EmbedDSU : un Framework de HotSwUp pour Cartes à puce Java

Agnes C. Noubissi, Julien Iguchi-Cartigny et Jean-Louis Lanet

Université de Limoges,
83 rue d'Isle,
87000, Limoges - France
agnes.noubissi@xlim.fr, julien.cartigny@xlim.fr, jean-louis.lanet@xlim.fr

---

Résumé
Dans le but de corriger les vulnérabilités, de reparer des bugs ou d'améliorer des fonctionnalités, les applications ou les systèmes sont appelés à évoluer, et donc à être mis à jour. Le processus de mise à jour traditionnel consiste à stopper l'application ou le système complet, puis d'appliquer la mise à jour et enfin de redémarrer les élément éteints. Cependant cette approche n'est pas adaptée pour certains systèmes embarqués (bancaires, télécommunication, etc.) où il est nécessaire de ne pas stopper les services offerts même durant la mise à jour.
Une solution appropriée est la mise à jour dynamique (Hot Software Update : HotSwUp). Celle-ci consiste à appliquer la mise à jour en ligne c'est-à-dire sans stopper l'exécution du composant. Dans la littérature, de nombreux travaux ont été effectués dans ce domaine pour les systèmes natifs ou les systèmes s'exécutant au-dessus d'une machine virtuelle. Mais ces travaux ont été réalisés pour des configurations comme des postes de travail ou des serveurs. La mise en place d'un mécanisme de HotSwUp pour les petits objets embarqués notamment les cartes à puce, reste encore un défi au regard des contraintes de ressources et de sécurité.
Dans cet article, nous présentons EmbedDSU, un système de mise à jour dynamique des composants systèmes Java pour carte à puce, basé sur une machine virtuelle Java (Java Card). L'idée étant de limiter le processus de mise à jour uniquement aux parties affectées par la modification afin de réduire la complexité de l'opération dans la carte. EmbedDSU est basé sur un ensemble de mécanisme à l'extérieur de la carte (off-card) et à l'intérieur de la carte (on-card). En off-card, il s'agit de déterminer les différences syntaxiques entre deux classes, d'exprimer celles-ci dans un langage spécifique conçu à cet effet, et de transférer à la carte, le fichier résultant (DIFF). En on-card, il s'agit d'interpréter le fichier de DIFF et de mettre à jour la définition de la classe, les instances d'objets et les autres structures de données affectées.

Mots-clés : Java Card, mise à jour dynamique, machine virtuelle Java, carte à puce

---

1. INTRODUCTION

Les applications sont appelées à évoluer dans le but de fixer des défauts ou des bugs, d'améliorer et/ou supprimer des fonctionnalités. Une approche simple de mise à jour consiste à stopper l'application ou le système à mettre à jour, d'appliquer la mise à jour et de redémarrer la nouvelle version. Cependant, cette approche n'est pas appropriée pour les applications nécessitant de s'exécuter continuellement même durant la mise à jour. Une solution adaptée est la mise à jour dynamique (HotSwUp) qui consiste à mettre à jour une application sans stopper cette dernière, ni stopper le système sur lequel elle s'exécute. Dans ce papier, nous présentons EmbedDSU, une approche de mise à jour des applications et/ou composants systèmes sur une carte à puce de type Java Card. L'approche est basée sur la modification de la machine virtuelle Java embarquée dans l'objectif d'offrir les fonctionnalités de mise à jour dynamique. EmbedDSU est divisé en deux parties : off-card et on-card. En off-card, dans le but d'appliquer la mise à jour uniquement sur les parties affectées par la modification, un générateur de DIFF a été implémenté. Le générateur de DIFF détermine les modifications syntaxiques entre deux versions de classes et exprime les différences trouvées dans un langage dédié. On obtient ainsi un fichier DIFF. Ce fichier est ensuite transféré dans la carte et utilisé pour la mise à jour.

Sous certaines hypothèses spécifiques - notamment que EmbedDSU fonctionne avec les composants presentants une équivalence hiérarchique (arbre d'héritage) entre l'ancienne et la nouvelle version -, l'approche permet des mise à jour génériques des applications et composants systèmes Java et possède les propriétés suivantes :

1. Pas d'intervention humaine : le processus est totalement automatique. En effet, étant données deux versions d'une classe, le générateur de DIFF est capable de déterminer les différences syntaxiques entre les deux versions au niveau des interfaces, champs, signatures de méthodes, instructions, et des métadonnées des classes telles que les pools de constante, etc.
2. Support de plusieurs granularités internes de mise à jour : la mise à jour peut être effectuée au niveau des champs de la classe, des signatures des méthodes, des blocs d'instructions et des classes relatives.
3. Atomicité de la mise à jour : dans le but d'assurer la cohérence du système, la mise à jour est appliquée de façon atomique (un mécanisme de roll-back est prévu).
4. Préservation de la sémantique : après la mise à jour, les nouvelles versions des classes obtenues en utilisant le fichier de DIFF ont le même comportement que si elles avaient été transférées et que la résolution et l'édition des liens aient été effectués dans la carte.

