de la taille du fichier de la nouvelle version cumulée à celui de la fonction de transfert, tout dépend du type de modification.

Le type de modification peut permettre de déterminer l'impact de celle-ci dans la nouvelle version. Par exemple, l'ajout d'un champ impacte :

1. la table de constante, et la modification de la table va dépendre du moment de son utilisation. En effet, la plupart des compilateurs Java, insère des entrées dans la table de constante au fur et à mesure de leur découverte dans le fichier de byte code, et donc dans l'ordre de leur utilisation.
2. et les instructions des méthodes. En effet, grâce à l'élimination de code mort dans le processus de compilation, un champ est considéré comme ajouté, lorsqu'il est réellement utilisé soit par le constructeur, soit par l'une des méthodes de la classe. Ce qui implique, de nouvelles instructions liées à l'ajout du champ.

[46] Distributeurs Automatiques de Billets

A partir de quelques échantillons de classes et fonctions de transfert, on peut en ressortir le tableau 5.1 qui résume le meilleur des cas et le pire des cas observé.

	Taille NC+FT (Octet)	Taille DIFF (Octet)	Pourcentage (%)	Type de modification
Pire des cas	2030	1241	38,86	Instr
Meilleur des cas	993	213	58,40	Champs

Tab. 5.1 – Coût du fichier de DIFF

Avec :

NC	Nouvelle classe
FT	Fonction de transfert
Instr	Instruction

Les observations, ont montré que le fichier de DIFF peut permettre de réduire en moyenne, de moitié la taille des informations à envoyer dans la carte ce qui permet de réduire de moitié l'empreinte mémoire du patch.

Le coût du fichier de DIFF est extrêmement lié au type de modification. Par exemple, s'il n'y a pas de modification au niveau des champs d'une classe ceci implique par exemple qu'il n'y a pas de fonction de transfert, influant ainsi la taille du fichier de DIFF.

5.3.4 Occupation de la mémoire EEPROM

Il s'agit de déterminer pour chaque module d'EmbedDSU, son coût en occupation mémoire en EEPROM.

Modules	Taille(ko)
Module interprétation DIFF	13 ko
Module de détection de point sur	12 ko
Module de mise à jour	total de 29 ko
* Mise à jour des instances	9 ko
* Mise à jour du code	13 ko
* Mise à jour des frames	7 ko
Module de restauration	13 ko
Total	**67 *ko***

Tab. 5.2 – EmbedDSU : Occupation mémoire EEPROM

A côté de l'occupation mémoire, il est nécessaire comparer la taille de la VM initiale à celle obtenue après insertion des modules de DSU, qui passe de 252 ko à 319 ko.

On constate une augmentation de ~7 %, ce qui reste acceptable dans le monde de la carte et sachant que le prototype actuel d'EmbedDSU peut être amélioré.

5.3.5 Surcoût en mémoire RAM

Le surcoût en mémoire RAM est déterminé principalement par les structures de données de mise à jour utilisées lors du processus de DSU en on-card. Les structures de données de mise à

jour permettent en particulier de mettre à jour les instances et la table de constante de la classe. L'initialisation de ces structures est effectuée par l'interpréteur du fichier de DIFF. Ces structures de données de mise à jour spécifient :

1. les méthodes ajoutées, les méthodes modifiées (leurs adresses en mémoire, leurs nouvelles tailles, etc) et les méthodes supprimées (les adresses) ;
2. les champs ajoutés et supprimés (les index dans la table de constante) ;
3. les entrées ajoutées et supprimés de la table de constantes ;
4. les nouvelles références des instances pour le processus de restauration.

La structure de donnée qui contient la sauvegarde des nouvelles références de la classe constitue un élément important dans le calcul du surcoût mémoire. Cependant, sa taille dépend du nombre d'instance de la classe mis à jour présent dans le tas de la machine virtuelle.

Ce surcoût en mémoire RAM varie aussi en fonction du type de modification.

5.3.6 Performances

L'algorithme de mise à jour des instances est basée sur l'algorithme du ramasse-miettes que nous avons modifié en vue d'obtenir les instances de la classe à mettre à jour présentes dans le tas de la machine virtuelle. Pour déterminer la performance, nous avons choisi trois types de d'opération sur une instance :

- l'ajout des champs ;
- la suppression des champs ;
- et le réarrangement des champs (instances avec les mêmes champs mais sous un autre ordre).

Ensuite, nous avons comparé le coût de la mise à jour des instances, pour chacune de ces types d'opérations, au coût du GC standard.

Considérons les versions d'instances présentées à la table suivante.

Original C	Re-arrangement	Ajout de champs	Suppression de champ
class C {	class C' {	class C' {	class C' {
int var1 ;	int var2 ;	*short var5 ;*	int var1 ;
int var2 ;	short var3 ;	int var1 ;	short var3 ;
int var3 ;	int var1 ;	*int var6 ;*	}
Object var4 ;	Object var4 ;	int var2 ;	
}	}	short var3 ;	
		Object var4 ; }	

Tab. 5.3 – Versions des instances utilisées

Nous avons constaté que l'ajout de champ est l'opération la plus coûteuse comparée aux autres types de modifications des instances. En effet, la mise en œuvre de l'opération d'ajout de champ à une instance consiste à :

1. rechercher le nouvel espace mémoire avec la taille appropriée pour contenir la nouvelle instance ;
2. copier les valeurs des champs non modifiés pour initialisé les champs correspondant dans la nouvelle version ;

3. et initialiser les champs ajoutés à partir des fonctions de transfert (générées automatiquement ou fournies par le programmeur), ce qui rajoute le coût de l'exécution de celles-ci au coût de l'opération.

L'opération de réarrangement est aussi coûteuse, cela nécessite de créer un nouvel objet et de recopier les valeurs des champs de l'ancien objet dans la nouvelle version de l'instance. Cependant, il n'y a pas besoin de récupérer les valeurs d'initialisation dans le fichier de DIFF ou d'exécuter des fonctions de transfert. Ce qui rend l'opération moins coûteuse que celle de l'ajout.

L'opération de suppression est la moins coûteuse que celle de l'ajout. Au cours de l'opération de suppression, nous n'avons pas de coût supplémentaire dû à l'exécution des fonctions de transfert (pour l'initialisation), ce qui réduit considérablement le coût. En effet, l'ancien objet est copié en éliminant les champs supprimés pour obtenir la nouvelle version de l'instance.

	T_a	Temps opération ajout
Ainsi, soit :	T_s	Temps opération suppression
	T_{ar}	Temps opération arrangement

On peut en déduire que :

$$T_s < T_{ar} < T_a$$

Le temps de mise à jour des instances varient selon le type de modification. Cependant, on observe que les modifications de type ajout, sont les plus coûteuses.

Le coût en terme de performance lors du processus de mise à jour dépend aussi du temps de recherche d'un point sûr. Le coût de la recherche d'un point sûr dépend du nombre de thread en cours d'exécution, du nombre de frames présentes dans la pile Java et surtout du nombre de méthodes modifiées (à mettre à jour), présentes sur la pile d'exécution (méthodes restreintes).

	C	la classe à mettre à jour
	Cr_t	le coût de recherche d'un point sûr à un instant t
	NbMr	le nombre de méthode restreintes
Soit :	Nbm	le nombre des autres méthodes présentes sur la pile d'exécution
	Pm	la probabilité de terminer l'exécution d'une méthode quelconque autre qu'une méthode restreinte
	Tm	le temps moyen d'exécution d'une méthode

$$Cr_t = NbMr \times Tm + \sum_{k=1}^{Nbm}(Pm_k \times Tm)$$

5.4 Conclusion

Ce chapitre a permis de présenter la mise en œuvre de l'approche proposée et son évaluation. Nous avons montré dans un premier temps que l'utilisation de simpleRTJ répondait à nos besoins, car il s'agit d'une machine conçu pour l'embarqué fonctionnant sur des principes similaires à une implémentation de Java Card. Les quelques différences (comme l'absence de bibliothèque java dans la plateforme et la génération d'un fichier binaire contenant toutes les classes nécessaires) ne sont

pas significatives pour l'évaluation d'un mécanisme de DSU. Le fait que la représentation des classes, méthodes et byte code soient similaires à celle d'un fichier CAP[47] nous permet de manipuler et mettre à jour le même type de structure. Nous pensons ainsi avoir réussi à rendre notre évaluation proche d'une implémentation de notre mécanisme de DSU dans une vrai Java Card.

Nous montrons aussi, à base d'expérimentation, que le surcoût d'un tel mécanisme n'est pas élevé au regard des contraintes d'une carte à puce. Que ce soit dans la taille d'un fichier DIFF, de l'occupation en mémoire vive durant la mise à jour, de la vitesse des algorithmes de mise à jour et de recherche de point sûr, ou de la taille de l'EEPROM après mise à jour.

Ainsi, nous pensons avoir réussi à présenter une évaluation correcte présentant les qualités du mécanisme de mises à jour dynamique dans simpleRTJ.

[47] Converted APplet : Format de fichier d'une application dans la Java Card 3.0 édition classique

Quatrième partie

Perspectives et conclusion

Chapitre 6

Perspectives

Sommaire

Les principales perspectives de recherche qui apparaissent à l'issue de cette thèse concernent la réutilisabilité de notre travail à travers son adaptation pour des environnements embarqués notamment ceux basés sur OSGi, et la formalisation du processus de DSU afin de vérifier la sûreté de l'approche.

6.1 Adaptation EmbedDSU pour OSGi/OSGi-ME

6.1.1 Définition OSGi/OSGi-Me

OSGi[48] [XZW+03, Ni03, PF07] est une spécification ouverte pour la définition d'une plateforme de déploiement et d'exécution des services administrés à distance dans des environnements embarqués. OSGi implémente un modèle de composants dynamiques. Chaque composant peut être installé, arrêté, démarré, mis à jour et désinstallé de manière distante sans nécessiter de redémarrage. Chaque composant possède un cycle de vie dont la gestion est effectuée à travers une API en appliquant une politique de gestion des téléchargements distants. OSGi peut être utilisé dans des environnements embarqués tels que :

[48] Open Service Gateway Initiative

- les passerelles résidentielles/domotiques ;
- les véhicules de transport ;
- les appareils électroménagers ;
- les contrôles industriels ;
- la téléphonie mobile, etc.