Pour valider notre approche, nous avons implémenté et testé EmbedDSU à l'aide de SimpleRTJ [19], une machine virtuelle Java pour l'embarqué. Nous avons modifié SimpleRTJ afin d'offrir les mécanismes de mise à jour dynamique. Cet article présente notre motivation, explique EmbedDSU, décrit son implémentation, présente les résultats préliminaires obtenus et développe les travaux futurs.

2. MOTIVATION

Une carte à puce est un ordinateur de la taille et forme d'une carte de crédit, sur lequel est embarqué un ou plusieurs microcontrôleurs. Généralement, les microcontrôleurs pour cartes possèdent un microprocesseur possédant plusieurs types de mémoires : RAM (pour stocker les données et les piles de thread durant l'exécution), ROM (dans lequel le système d'exploitation et les applications sont gravées) et EEPROM (pour le stockage des données persistantes). En raison des contraintes de taille de la puce, la taille de la mémoire est faible. La plupart des cartes à puce vendues sur le marché aujourd'hui ont au plus 5 Ko de RAM, 256 Ko de ROM et 256 Ko d'EEPROM. La communication entre le lecteur et la carte est réalisée par une liaison série Half duplex avec un protocole appelé APDU (Application Protocol Data Unit). La carte agit comme un serveur et attend généralement les requêtes envoyées par le lecteur.

Les cartes à puce modernes intègrent une machine virtuelle Java qui interprète les applications préalablement gravées dans la ROM et aussi celles qui sont téléchargées dans l'EEPROM après émission de la carte (opération de Post-issuance). La norme Java Card 3.0 [6, 7, 8] est un exemple de spécification de la machine virtuelle pour carte à puce, disponible en deux éditions : édition classique ou édition connectée. La machine virtuelle Java Card (JCVM) interprète les applications Java et gère l'accès aux ressources de la carte à puce, servant ainsi de système d'exploitation à la carte. La JCVM est une machine virtuelle qui une fois instanciée sur la carte, devient une application qui ne s'arrête jamais. En effet, sa durée de vie est similaire à celle de la carte, car les objets persistants sont conservés même après l'expiration des sessions de communication avec le lecteur. Ainsi, la JCVM fonctionne en permanence et la mise à jour des composants doit être réalisée durant l'exécution.

La sécurité de la plateforme du point de vue logiciel repose sur plusieurs mécanismes. Tout d'abord, la possibilité de télécharger une application dans la carte est contrôlée par un protocole défini par Global Platform [18]. Ce protocole garantit que le propriétaire de l'application possède les droits nécessaires pour effectuer une telle action. Il définit aussi des domaines de sécurité pour chaque fournisseur d'application. En effet, Java Card est une plateforme ouverte, c'est-à-dire, qu'elle offre la possibilité de charger et d'exécuter de nouvelles applications après émission de la carte, et des applications appartenant à plusieurs fournisseurs peuvent coexister sur la même carte. Grâce à la vérification du système de type, les byte codes produits par le compilateur sont sans danger c'est-à-dire que l'application chargée ne peut pas interférer avec d'autres applications. En outre, la Java Card possède un pare-feu dont le but est de contrôler les autorisations entre les applications des domaines de sécurité différents et l'isolement de celles-ci lors de l'exécution.