OSGi-Me [BR10] est une version d'OSGi pour périphériques embarqués avec quelques améliorations comme la compatibilité avec Java ME[49] CLDC[50], la simplification de la technologie OSGi pour des simples besoins, la réduction de l'empreinte mémoire afin cibler les environnements plus contraints.

Un module OSGi est appelé un bundle. Une application peut être composée d'un ou plusieurs bundles. Un bundle peut exporter ou importer un ou plusieurs services. Il existe un répertoire de services encore appelé «*service registry*» qui permet aux bundles de détecter l'addition de nouveaux services, ou la suppression de services et de s'y adapter. L'architecture générale simplifiée d'OSGi peut être illustrée par la figure 6.1.

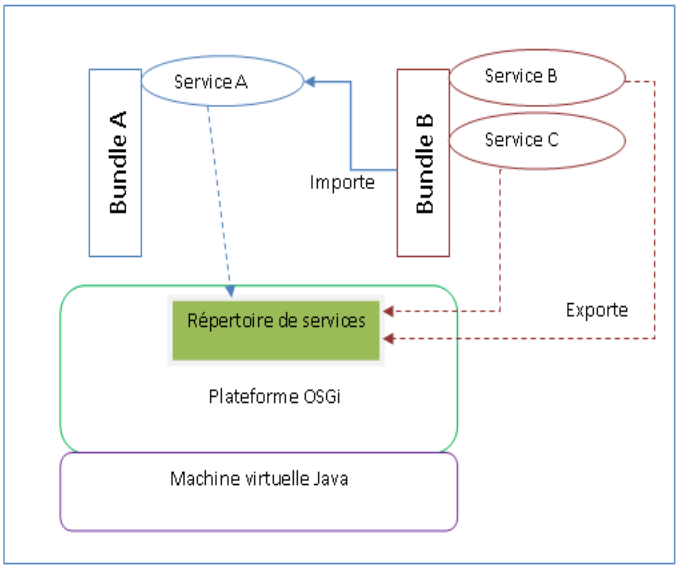

Fig. 6.1 – Architecture simplifiée d'OSGi

6.1.2 OSGi et DSU

Un bundle est un fichier archive (*.jar)* possédant un ensemble de classes et un fichier spécial appelé *Manifest.MF*. Ce fichier *Manifest.MF* contient toutes les méta-informations sur le bundle telles que :

- le nom du bundle ;
- la version ;
- la liste des services à exporter et à importer ;
- la version minimale de la JDK sur laquelle le bundle peut s'exécuter, etc.

[49] Java 2 Micro Edition
[50] Connected Limited Device Configuration

Au départ le bundle est chargé et est dans l'état installé. Une fois la résolution de dépendances réalisée, il passe dans l'état résolu. L'état transitoire entre l'état Résolu et l'état Actif est le démarrage (état Démarré). Lorsque le bundle est démarré avec succès, il passe à l'état Actif. Dans cet état, le bundle et les services qu'ils exposent sont disponibles pour les autres bundles. L'activation ou la désactivation d'un bundle résulte de la création ou la destruction d'une unité logique, matérialisée par l'instance d'une classe dans le bundle, appelée *bundle Activator*. Quand un bundle est activé, la plateforme OSGI appelle une méthode qui signale que le bundle est démarré. De même lorsqu'un bundle est arrêté, le bundle appelle une méthode de désactivation. Quand un bundle est actif il peut alors publier ses services, découvrir les services disponibles, ainsi que de se tenir informer des changements (évènements) qui surviennent. Ce cycle de vie d'un bundle peut être illustré par la figure 6.2

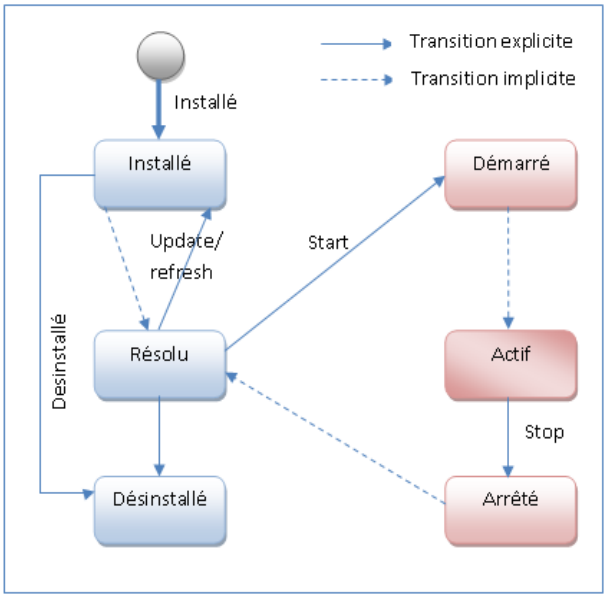

Fig. 6.2 – Cycle de vie d'un Bundle

En analysant ce cycle de vie, on remarque que la mise à jour d'un bundle consiste à désactiver et désinstaller l'ancienne version, à charger la nouvelle version, à l'installer et à l'activer. Les bundles peuvent êtres ajoutés et mis à jour de façon dynamique durant l'exécution et grâce au système d'événements supporté par la plateforme OSGi, les bundles se réadaptent automatiquement afin de préserver la cohérence du système et le répertoire de service est automatiquement mis à jour. On constate dans ce processus de mise à jour que la machine virtuelle n'est pas arrêté, la plateforme OSGi n'est pas arrêté mais le bundle est désactivé et désinstallé. L'arrêt du bundle implique la perte de l'état d'exécution du bundle lors de sa mise à jour. Cet arrêt implique aussi l'arrêt des services exportés et l'arrêt des autres bundles qui importent les services fournis par l'ancienne version du bundle.

6.1.3 Proposition d'une architecture de DSU pour OSGi

L'objectif est donc de proposer une solution permettant de préserver l'état système du bundle lors de sa mise à jour tout en permettant une mise à jour avec un niveau de granularité plus fin,

pour permettre la réduction du coût de la mise à jour. En effet, si une seule méthode d'une classe donnée du bundle est modifiée, il n'est pas nécessaire de mettre à jour tout le code du bundle correspondant. De plus, il n'est pas nécessaire de stopper tous les services même ceux offerts par les classes du bundle non impactées par la mise à jour.

L'approche consiste donc à descendre à un niveau de granularité plus bas qui est la classe d'un bundle. Ainsi, étant donné un bundle, sa mise à jour consiste en :

- la mise à jour des classes impactées par les modifications ;
- et à un transfert d'état de l'ancienne version du bundle vers la nouvelle obtenue à l'aide d'un patch.

Avec cette méthode, la mise à jour peut s'appliquer à une classe ou à un bundle (un ensemble de classes). L'approche que nous proposons est basée sur EmbedDSU. Il s'agit de modifier l'architecture d'EmbedDSU pour l'adapter à la plateforme OSGi. Le processus d'EmbedDSU étant basé sur une machine virtuelle Java, il s'agira plus précisément de modifier et d'étendre la machine virtuelle afin d'y ajouter les mécanismes de mise à jour dynamique qui tient compte du framework OSGi sur lequel s'exécute le bundle à mettre à jour. L'architecture de la solution proposée peut être illustrée suivant la figure 6.4.

De même qu'EmbedDSU, cette proposition appelée EmbedDSU_OSGi est constituée d'un ensemble de mécanismes off-card et on-card. La partie off-card permet de préparer la mise à jour, c'est-à-dire l'analyse des bundles, la comparaison des fichiers de classes et la génération des fichiers de DIFF avec l'inclusion des fonctions de transfert. La partie on-card est constituée en un ensemble de modules (certains de ces modules sont structurés sous forme de bundles). Le module de mise à jour est constitué de deux grandes parties :

1. la première, encapsulée dans un bundle OSGi, s'exécute sur la plateforme OSGi et se charge de mettre à jour les services dans le répertoire de services ;
2. la deuxième, intégrée dans la machine virtuelle Java, permet de fournir les fonctionnalités d'introspection des structures de données de la VM et de la plateforme OSGi, de détection de point sûr, de transfert d'état et de *roll-back* en cas d'échec.

Dans EmbedDSU_OSGi, le processus de mise à jour peut être divisé en deux grandes phases. Première phase : une fois, les fichiers de DIFF obtenus, ceux-ci sont interprétés, un point sûr du système est déterminé et les informations de mise à jour sont dispatchées au module de transfert d'état de la plateforme OSGi, et au module de transfert d'état de la machine virtuelle Java. Deuxième phase : il s'agit du transfert d'état proprement dit. Il est effectué en plusieurs étapes.

1. Adaptation de l'architecture
   Il s'agit de mettre à jour la structure de l'application liée par exemple à l'ajout de nouveaux fichiers de classe exportant un service dans le bundle. Dans ce cas, le fichier *Manifest.MF* peut nécessiter d'être mise à jour. La mise à jour des bundles impactés peut aussi être effectuée.
2. Adaptation de l'interface
   Il s'agit de la modification du répertoire de service pour mettre à jour les services ajoutés et/ou modifiés. Le système d'événements déjà fourni par OSGi peut être utilisé pour gérer le répertoire de service.