Le langage Java Card est un sous ensemble de Java, sans multithread, avec un système de type simple (pas de flottant), tableau unidimensionnel, etc. Les applications sont relativement simples à développer et elles sont basées sur le modèle d'applet de Java Card. Actuellement, il est possible de mettre à jour des applications Java Card, en remplaçant l'ancienne version par une nouvelle version grâce au mécanisme de la post-issuance. Mais pour cela, l'application doit être arrêtée avant la mise à jour, ce qui est impossible pour les composants systèmes Java. Notre objectif est de mettre à jour dynamiquement un composant d'une API qui est gravé dans la ROM, ceci à travers une table de redirection en mémoire EEPROM.
Un cas réel d'application est le passeport électronique. En effet, dans le cas des E-passport, si une faiblesse dans un algorithme cryptographique est découvert, l'unique solution est de retourner tous les passeports à l'émetteur. Le problème de faiblesse d'algorithme n'est pas un cas hypothétique. En effet, des exploits réçents sur la carte à puce basée sur MIFARE [5] (une technologie de carte à puce sans contact basée sur la norme ISO 14443) ont été réalisés. Notamment, Plätz et Karsten [13, 14] ont décrit une méthode partielle de rétro-ingénierie pour la puce MIFARE classique. De même, Gans et Al.[15] ont présenté une technique permettant de manipuler le contenu d'une carte à puce MIFARE classique, de calculer des clés cryptographiques, le code PIN, ou de modifier certains tests de sécurité.
L'idée est de prévoir un mécanisme permettant de mettre à jour de façon dynamique les composants Java du système lorsque la carte est par exemple dans un lecteur ou à proximité d'un lecteur dans le cas des cartes sans contact.

3. EmbedDSU : ARCHITECTURE & IMPLÉMENTATION

Dans cette section, nous présentons le système EmbedDSU : l'architecture du système, les types de mise à jour supportés et sa mise en oeuvre.

3.1. Architecture

La figure 1 illustre le processus de mise à jour dynamique. La partie gauche représente le travail effectué à l'extérieur de la carte. En entrée, deux versions d'une classe à mettre à jour : l'ancienne et la nouvelle. Celles-ci sont transmises au générateur de DIFF afin de déterminer et d'exprimer les modifications syntaxiques, ce qui permet de générer un fichier de DIFF. Ce fichier de DIFF est ensuite transmis à l'intérieur de la carte et utilisé pour la mise à jour, comme représenté à droite.
A droite de la figure, nous présentons deux processus : Celui exécutant une version du composant en présentant l'état d'exécution composé des piles, du code et du tas, et l'autre exécutant la nouvelle version du composant. L'objectif est de passer d'un processus exécutant l'ancienne version à celui exécutant la nouvelle sans arrêter, ni redémarrer le composant, ceci grâce au fichier de DIFF calculé à l'extérieur de la carte.

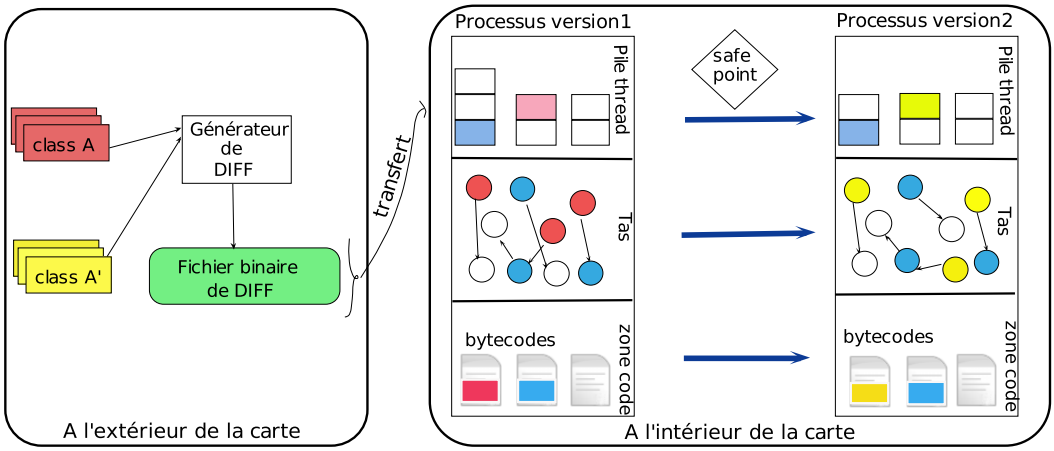

FIGURE 1 – Architecture EmbedDSU

Le générateur de DIFF compare syntaxiquement deux versions de classes et fournit une spécification pour la mise à jour. Le fichier de DIFF résultant peut comporter plusieurs parties notamment pour une classe donnée, les interfaces, les champs et les méthodes relatives aux modifications constatées entre les deux versions. Ces spécifications sont utilisées à l'intérieur de la carte pour :
– transformer le code des classes, les objets dans le tas, et les frames dans la pile de thread pour obtenir ceux correspondants à la nouvelle version,
– déterminer les points du système dit 'sûrs', durant lequel la mise à jour peut être effectuée en conservant la cohérence du système.

Détecter un point du système dit 'sûrs', passe par la restriction de certaines méthodes à apparaitre dans la pile de thread. Ces méthodes restreintes peuvent être, pour une classe donnée : (1) ses méthodes modifiées, (2) et les méthodes des classes relatives impactées par la mise à jour comme décrit plus loin dans la section 3.3.2.

3.2. Types de mise à jour supportés

EmbedDSU supporte les modèles de mise à jour les plus fréquents, pour une classe donnée. Il s'agit de la modification :
– Membres données : une mise à jour peut porter sur la modification des champs statiques ou d'instances d'une classe. Il peut s'agir plus précisément de la mise à jour des modificateurs (*public, private, protected, etc*), des types, de l'ordre, ou même juste des noms des membres données d'une classe.
– Signature de méthodes : il s'agit de la modification du type et du nombre d'arguments des méthodes de la classe. La modification de la signature d'une méthode d'une classe impacte les autres classes en relation avec cette dernière. En effet, ces classes peuvent faire appel à ces méthodes dont la signature a changé, et elles doivent donc être aussi mises à jour.
– Variable locale des méthodes : bien que le seul modificateur autorisé pour les variables locales est *final*, d'autres types de modifications peuvent survenir notamment au niveau du type de la variable.
– Bloc d'instructions : la modification du corps des méthodes est le modèle de mise à jour le plus couramment pris en charge par les systèmes de mise à jour existants.

3.3. Implémentation

3.3.1. A l'extérieur de la carte

La première étape consiste à calculer les changements entre deux versions (l'originale et la nouvelle) du module système à mettre à jour dans le but de déterminer les modifications effectuées. EmbedDSU utilise, pour produire un fichier de DIFF, un générateur de DIFF [1, 2]. Ce fichier contient les modifications syntaxiques entre les deux versions, exprimées dans un langage dédié conçu à cet effet. Ce fichier est ensuite interprété et formaté dans un format binaire afin de réduire la taille et réduire aussi le coût de l'interprétation dans la carte. Enfin, le fichier binaire est transféré à la carte pour réaliser la modification des structures de données et informations appropriées pour une mise à jour effective.

3.3.2. A l'intérieur de la carte

EmbedDSU est basé sur la modification de la machine virtuelle SimpleRTJ dans le but d'offrir des mécanismes de mise à jour dynamique. Cette modification inclut l'ajout de plusieurs modules tels que :

1. Module d'interprétation du fichier de DIFF qui permet d'interpréter le fichier de DIFF et d'initialiser les structures de données nécessaires à la mise à jour. Celles-ci seront par la suite utilisées par les autres modules.
2. Module d'introspection qui fournit des fonctions de recherche à travers les structures de données de la machine virtuelle telles que la table de référence, la table de threads, la table de classes, la table des objets statiques, le tas de la machine virtuelle et bien sûr la pile de thread. L'objectif étant de récupérer les informations nécessaires à d'autres modules.
3. Module de détection de *Safe point* permettant de détecter l'instant auquel on peut appliquer la mise à jour en préservant la cohérence du système.
4. Module de mise à jour, module permettant de modifier les instances d'objets dans le tas, modifier les blocs d'instructions dans le corps des méthodes, les métadonnées des classes, les références d'objets dans la table de référence, les registres affectés dans la pile de thread et d'autres structures de données affectées à la machine virtuelle.

5. Module de restauration (*Roll-back*) qui permet d'obtenir la version précédente du code, des instances, de la pile de thread et des structures de données de la machine virtuelle en cas de non-atomicité de la mise à jour (arrachage de la carte dans le lecteur par exemple).

Tous ces modules interagissent pour fournir des fonctionnalités de mise à jour dynamique durant l'exécution.
Nous avons aussi modifié la machine virtuelle pour qu'elle puisse fonctionner sous trois modes d'exécution.

– Le mode standard : mode d'exécution normal avant la détection d'une nouvelle version de l'application c'est-à-dire avant la détection d'un fichier de DIFF,
– Le mode de recherche d'un point 'sûr' : mode d'exécution après réception du fichier de DIFF et avant la détection d'un point *sûr*.
– Le mode de mise à jour : mode après détection du *safe point* durant lequel la mise à jour est effectuée.

Mode standard
C'est le mode de fonctionnement normal de la machine virtuelle. Dans ce mode, la machine virtuelle fonctionne relativement sans coût supplémentaire dû au processus de HotSwUp. Après réception d'un fichier DIFF de mise à jour par le gestionnaire de carte, et après vérification et interprétation du fichier, la machine virtuelle passe dans le mode de recherche d'un point 'sûr' du système.