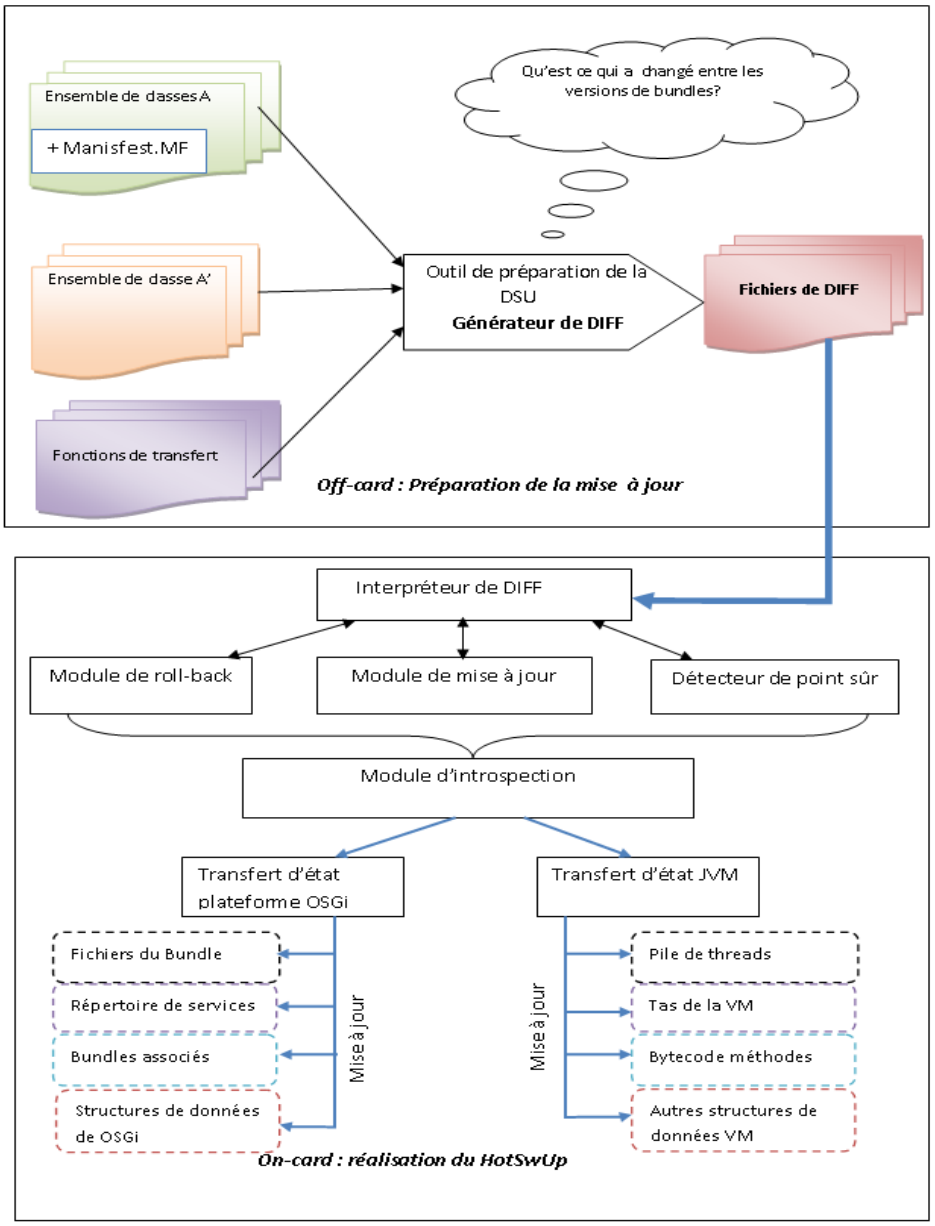

Fig. 6.3 – Proposition d'architecture DSU pour OSGi

3. Adaptation de l'état
   Il s'agit de la mise à jour des structures de données dans la plateforme OSGi, qui sont impactées par la mise à jour d'une ou plusieurs classes du bundle. En plus du transfert d'état d'OSGi, le transfert d'état de la JVM est aussi effectué. Il s'agit de mette à jour les instances dans le tas de la machine virtuelle, de mettre à jour la pile Java, et les autres structures de données de la VM.

En plus du transfert d'état d'OSGi, le transfert d'état de la JVM est aussi effectué. Il s'agit de mette à jour les instances dans le tas de la machine virtuelle, de mettre à jour la pile Java, et les autres structures de données de la VM.

6.2 Sûreté de l'approche d'EmbedDSU

L'objectif est de proposer un cadre formel pour EmbedDSU qui permettrait :

1. de décrire le processus de mise à jour tel qu'il est implémenté par EmbedDSU en utilisant des notations mathématiques et des règles de sémantique formelle ;
2. et de raisonner sur les propriétés du système. Il s'agit de définir et d'appliquer un certain nombre de règles formelles selon la spécification du comportement du système pour assurer la correction de celui-ci vis-à-vis du comportement souhaité en effectuant la mise à jour ;

L'idée dans cette partie est de vérifier que la nouvelle version de l'application obtenue en on-card à partir de la DIFF correspond bien à la nouvelle version en off-card et de vérifier aussi que le plan mémoire obtenu par EmbedDSU correspond bien au plan mémoire qui aurait été obtenu si l'exécution de la nouvelle version de la classe était lancée à partir du début.

6.2.1 Sûreté de DSU

Les produits basés sur des cartes à puce sont souvent évalués par un processus de certification permettant de fournir des éléments garantissant la qualité du processus de développement. Parfois un cadre réglementaire nécessite de disposer d'un niveau de certification afin de pouvoir utiliser ces produits. Ainsi l'Union Européenne impose l'utilisation d'un niveau de certification EAL4[51] dans le schéma des critères communs[52] si l'application utilise un mécanisme de signature électronique. Les documents électroniques comme les passeports visés dans le cadre de cette thèse devront évidement passer ce niveau de certification afin de pouvoir être déployés. Il faut donc raisonner sur la capacité de EmbedDSU à franchir cette étape de certification et de donner des pistes pour l'industriel désireux d'implémenter ce mécanisme dans un produit. Il faut être capable de démontrer qu'un programme chargé dans la carte a un comportement équivalent à un programme ayant été mis à jour. Ainsi, il s'agit d'un problème connu de compilation certifiée. Si on est capable de démontrer qu'un programme mis à jour est équivalent sémantiquement à un programme compilé alors nous assurons d'une part de la correction de notre approche mais aussi nous donnons les éléments permettant d'obtenir une certification de niveau sept dans le cadre des critères communs.

6.2.2 Solutions possibles

Dans le but de modéliser et analyser formellement la DSU, plusieurs travaux ont été effectués.

Les travaux de Gupta et al.

Gupta et al. [GJB96] proposent un cadre formel pour la modélisation de la DSU basé sur la notion d'états dit atteignables : un état que l'exécution d'un programme peut atteindre à partir d'un état initial. Les auteurs démontrent l'indécidabilité du problème de la validation de la DSU et proposent un ensemble de conditions qui permet d'en garantir la validité.

[51] Evaluation Assurance Level, `http://fr.wikipedia.org/wiki/Evaluation_Assurance_Level`
[52] `http://fr.wikipedia.org/wiki/Critères_communs`

Les travaux de G. Bierman et al.

Bierman et al. [BHSS03a] proposent un système de typage pour un langage supportant la DSU appelé *update-calculus* : un langage fonctionnel basé sur la notion de module avec l'introduction d'une primitive *update*. La validité de la mise à jour dynamique est basée sur la propriété du bon typage.

Ce travail est revisité par N. Charlton et al. [CHR11], avec l'introduction de la logique de Hoare et de la logique de Reynolds. La logique de Hoare a permis aux auteurs de démontrer des propriétés de correction sémantique.

Les travaux de S. Magill et al.

Magill et al. [MHH+11] présentent un cadre pour spécifier et vérifier la DSU basé sur la notion d'événement avec l'introduction d'un événement particulier appelé *upd* pour dénoter une mise à jour. La spécification y est classée en trois catégories : la spécification de ce qui est compatible dans les deux versions d'un programme, spécification de ce qui est apporté par une mise à jour et la spécification des propriétés affectant des comportements existants. La vérification est effectuée en fusionnant l'ancienne et la nouvelle version et en raisonnant sur le programme obtenu.

La caractéristique de ce système dédié aux mises à jour dynamique des applications Java Card est qu'il opère des modifications, préalablement définies, sur le code des applications s'exécutant sur les cartes à puce. Nous souhaitons garantir la validité des ces modifications et donc de donner une sémantique formelle à ces différentes opérations.

6.2.3 Modélisation

Dans un premier temps nous nous limitons à un sous langage de Java comme celui défini par Freund et Mitchell [FM98]. Ce sous langage est limité à une dizaine d'instructions représentatives de la manipulation des différents éléments de la mémoire : la pile et le tas.

La syntaxe de ces instructions est la suivante :

Instruction : := | pop | if L| store x| load x| new A| goto L | add | invokevirtual A l t|getfield A f t|putfield A f t

Ce sous langage permet de prendre en compte le flot de contrôle (hors exception), de manipuler la pile java et les variables locales, de créer des objets et d'accéder aux champs des instances. Nous augmentons ensuite les instructions habituelles avec un ensemble d'instructions de mise à jour appelées DSU_Instructions. Il s'agit d'instructions pour ajouter, supprimer et modifier une instruction donnée à une adresse donnée dans le code (exprimé par *at* pc).

DSU_Instruction : := Add_inst Instruction at pc |Dlt_inst Instruction at pc |Mod_inst Instruction *at* pc

6.2.4 Sémantique statique

La sémantique statique permet de représenter l'effet d'une instruction de mise à jour sur une configuration représentant l'état du byte code. Nous définissons les notions suivantes pour exprimer les règles sémantiques :

- F : cette fonction permet de garder trace des types des variables à une ligne donnée.
- S : une pile qui permet de garder trace du type de la pile d'exécution. Retourne, pour une adresse i, le type de la pile d'exécution au niveau de cette ligne.
- SD (*Stack Depth*) : garde à chaque ligne la hauteur de la pile d'exécution.
- M (*Mapping*) : c'est une fonction qui associe une instruction à chaque numéro de ligne.
- Dom :le domaine d'une méthode est l'ensemble des adresses qu'elle utilise.
- configuration : on appelle configuration du code à une ligne i, le quadruplet <(F,S,SD,M),i>.

Toutes les instructions sont donc formalisées par des règles. Par exemple l'ajout de l'instruction *new* A à la ligne (i+1) permet de passer d'une configuration à une autre si : la hauteur de la pile est incrémentée de 1, les variables locales ne sont pas affectées entre i et i+1, à la pile des type en ligne i est inséré un type A pour obtenir la pile des types de la ligne i+1, l'adresse i+1 appartient au domaine du byte code, et le *mapping* M2 est le résultat d'un ensemble d'opérations sur M1 afin d'y insérer l'instruction.

$$\frac{\begin{array}{c} Add\_inst \;\; new\ A(i+1) \\ SD_{i+1} = SD_i + 1 \\ S_{i+1} = A.S_i \quad F_{i+1} = F_i \\ M2 = Add\_inst(M1, new\ A, i+1) \\ PC\_MAX++ \quad i+1 \in DOM(BC) \end{array}}{<(F_i,S_i,SD_i,M1),i> \rightarrow <(F_{i+1},S_{i+1},SD_{i+1},M2),i+2>}$$

Fig. 6.4 – Proposition d'une sémantique statique

6.2.5 Sémantique opérationnelle

La mémoire consiste en trois éléments : le tas, le tas statique et la pile. Le tas contient les instances créées lors de l'exécution d'un programme. Il est modélisé par une liste d'objets. Le tas statique contient une structure pour chaque classe. Il est modélisé par une liste de structures. La pile contient une frame pour chaque méthode en exécution. Elle est représentée par une liste de frames. Un état de la mémoire est un enregistrement constitué du tas, du tas statique et d'une pile d'exécution.

record object := c : classe ; ref : reference ; lc : field list
record frame := opstack : val list ; locVar : val list ; pc :int ; max_op_stk :int
record structure :=f : field list ; m :method list ; I :idx ; super : classe ; var :val list
heap := list of object stack := list of frame static_heap := list of structure
record memory_state := SH : static_heap ; H : heap ; S : stack

La mise à jour du code source d'une méthode est décomposée en deux parties : la mise à jour du tas statique et la redirection vers la nouvelle méthode. Ceci est représenté par une fonction

(upd_code_source) qui prend en entrée un état mémoire, l'adresse de la méthode à modifier et une liste d'instructions DSU. Elle retourne un nouvel état mémoire et la nouvelle référence. La fonction *upd_SH* implémente les opérations DSU décrites dans la sémantique statique.

```
function upd_code_source(m :memory_state,meth_ref :reference,l :DSU_instr
list :=
    let (SH1, new_ref)= upd_SH(m, meth_ref, l) in
        let m1 = upd_old_ref ( m, new_ref) in SH1, m1.H, m1.S
```

La mise à jour du code source de la méthode est complétée par une redirection vers la nouvelle référence de la méthode. Ceci est modélisé par les fonctions d'introspection du tas et de la pile (*introspect_heap*, *introspect_stack* et les fonctions de mise à jour (*upd_frames*, *upd_heap*).

Ce travail est actuellement en cours [Raz12] et ne porte que sur la formalisation des sémantiques statiques et opérationnelles. Lorsque ces dernières seront complètes il nous faudra définir les propriétés d'équivalence représentées par l'état des mémoires de la pile et du tas. Ceci nécessitera l'introduction des fonctions de mise à jour de l'état. A l'aide d'un assistant de preuves (*Coq*) nous espérons effectuer les preuves formelles d'équivalence.

6.3 Conclusion

Les systèmes de DSU permettent de modifier les fonctionnalités d'une application ou d'un système de façon dynamique et autonome durant leur exécution par le biais d'opérations d'introspection, d'ajout, de suppression et de transfert d'état d'exécution. Un des moyens de favoriser leur robustesse et d'assurer leur validité est de disposer d'un support formel permettant de modéliser ces applications ou systèmes, de spécifier les systèmes de DSU, d'y exprimer des propriétés et de les vérifier. Ainsi, une des perspectives de cette thèse est de proposer un cadre formel de spécification et de raisonnement sur les applications et le système EmbedDSU pour la mise à jour dynamique. En outre, EmbedDSU s'applique sur les systèmes à ressources limitées notamment les cartes à puce. L'idée est d'étendre la solution d'EmbedDSU au domaine du système embarqué en général notamment sur des systèmes basés sur OSGi, d'où l'autre perspective de cette thèse.

Chapitre 7

Conclusion

Sommaire

En réponse au besoin croissant d'évolutivité des systèmes et en parallèle du besoin de fiabilité, l'objectif de cette thèse était de concevoir et de mettre en place un système de DSU pour système à ressources limitées et plus particulièrement pour Java Card. Les systèmes de DSU sont un moyen de faire évoluer les applications et/ou systèmes sans les arrêter en préservant leurs disponibilités et l'état d'exécution de ceux-ci. Nous avons considéré les hypothèses suivantes pour la mise en place du système EmbedDSU :

1. Les mises à jour sont *dynamiques* : elles se produisent pendant l'exécution du système ou de l'application.
2. les mises à jour sont *non-anticipées* : une mise à jour n'est pas nécessairement définie et prévue au moment de la conception ou du déploiement du système.
3. les mises à jour sont *atomiques* : bien que plusieurs mises à jour dynamiques puissent être exécutées en même temps sur le système, elles doivent être exécutées entièrement avant que la nouvelle version puisse continuer l'exécution. En effet, les systèmes de DSU procèdent à des transformations de l'état du système qui peuvent à ce titre rendre ce dernier incohérent et dans l'incapacité de fournir les fonctionnalités attendues. Il est donc important de garantir la cohérence des systèmes mis à jour.

7.1 Problématique

Les différents travaux autour de la mise à jour dynamique, présentés dans cette thèse, montrent la complexité et la délicatesse de cette problématique. Cette dernière est encore plus accentuée lorsqu'il s'agit des systèmes à forte contraintes de ressources (processeurs, mémoires, etc.). La complexité est liée à l'absence de mécanisme de haut niveau, permettant d'intervenir dynamiquement sur une application ou sur un système. Ceci est d'autant plus vrai dans un environnement de

carte à puce. La délicatesse est liée au fait qu'on agisse sur une application en cours d'exécution dont l'état change constamment. Une mise à jour non réussie peut mettre en cause l'intégrité du système et conduire l'application ou le système à s'arrêter causant ainsi des anomalies notamment pour les utilisateurs.

La mise à jour dynamique est un sujet sur lequel se penchent depuis des années, une grande communauté de chercheurs. Les problèmes scientifiques généralement abordés par ces travaux peuvent être présentés en quatre points :

- Identifier les moments où la mise à jour peut être réalisée tout en préservant le système de type et la cohérence du système ;
- Proposer des approches de transfert d'état d'exécution de l'ancienne version vers la nouvelle ;
- Proposer des approches de mise à jour du code proprement dit ;
- Proposer un cadre formel de vérification de la validité du processus de mise à jour.

Cette liste peut s'étendre. En effet, plusieurs autres questions liées à la problématique de mise à jour dynamique restent ouvertes et sujettes à des travaux de recherche notamment pour la tolérance aux fautes dans les systèmes reparties.

7.2 Principales contributions

Le but de cette thèse est la mise en place d'un système de mise à jour dynamique des applications et composants systèmes pour cartes à puce basées sur une machine virtuelle Java. Pour cela, nous avons proposé une solution axée autour de plusieurs propriétés :

1. Pas d'intervention humaine : EmbedDSU est un système de mise à jour dynamique totalement automatique. En effet, étant donné deux versions d'une application, il détermine les classes modifiées, ajoutées, supprimées et inchangées. Pour chaque classe modifiée, il ressort le fichier de DIFF associé aux deux versions et y ajoute ensuite les fonctions de transfert. Après le transfert du fichier de DIFF binaire, il interprète celui-ci et applique la mise à jour.
2. Support des niveaux de granularité de modification : EmbedDSU supporte plusieurs niveaux de granularité de modification pour la mise à jour dynamique : signature de classes, signature de méthodes, bloc d'instructions, variables locales, etc.
3. Atomicité de la mise à jour : EmbedDSU préserve la cohérence du système en assurant une mise à jour atomique tout en proposant un processus de restauration en cas de non-atomicité.
4. Préservation de la sémantique : EmbedDSU préserve le système de type et transforme l'ancienne version en on-card pour qu'elle corresponde à la nouvelle version en off-card, ceci grâce uniquement au fichier de DIFF.

Les contributions de cette thèse se décomposent en deux parties :

1. la proposition d'une approche et architecture appropriée au monde de la carte, basée notamment sur des mécanismes off-card et on-card.
2. l'implémentation de l'approche proposée en se basant sur une machine virtuelle Java pour l'embarqué appelée simpleRTJ.

Plusieurs perspectives d'EmbedDSU ont été définies, notamment :
(1) adapter EmbedDSU pour proposer une solution de DSU pour des environnements embarqués

basés sur un framework orienté service tel que OSGi-Me ;
(2) proposer une solution de formalisation pour démontrer que les nouvelles versions de code et d'instances obtenues en on-card grâce au fichier de DIFF correspond bien à celle qu'on aurait obtenu en chargeant entièrement la nouvelle version dans la carte.

7.3 Synthèse générale

Le premier résultat de ses travaux est la faisabilité d'une telle approche. Avec une granularité au niveau de la classe, il est possible de mettre à jour dynamiquement des applications java avec peu de ressources. Les surcoûts en terme de mémoire sont limités (les classes mises à jour doublent de taille lors de l'opération) et l'opération de modification des classes lors de la recopie permet un coût limité en terme de temps de calcul. De plus, toutes les opérations off-card sont simples et répondent efficacement au problème de détection des modifications et du transfert d'état.

Deuxièmement, en ayant réussi à mettre les opérations coûteuses et sensibles en grande majorité en dehors de la carte, le dispositif on-card est simple à implémenter et peu coûteux en taille de code embarqué dans la carte. Ce code peu complexe et léger peut donc être vérifié plus facilement. Les perspectives ont d'ailleurs présenté une première solution de formalisation afin d'avoir des garanties offertes sur le fait que les nouvelles versions de code et d'instances obtenues en on-card grâce au fichier de DIFF correspondent bien à celles qu'on aurait obtenu en chargeant entièrement la nouvelle version dans la carte.

Enfin, la réalisation d'un prototype fonctionnel sur une plate-forme AT91 EB40A très proche de l'architecture d'une vraie carte à puce a permit de démontrer ce faible surcoût. Même si la machine virtuelle utilisée n'était pas une vraie Java Card, les ressemblances entre les deux architectures sont suffisamment proche pour valider le travail présenté dans ce mémoire.

Cinquième partie

Annexes

Annexe A

Sommaire

A.1 Présentation du format DIFF binaire

Le fichier de DIFF binaire est contient les éléments suivantes de façon linéaire :

1. Une entête,
2. Les informations sur la table de constante,
3. Les informations sur les méthodes,
4. Les informations sur les champs,
5. Pour chaque méthode,
   - Une entête de méthode,
   - et la liste des modifications effectuées au niveau des instructions.
6. Une entête de la fonction de transfert et les instructions de transfert d'état.

A.1.1 Entête du fichier de DIFF

Le fichier commence par l'entête et le premier champ de l'entête indique le nombre magique sur quatre octets pour signaler le format du fichier. Il doit avoir la valeur hexadécimale 0xD1FF. Ces informations dans l'entête permettent d'accéder directement aux autres structures de données

contenues dans le fichier. Il est représenté par la structure de donnée suivante :

```
Diff_header                {
uint32 magic;              //0xD1FF
uint32 class_ptr;          // Adresse de la classe
uint16 flags;              // Les drapeaux de la classe
uint32 inst_size;          // Taille d'une instance de la classe
uint16 *ifaces;            // Adresse de l'interface
uint16 methmod_count;      // Nombre de méthode modifié
uint16 *meth_tbl;          // Adresse de la table de méthode
uint16 *field_tbl;         // Adresse de la table de champs
uint16 *list_modmeth;      // Adresse de la liste de méthodes modifiées
uint16 nbfields;           // Nombre total de champs de la classe
uint16 nbmeth;             // Nombre total de méthodes de la classe
uint16 nbentriescp;        // Nombre total d'entrée dans la table de constante
```

Valeur des drapeaux

Nom du drapeau	Value
ACC_PUBLIC	0x0001
ACC_FINAL	0x0010
ACC_SUPER	0x0020
ACC_INTERFACE	0x0200
ACC_ABSTRACT	0x0400

A.1.2 La table de constante

Il s'agit d'exprimer les modifications effectuées au niveau des tables de constantes des deux versions de classes.

```
cp_info {
uint8 action_type;     // Le type de modification (ADD, DEL, MOD), le type correspondant
                       à l'entrée (Méthode ou champ ou classe, etc).
uint16 oldindex;       // Ancien index
uint16 newindex;       // Nouvel index
uint16 newpos;         // Nouvelle position occupée dans la table de méthode ou des champs
}
```

L'action peut être :

ADD =	Ajouté (0x1)
DEL =	Supprimé (0x2)
MOD =	Modifié (0x3)

Le type correspondant à une entrée de la table de constante peut être :

Type de constante	Valeur
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_Integer	3
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

A.1.3 La table de méthodes

Cette partie décrit les modifications effectuées sur les entrées de la table de méthodes des fichiers classes.

```
method_info {
uint8 action_type;    // Type de modification et type de retour de la méthode
uint16 oldindex;      // Ancien index
uint16 newindex;      // Nouvel index
uint32 hash;          // Nouveau Hash de la méthode
}
```

Type de retour de la méthode

Caractère	Type	Interpretation
B	byte	Byte signé
C	char	Caractère unicode
I	int	Entier
L<classname>;	reference	Une instance de la classe <classname>
S	short	Entier court signé
Z	boolean	True or false
m	reference	Tableau à une seule dimension

A.1.4 La table de champs

Elle décrit les modifications détectées sur les tables de champs des deux versions de fichiers de classe.

```
field_info {
uint8 action_type;    // Type de modification et le type de la variable
uint16 oldindex;      // Ancien index
uint16 newindex;      // Nouvel index
uint16 flag;          // Flag d'accès
uint16 value;         // Valeur d'initialisation
}
```

Les drapeaux associés aux champs

Nom du drapeau	Valeur
ACC_PUBLIC	0x0001
ACC_PRIVATE	0x0002
ACC_PROTECTED	0x0004
ACC_STATIC	0x0008
ACC_FINAL	0x0010

A.1.5 La liste de méthodes

Cette partie permet de décrire en détail les modifications détectées pour chaque méthode modifiée de la classe. Elle comporte pour chaque entrée de la lise, une entête permettant de spécifier la méthode correspondante, ensuite les modifications constatées au niveau des variables locales, de la signature de la méthode et celles constatées au niveau des byte codes de la méthode. La liste de méthode est présentée en un ensemble d'entrée. Chaque entrée est associée à une méthode et comporte une partie entête et la liste de modifications détectées sur celle-ci.

Cette structure présente l'entête d'une entrée de la liste des méthodes

```
method_head_info {
uint8 action_typere;    // Type de modification (MOD or ADD) et le type de retour
uint16 oldindex;        // Ancien index dans la table de méthode (dans le fichier .class)
uint16 newindex;        // Nouvelle index ou pos dans la table de méthode (dans le fichier .class)
uint8 nbargs;           // Nombre d'arguments
uint16 flag;            // Drapeaux d'accès (public, private, etc.)
uint32 hash;            // Le hash de la méthode
uint8 nbentry;          // Le nombre d'entrée correspondant aux modifications au niveau de la méthode
uint16 nblocalvar;      // Le nombre de variables locales de la méthode
}
```

Droit d'accès à la méthode

Nom du drapeau	valeur
ACC_PUBLIC	0x0001
ACC_PRIVATE	0x0002
ACC_PROTECTED	0x0004
ACC_STATIC	0x0008
ACC_FINAL	0x0010
ACC_ABSTRACT	0x0400

Ensuite, pour cette méthode, la structure correspondante à une entrée est décrit comme suit :

```
method_entry_info {
uint8 action;           // Le type de modification
uint16 pc_deb;          // Le numéro du compteur de programme où commence la modification
uint8 length;           // La taille des informations contenues dans le champ listbytecode de la structure
uint8 nbinstr;          // Le nombre d'instruction contenu dans le champ listbytecode de la structure
uint32 * listbytecode;  // La liste de byte codes concernée par la modification
}
```

Toutes ces informations structurées et réunies constituent le fichier de DIFF binaire à transférer dans la carte. Cependant, EmbedDSU prévoit un module de vérification de ce fichier avant son transfert on-card.

Annexe B

Sommaire

B.1 Vérification des instructions du fichier de DIFF

B.1.1 Type de vérification

Il s'agit de vérifier que les instructions spécifiées dans le fichier sont bien des instructions valides. Vérifier que les index de la table de constantes, les index des variables locales, le index des méthodes spécifiés dans les instructions sont valides. Vérifier que les PCs des instructions sont valides par rapport à la taille des méthodes associées.

La vérification s'effectue à deux niveaux :

1. Vérifier le format de la DIFF et la cohérence du fichier de DIFF (vérification du nombre magique et de la cohérence des valeurs des structures de données du fichier) ;
2. Vérifier le flow d'instruction dans chaque entrée de la liste de méthode
   - Vérifier que les instructions ne sont pas illégales et que le nombre d'arguments est valide ;
   - Pour une suite d'instructions ajoutées, vérifié si la suite peut être valide, (en utilisant la notion de pattern d'instruction) ;
   - Pour chaque instruction, vérifier
     (a) si les arguments sont valides ;
     (b) index_cp par exemple correspond à une entrée valide de la table de constante ;
     (c) index d'une variable locale ne peut pas avoir une valeur supérieur au nombre de variables locales de la méthode ;

(d) le PC d'une instruction de branchement conditionnel ou inconditionnel, ne peut pas dépasser la taille de la méthode ou être négatif ;

(e) Index d'une méthode ne peut pas dépasser le nombre de méthode de la classe.

– Vérifier que les instructions sont applicables sur les types autorisés, par exemple en Java Card, il n'y a pas de type *float*, donc vérifier qu'on n'a pas d'instruction s'appliquant sur ce type là (*fmul*, *fadd*, etc.)

B.1.2 Vérification du format du fichier

Il s'agit :

1. de s'assurer que le nombre magique est bien présent dans le fichier, de vérifier que les adresses de déplacements (offset) dans le fichier correspondent bien à des éléments accessibles ;
2. que le nombre de méthodes modifiées précisé dans l'entête correspond bien au nombre d'entrée dans la table des méthodes modifiées, vérifier la correspondance entre les méthodes de la liste de méthodes et la table des méthodes ;
3. vérifier que les index de la table de constante spécifiés correspond à des index valides et que les index des méthodes spécifiés correspond bien à des index valides.

B.1.3 Vérification des instructions

Il s'agit de :

1. de vérifier que les instructions spécifiées dans le fichier sont bien des instructions valides.
2. Vérifier que les index de la table de constantes, les index des variables locales, le index des méthodes spécifiés dans les instructions sont valides.
3. Vérifier que les PCs des instructions sont valides par rapport à la taille des méthodes associées.

B.1.4 Code d'erreur

Le tableau ci-dessous présente quelques types d'erreurs pouvant être détectées dans le fichier de DIFF binaire.

Type de l'érreur	Code de l'erreur	Signification
Err_diff_header	0x0A	erreur localisée dans la structure Diff_header
Err_index	0x0D	index non valide ou doublons
Err_flag	0x10	Droit d'accès non valide (autre que 0x01, 0x02, 0x04, 0x08, 0x10)
Err_hash	0x11	hash de la méthode a été modifié
Err_args	0x15	erreur d'arguments dans une instruction
Err_limit_jvm	0x13	Erreurs liée aux limitations de la machine virtuelle
Err_action_type	0x0B	Erreur sur le champ action_type
Err_inst	0x14	Instruction non supportée par la version 2.2.

Tab. B.1 – Quelques erreurs pouvant être détectées dans le fichier de DIFF binaire

B.2 Limitations machine virtuelle standard

255 classes publiques et interfaces publiques dans un package
15 interfaces implémentées par une classe en incluant celle implémentée par la superclasse
Pour une classe
255 champs statiques
255 méthodes statiques
128 méthodes d'instances (public et protegé) en incluant ceux de la classe mère
255 champs d'instances, où entier occupe deux champs
* Nombre total de méthodes statiques et d'instances = 383
* Nombre total de champs statiques et d'instances = 383
Taille de tableau fixé à 32 757 champs
Pour une méthode
255 variables locales, où une variable de type entier occupe deux variables locales
Taille du code d'une méthode : 32 767 octets
Nombre de cas d'une instruction switch est de 65 536

Annexe C

Sommaire

C.1 Format d'une application sous simpleRTJ

Le format d'une application sous simpleRTJ (*binFile*) est constitué d'un ensemble de sections de taille variable, dans un ordre prédéfini suivant.

1. Entête du fichier
2. Table des offsets de classe
3. Table des offsets des classes d'exceptions
4. Les classes, et pour chaque classe
   - Entête de classe
   - Table de constante
   - Table de méthode
   - Table des valeurs de constante
   - Table de champs
   - Les méthodes et pour chaque méthode
     (a) Entête de méthodes
     (b) Bytecode

C.2 Entête du fichier

Le *binFile* possède un entête de fichier contenant les informations générales sur l'application. Ces informations peuvent être :

- L'adresse du point d'entrée de l'application (adresse du main),
- La taille de l'application,
- La version du classlinker utilisée,
- L'adresse de la table d'exception,
- La taille maximale d'une frame,
- La taille maximale d'une instance de la classe, etc

C.3 Les tables

Table d'offset de la classe
C'est une table qui liste les adresses de l'ensemble des classes de l'application et des autres classes nécessaires inclut statiquement (*java.lang.Object*, *java.lang.Thread*, *java.lang.throwable*, etc).

Table d'offset de classe d'exception
Il s'agit des classes d'exceptions pouvant être levées lors de l'exécution de l'application. Cette table liste donc l'ensemble des adresses des classes d'exceptions.

C.4 Les classes

Les classes. Les informations sur la liste des classes. Le contenu de chaque classe est présenté dans les paragraphes suivants.

C.4.1 L'entête de la classe

Il présente les informations générales sur la classe telles que :

- Le nombre de méthodes,
- L'adresse de la table de méthode,
- L'index de la classe dans la table d'offset de la classe,
- Les flags d'accès,
- L'adresse de l'interface implémentée,etc

C.4.2 La table de constante

Il contient l'ensemble des adresses des méthodes et variables de la classe dans leur ordre d'utilisation, leurs adresses et leur type (champ ou méthode). Les variables et les méthodes non utilisées dans la classe n'y apparaissent pas. Et ainsi, à partir de l'index de la table de constante, obtenu lors de la lecture d'une instruction, on peut avoir l'adresse de la variable ou de la méthode et donc accéder soit aux bytecode de la méthode, soit aux informations sur la variable.

C.4.3 Les tables

Table de méthode
Elle contient les adresses relatives et les hashs de l'ensemble des méthodes de la classe.

Table des valeurs de constante
Liste l'ensemble des valeurs de constantes de la classe.

Table de champs
Elle contient les informations sur l'ensemble des champs ou variables de la classe. Ces informations peuvent être : le type, l'index dans la table de champs et les drapeaux (droit d'accès).

C.4.4 Les méthodes de la classe

Pour chaque méthode de la classe, les informations sur celle-ci sont fournies par l'entête de la méthode et suivi de son byte code. L'entête de la méthode fourni la taille de la méthode, l'adresse de début du byte code de la méthode, le nombre d'arguments, la taille des variables locales, les drapeaux, etc. Cet entête est suivi de la liste des instructions (du byte code) de la méthode. Ces instructions peuvent faire référence à une entrée dans sa table de constante.

Annexe D

Sommaire

D.1 Définition langage DSL

Les DSLs (*Domain Specific Language*) sont des langages dédiés à des domaines particuliers. L'objectif étant de représenter un problème d'un domaine particulier et précis tout en masquant la complexité technique du domaine. Ce qui permet généralement d'éviter des erreurs de programmation dues à l'utilisation d'un langage généraliste, de vérifier le code avant de le compiler, de rendre le code plus clair, facilement compréhensible et donc facile à valider par un expert du domaine.

Les DSLs contrairement aux langages généralistes sont conçus pour être utiles à une tache spécifique dans un domaine restreint. Les DSLs sont des langages dont les buts sont assez précis. Dans la littérature, les DSLs sont utilisé dans de nombreux domaines [Vis08, DK01]. Les utilisations les plus courantes sont dans les domaines de la documentation de la connaissance, dans la définition ou la génération de tests, dans les vérifications formelles, dans la configuration des systèmes de déploiement, la génération de code, etc. Et leur conception est fortement influencée par les besoins. L'objectif est donc de concevoir un DSL léger capable de décrire les changements entre deux versions d'une classe en étant simple, et concis. Généralement, l'évolution de code implique souvent l'analyse de deux versions : l'original et la version modifiée. Notre but est donc de fournir un outil capable d'analyser deux versions de classes, en effectuant un *mapping* entre les différentes entités de la classe telle que les champs, les méthodes, les instructions entre l'original et la version modifiée correspondante et d'en ressortir et classifier les entités ajoutées, supprimées, modifiées ou inchangées avec des informations complémentaires relatives telles que l'index de la table de constante modifié, le PC des instructions supprimées, etc. Une fois, ces différences trouvées, de les exprimer dans le langage DSL conçu à cet effet.

D.2 DSL EmbedDSU

D.2.1 Mise en place du DSL

Il s'agit de mettre en place un langage ou plutôt un format d'expression des différences trouvées lors de la comparaison des deux fichiers de classe par le Générateur de DIFF. Ce langage dédié à l'expression des différences est basé sur les types de modifications pouvant être effectuées dans un fichier Java notamment sur celles traitées par EmbedDSU. Ces types de modifications peuvent être illustrées selon le schéma D.1.

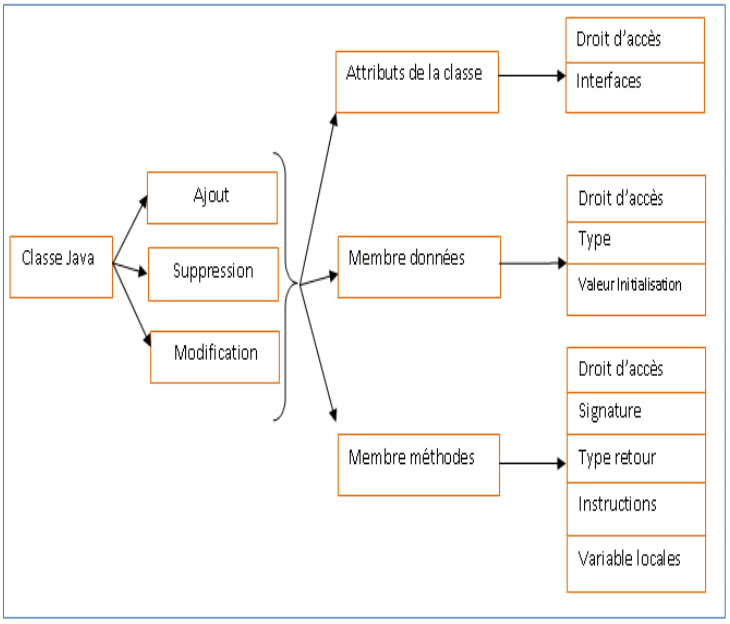

Fig. D.1 – Vue sur l'ensemble des modifications considérées

D.2.2 Grammaire

Une syntaxe de la grammaire proposée peut être présentée à partir de la figure D.2.

Cette grammaire est représentée par l'axiome <file_element>, par un ensemble de variables. On peut ressortir aisément les règles syntaxiques et les règles terminales de celle-ci.

Update_unit ::= <file_element>

<File_element> ::= 0xD1FF <index_class> { (<statement_block>)*}

<statement_block> ::= <object_modif> { (<object_attrib> : (<block_instr>)*)*}<end_object>

<object_attrib> ::= <Method_attrib> | <Field_atrrib> | <Interface_attrib> | <Attribute_attrib>

<object_modif> ::= Method| Field | Interface |Attribute

<Meth_attr> ::= name | signature | return_type | acces_flags | instr

<Field_attr> ::= name | init_value | index_cp | type | acces_flags

<Attribute_attrib> ::= name | superclasse | access_flag | subclasse | package_name

<block_instr> ::= <type_modif><instr><condition><impact>

<type_modif> ::= (add | del | mod)%

<instr> ::= <instr_bytecode> ;

<condition> ::= (after| before |within)* <position_pc>

<impact> ::= (<list_of_concerned_classes>,)*

<end_object> ::= (end_meth | end_attr | end_inter)

Fig. D.2 – Grammaire DSL

Bibliographie

[ABBC98] David Atkins, Thomas Ball, Glenn Bruns et Kenneth Cox. Mawl : a domain-specific language for form-based services. *Dans International conference of software Reuse, Victoria, Canada*, 1998.

[ABBS05] Gautam Altekar, Ilya Bagrak, Paul Burstein et Andrew Schultz. OPUS : Online Patches and Updates for Security. *Dans 14th USENIX Security Symposium*, pages 287–302, 2005.

[AFK+97] Marc Auslander, Hubertus Franke, Orran Krieger, Ben Gamsa, et Michael Stumm. Customization-Lite. *Dans 6th Workshop on Hot Topics in Operating Systems*, pages 43–48, 1997.

[AHS+02] Jonathan Appavoo, Kevin Hui, Michael Stumm, Robert Wisniewski, Dilma da Silva, Orran Krieger et Craig Soules. An Infrastructure for Multiprocessor Run-Time Adaptation. *Dans WOSS*, pages 3–8. ACM, 2002.

[AK09] Jeff Arnold et M. Frans Kaashoek. KSplice : Automatic Rebootless Kernel Updates. *Dans ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 187–198, 2009.

[Apa06] Apache. http ://jakarta.apache.org/bcel/manual.html. *Apache Software Foundation*, 2006.

[AR00] Jesper Andersson et Tobias Ritzau. Dynamic Code Update in JDRUMS. *Dans ICSE'00 Workshop on Software Engineering for Wearable and Pervasive Computing*, 2000.

[Arc] Java Platform Debugger Architecture. This supports class replacement. `http://java.sun.com/j2se/1.4.2/docs/guide/jpda/`.

[Arn09] Jeff Arnold. Security Without Disruption : Ksplice Kernel Update. Rapport technique, Open Source Convention (OSCON), 2009.

[Atm09] Atmel. http ://www.atmel.com/products/at91/. *Atmel Corporation*, 2009.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström et Mike Williams. Concurrent programming in ERLANG Second Edition. Prentice Hall International Ltd, 1996.

[Bau07] Andrew Baumann. *Dynamic Update for Operating Systems*. Thèse de doctorat, School of Computer Science and Engineering, University of New South Wales, 2007.

[BE03] Miles Barr et Susan Eisenbach. Safe Upgrading without Restarting. *Dans IEEE Conference on Software Maintenance ICSM'2003. IEEE*, pages 129–137, 2003.

[BH00] Bryan Buck et Jeffrey K. Hollingsworth. An API for runtime code patching. *Dans Journal of High Performance Computing Applications*, pages 317–329, 2000.

[BHS+05] Andrew Baumann, Gernot Heiser, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski et Jeremy Kerr. Providing dynamic update in an operating system. *Dans USENIX Annual Technical Conference*, pages 279–291. USENIX Association, 2005.

[BHSS03a] Gavin Bierman, Michael Hicks, Peter Sewell et Gareth Stoyle. Formalizing Dynamic Software Updating. *Dans Second International Workshop on Unanticipated Software Evolution*, pages 13–23, 2003.

[BHSS03b] Gavin Bierman, Michael Hicks, Peter Sewell et Gareth Stoyle. Formalizing Dynamic Software Updating. *Dans Workshop on Unanticipated software Evolution*, 2003.

[BKA+05] Andrew Baumann, Jeremy Kerr, Jonathan Appavoo, Dilma Da Silva, Orran Krieger et Robert W. Wisniewski. Module Hot-Swapping for Dynamic Update And Reconfiguration in K42. *Dans 6th Linux Conference*, 2005.

[BLS98] Don Batory, Bernie Lofaso et Yannis Smaragdakis. Jts : Tools for implementing domain-specific languages. *Dans International conference of software Reuse, Victoria, Canada*, 1998.

[BLS+03] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh et Steven Richman. Lazy Modular Upgrades in Persistent Object Stores. *Dans Object-Oriented Programming, Systems, Languages, ans Applications*, pages 403–417, 2003.

[Bod06] Eric Bodden. proof-carrying code, 2006.

[BPN08] Gavin Bierman, Matthew Parkinson et James Noble. UpgradeJ : Incremental Typechecking for Class Upgrades. *Dans 22nd European conference on Object-Oriented Programming*, page 235–259. ECOOP, 2008.

[BR10] Andre Bottaro et Fred Rivard. OSGi ME, An OSGi Profile for Embedded Devices. *Dans OSGi Community Event, London*, 2010.

[Bru07] Eric Bruneton. *ASM 3.0 A Java Bytecode engenieering library.* February 2007.

[BSKW04] Andrew Baumann, Dilma Da Silva, Orran Krieger et Robert W. Wisniewski. Improving operating system availability with dynamic update. *Dans 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, pages 21–27, 2004.

[BSP+95] Brian N. Bershad, Stefan Savage, Przemys Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers et Susan Eggers. Extensibility, safety and performance in the SPIN operating system. pages 267–284, 1995.

[BWD+07] Andrew Baumann, Robert W. Wisniewski, Dilma Da, Silva Orran et Krieger Gernot Heiser. Reboots are for Hardware : Challenges and Solutions to Updating an Operating System on the Fly. *Dans USENIX Annual Technical Conference*, pages 337–350, 2007.

[Car08a] Dot Net Smart Card. http ://www.dotnetsmartcard.com/. 2008.

[Car08b] Java Card. The java card 3.0 specification. *http ://java.sun.com/javacard/*, March 2008.

[CFW10] Daniel CHEUNG-FOO-WO. *Adaptation dynamique par tissage d'aspects d'assemblage*. Thèse de doctorat, Université de Nice-Sophia Antipolis, 2010.

[Che00] Z. CHEN. *Java Card Technology for Smart Cards*. Addison, Wesley, 2000.

[CHR11] Nathaniel CHARLTON, Ben HORSFALL et Bernhard REUS. Formal Reasoning about Runtime Code Update. *Dans International Workshop on Hot Software Updates*, pages 134–138, 2011.

[CHYZ11] Haibo CHEN, Jie Yu C. HANG, Pen YEW et Binyu ZANG. Dynamic Software Updating Using a Relaxed Consistency Model. *Dans IEEE Transactions on Software Engineering*, pages 679–694, 2011.

[CYC+07] Haibo CHEN, Jie YU, Rong CHEN, Binyu ZANG et Pen chung YEW. POLUS : A POwerful live updating system. *Dans 29th International Conference on Software Engineering*, pages 271–281, 2007.

[DE03] S. DROSSOPOULOU et S. EISENBACH. Flexible, source level dynamic linking and relinking. *Dans Workshop on Formal Techniques for Java Programs*, 2003.

[DGW97] Stephen Gilmore DILSUN, Stephen GILMORE et Christopher WALTON. Dynamic ML without Dynamic Types. Rapport technique, University of Edinburgh, 1997.

[DK01] Arie V. DEURSEN et Paul KLINT. Domain-Specific Language Design Requires Feature Descriptions. *Dans Journal of Computing and Information Technology*, volume 10, page 2002, 2001.

[Dmi01] M. DMITRIEV. Towards flexible and safe technology for runtime evolution of Java language applications. *Dans Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference*, 2001.

[Dug01] D. DUGGAN. Type-based hot swapping of running modules. *Dans ICFP*, 2001.

[FM98] Stephen N. FREUND et John C. MITCHELL. A Type System for Object Initialization in the Java TM Bytecode Language. *Dans ACM SIGPLAN Conference on ObjectOriented Programming, Systems, Languages Application*, pages 310–327. ACM Press, 1998.

[For06] Milan FORT. Smart card application development using javacard technology. 2006.

[FQ03] Stephen J. FINK et Feng QIAN. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. *Dans International Symposium on Code Generation and Optimization*, pages 241–252, 2003.

[FS91] O. FRIEDER et M. E. SEGAL. On dynamically updating a computer program : From concept to prototype. *Dans The Journal of Systems and Software*, pages 111–128, 1991.

[GHG08] G. Koning GANS, Jaap-Henk HOEPMAN et Flavio D. GARCIA. A Pratical Attack on MIFARE Classic. *Dans Smart Card Research and Advanced Application Conference*, pages 267–282, 2008.

[GJB96] Deepak GUPTA, Pankaj JALOTE et Gautam BARUA. A Formal Framework for On-line Software Version Change. *Dans IEEE Transactions on Software Engineering*, 1996.

[GKAS97] Benjamin GAMSA, Orran KRIEGER, Jonathan APPAVOO et Michael STUMM. Tornado : Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. *Dans OSDI*, pages 87–100. ACM, 1997.

[GR89] A. Goldberg et D. Robson. Smalltalk 80 - the Language and its Implementation. Addison-Wesley, 1989.

[Gup94] D. Gupta. *On line software version change*. Thèse de doctorat, Department of Computer Science and Engineering, Indian Institute of Technology, 1994.

[Ham10] Mohamad Fady Hamoui. *Un Système Multi-Agents à Base de Composants pour l'Adaptation Autonomique au Contexte Application à la Domotique*. Thèse de doctorat, Université de Montpellier II, 2010.

[HAW+01] Kevin Hui, Jonathan Appavoo, Robert Wisniewski, Marc Auslander, David Edelsohn, Ben Gamsa, Orran Krieger, Bryan Rosenburg et Michael Stumm. Position Summary : Supporting Hot-Swappable Components for System Software. *Dans Workshop on Hot Topics in Operating Systems*. ACM, 2001.

[HG98] Gísli Hjálmtýsson et Robert Gray. Dynamic C++ classes, a lightweight mechanism to update code in a running program. *Dans USENIX Annual Technical Conference*, pages 65–76, 1998.

[Hic01] Michael Hicks. *Dynamic Software Updating*. Thèse de doctorat, Department of Computer and Information science, University of Pennsylvania, 2001.

[ISO97a] International Standard ISO/IEC. Iso/iec 14443-1. *ISO/IEC JTC1/SC*, 1997.

[ISO97b] International Standard ISO/IEC. Iso/iec 7816. *ISO/IEC JTC1/SC*, 1997.

[IT02] Yuuji Ichisugi et Akira Tanaka. Difference-based modules : A class independent module mechanism. *Dans ECOOP 2002, volume 2374 of LNCS, Malaga*, pages 62–88. Springer Verlag, 2002.

[JM07] Maria Jump et Kathryn McKinley. Cork : Dynamic Memory Leak. Detection for Garbage-Collected Languages. *Dans ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 31–38, 2007.

[Kim09] Dong Kwan Kim. *Applying Dynamic Software Updates to Computationally-Intensive Applications*. Thèse de doctorat, Faculty of the Virginia Polytechnic Institute and State University, 2009.

[KS97] Orran Krieger et Michael Stumm. HFS : A performance-oriented flexible file system based on building block composition. *Dans ACM Transactions on Computer Systems*, pages 286–321. ACM, 1997.

[KT08] Dong Kwan Kim et Eli Tilevich. Overcoming JVM HotSwap Constraints via Binary Rewriting. 2008.

[Lab09] Embedded Systems Laboratory. Jazelle - ARM Architecture Extensions for Java Applications. `http://cadal.cse.nsysu.edu.tw/seminar/seminar_file/sywang_0917.pdf`, 2009.

[LC06] Yueh-Feng Lee et Ruei-Chuan Chang. Hotswapping Linux kernel modules. *Dans Journal of Systems and Software*, 2006.

[Leg09] Marc Leger. *Fiabilité des Reconfigurations Dynamiques dans les Architectures à Composants*. Thèse de doctorat, Ecole Nationale Supérieure des Mines de Paris, 2009.

[Mak09] Kristis Makris. *Whole-Program Dynamic Software Updating*. Thèse de doctorat, Arizona State University, 2009.

[MB09] Kristis Makris et Rida A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. *Dans USENIX Annual Technical Conference*, pages 397–410, 2009.

[MHH+11] Stephen Magill, Christopher M. Hayden, Michael Hicks, Nate Foster et Jeffrey S. Foster. Specifying and Verifying the Correctness of Dynamic Software Updates, 2011.

[MHN01] Jonathant Moore, Michael William Hicks et Scott M. Nettles. Dynamic Software Updating, Programming Language Design and Implementation. *Dans Programming Language Design and Implementation (PLDI)*, 2001.

[MPG+00] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr et J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. *Dans Fourteenth European Conference on Object-Oriented Programming*, pages 337–361. Springer-Verlag, 2000.

[MR07] Kristis Makris et Kyung Dong Ryu. Dynamic and adaptive updates of nonquiescent subsystems in commodity operating system kernels. *Dans ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 327–340, 2007.

[Mul09] Multos. http ://www.multos.com/. 2009.

[Myc03a] Sun Mycrosystems. Java CardTM 2.2.1 Application Programming Interface. SunMicrosystems, 2003.

[Myc03b] Sun Mycrosystems. Java CardTM 2.2.1 Virtual Machine JCVM) Specification. SunMicrosystems, 2003.

[Myc09a] Sun Mycrosystems. Java CardTM 2.2.1 Runtime Environment (JCRE) Specification. SunMicrosystems, 2009.

[Myc09b] Sun Mycrosystems. Java CardTM 3.0.1 Application Programming Interface Specification. SunMicrosystems, 2009.

[Myc09c] Sun Mycrosystems. Java CardTM 3.0.1 Runtime Environment (JCRE) Specification. SunMicrosystems, 2009.

[Myc09d] Sun Mycrosystems. Java CardTM 3.0.1 Servlet Specification, Connected Edition. SunMicrosystems, 2009.

[Myc09e] Sun Mycrosystems. Java cardtm 3.0.1 specification, classic edition. SunMicrosystems, 2009.

[Myc09f] Sun Mycrosystems. Java CardTM 3.0.1 Virtual Machine (JCVM) Specification. SunMicrosystems, 2009.

[NCL09a] Agnes C. Noubissi, Julien Cartigny et Jean-Louis Lanet. Carte à puce : attaques et contremesures. *Dans Majecstic*, 2009.

[NCL09b] Agnes C. Noubissi, Julien Cartigny et Jean-Louis Lanet. Carte à puce, vers une durée de vie infinie. *Dans Majecstic*, 2009.

[NCL09c] Agnes C. Noubissi, Julien Cartigny et Jean-Louis Lanet. Secure hotswapping, un réel problème pour carte à puce. *Dans Crypto-puce*, 2009.

[NCL10a] Agnes C. Noubissi, Julien Cartigny et Jean-Louis Lanet. Convergence Osgi – Javacard : fine – grained dynamic update. *Dans Eurosmart Smart Card Security Conference and Java Card, e-Smart'10*, 2010.

[NCL10b] Agnes C. Noubissi, Julien Cartigny et Jean-Louis Lanet. Incremental Dynamic Update For Java-based Smart Cards. *Dans Fifth International Conference on Systems*, pages 110 – 113, 2010.

[NCL11a] Agnes C. Noubissi, Julien Cartigny et Jean-Louis Lanet. EmbedDSU : Un framework de HotSwUp pour cartes à puce Java. *Dans Conférence Française en Systèmes d'Exploitation*, 2011.

[NCL11b] Agnes C. Noubissi, Julien Cartigny et Jean-Louis Lanet. Hot Updates for Java Based Smart Cards. *Dans IEEE 27th International Conference on Data Engineering Workshops*, pages 168 – 173. Third Workshop on Hot Topics in Software Upgrades HotSwUp'11, 2011.

[NEP08] Karsten Nohl, David Evans et Henryk Plötz. Reverse-Engineering a Cryptographic RFID Tag. *Dans Security Symposium, San Jose, CA.*, July 2008.

[NH09] Iulian Neamtiu et Michael Hicks. Safe and Timely Dynamic Updates for Multithreaded Programs. *Dans Programming Language Design and Implementation*, pages 13–24, 2009.

[NHFP08] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster et P. Pratikakis. Contextual Effects for Version-Consistent Dynamic Software Updating and Safe Concurrent. *Dans ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37–50, 2008.

[NHSO06] Iulian Neamtiu, Michael Hicks, Gareth Stoyle et Manuel Oriol. Practical Dynamic Software Updating for C. *Dans ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–83, 2006.

[Ni03] Zhang Lee Ni. Open Service Residential Gateway For Smart Homes, 2003.

[NM02] Natalya F. Noy et Mark A. Musen. PROMPTDIFF : A Fixed-Point Algorithm for Comparing Ontology Versions. *Dans Eighteenth National Conference on Artificial Intelligence*, pages 744–750, 2002.

[NP07a] Karsten Nohl et Henryk Plötz. MIFARE, Little Security, Despite Obscurity. *Dans 24th Congress of the Chaos Computer Club in Berlin*, December 2007.

[NP07b] Karsten Nohl et Henryk Plötz. MIFARE, Little Security, Despite Obscurity. *Dans Presentation on the 24th Congress of the Chaos Computer Club in Berlin*, December 2007.

[Ora96] Oracle. http ://www.oracle.com/technetwork/java/javame/javacard. *Oracle Corporation*, 1996.

[ORH02] Alessandro Orso, Anup Rao et Mary Jean Harrold. A technique for dynamic updating of Java Software. *Dans International Conference on Software Maintanence*, pages 649–658, 2002.

[PF07] Pierre Parrend et Stephane Frenot. Supporting the secure deployment of osgi bundles. *Dans First IEEE WoWMoM Workshop on Adaptive and DependAble Mission- and bUsiness-critical mobile Systems*, 2007.

[Pol08] Juraj Polakovic. *Architecture logicielle et outils pour systèmes d'exploitation reconfigurables.* Thèse de doctorat, Institut National Polytechnique de Grenoble, 2008.

[RA00] Tobias Ritzau et Jesper Andersson. Dynamic Deployment of Java Applications. *Dans Java for Embedded Systems Workshop*, 2000.

[Raz12] Razika Lounas and Mohamed Mezghiche and Jean-Louis Lanet . Une sémantique formelle pour la mise à jour dynamique des applications java card. *Dans AFADL 2012 (Soumis)*, 2012.

[RC00] Barry Redmond et Vinny Cahill. Iguana/J : Towards a Dynamic and Efficient Reflective Architecture for Java. *Dans ECOOP workshop on Reflection and Meta-Level Architectures*, 2000.

[RC02] Barry Redmond et Vinny Cahill. Supporting Unanticipated Dynamic Adaptation of Application Behaviour. *Dans 16th European Conference on Object-Oriented Programming*, pages 205–320, 2002.

[SHM09] Suriya Subramanian, Michael Hicks et Kathryn S. McKinley. Dynamic Software Updates in Java : A VM-centric approach. *Dans Programming Language Design and Implementation (PLDI)*, pages 1–12, June 2009.

[Sty02] Semiconductors Austria GmbH Styria. http ://www.mifare.net/. *Mifare Foundation*, 2002.

[Sub10] Suriya Subramanian. *Dynamic Software Updates : A VM-Centric Approach.* Thèse de doctorat, University of Texas at Austin, 2010.

[SYHK08] Holger Schmidt, Jon H. Yip, Franz J. Hauck et Rüdiger Kapitza. Decentralised Dynamic Code Management for OSGi. *Dans MiNEMA Workshop*, 2008.

[Van07] Yves Vandewoude. *Dynamically Updating Component-oriented Systems.* Thèse de doctorat, University of Leuven, 2007.

[Vis08] Eelco Visser. WebDSL : A Case Study in Domain-Specific Language Engineering. *Dans Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*. Springer, 2008.

[Wei91] Mark Weiser. The Computer for the Twenty-First Century. *Dans Scientific American*, 1991.

[Wik] Wikipédia. http ://en.wikipedia.org/wiki/jazelle. *Wikipedia Project.*

[XZW+03] Daqing Zhang Xiaohang, Daqing Zhang, Xiaohang Wang, Karianto Leman et Weimin Huang. Osgi based service infrastructure for context aware connected homes. *Dans 1st International Conference on Smart Homes and Health Telematics*, pages 81–88. IOS Press, 2003.