Mode de recherche d'un point 'sûr'
Pour obtenir un point *sûr*, il est nécessaire d'introspecter l'état d'exécution du composant à mettre à jour.
Nous avons défini l'état d'exécution comme composé de :

– Classes : il s'agit de l'ensemble des classes affectées par la mise à jour (les byte codes des méthodes, et les meta-données)
– Frames : il s'agit des frames, dans la pile de frames, relatives aux méthodes à mettre à jour. Sachant qu'à chaque appel de méthode, un frame associé est créé dans la pile de thread et contient la table de variables locales, les paramètres de méthodes, le registre de compteur d'instructions, etc.
– Instances : il s'agit de l'ensemble des instances actives de la classe à mettre à jour existant dans le tas de la machine virtuelle. Les instances actives correspondant à celles qui sont encore référencées dans un frame sur la pile de thread ou référencées par un champ d'une autre instance active.

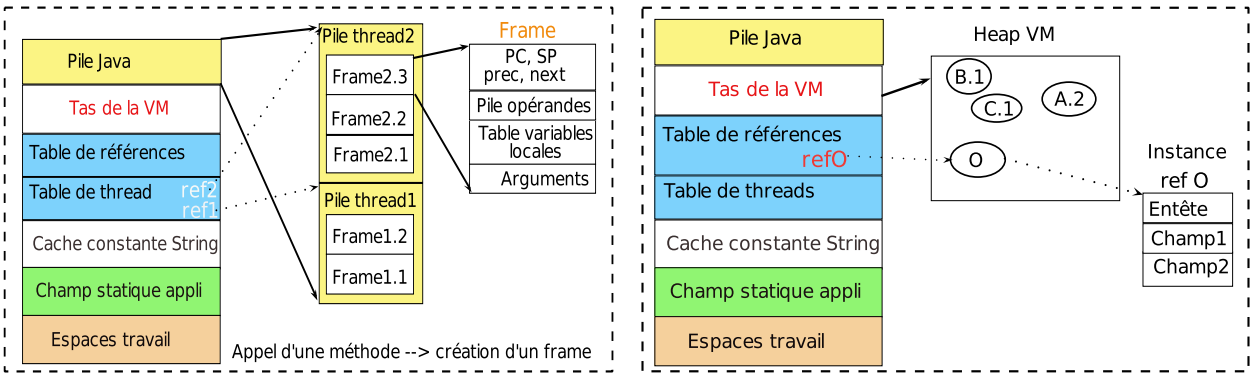

FIGURE 2 – Vue sur les structures de données de SimpleRTJ, la pile de thread et le tas

Afin de préserver la cohérence et la stabilité du système, il est nécessaire de détecter le moment opportun, un point 'sûr' durant lequel la mise à jour peut être faite. Détecter un point 'sûr' passe par la restriction de certaines méthodes (voir 3.3.2) à apparaître dans la pile de thread. Cette restriction permet d'obtenir une propriété forte de mise à jour dynamique : aucun code de la nouvelle version ne peut s'exécuter avant que la mise à jour soit complète et aucun code de l'ancienne version ne doit s'exécuter

après. Cette restriction permet de préserver le système de type en s'assurant que les méthodes modifiées ou impactées par la mise à jour n'accèdent pas de façon inappropriée à un champ d'une instance ou à un espace mémoire non dédié.
Le méthodes restreintes se divisent en deux catégories :
– Celles dont le bytecode a été modifié.
– Celles dont le bytecode n'a pas été modifié mais qui accèdent à une classe à mettre à jour, soit aux instances d'une classe à mettre à jour.

En effet, supposons que l'une des méthodes restreintes est présente sur la pile de thread avant la mise à jour, et qu'après la mise à jour, cette méthode continue son exécution et :
– soit fait appel à une méthode supprimée dans la nouvelle version de la classe et donc inexistante,
– soit accède à un champ supprimé de la classe et donc un champ inexistant dans l'instance de la classe en mémoire,
– soit effectue une opération inappropriée sur un champ dont le type a été modifié, étant donné que durant la mise à jour, toutes les instances d'objets de la classe ont été mises à jour (dans le cas où des champs de la classe ont été modifiés).
Dans ces exemples cités, on obtient une défaillance du système.

Détecter un point 'sûr' du système consiste donc à déterminer le moment où il n'existe pas dans la pile de thread, une méthode restreinte. Pour cela, la machine virtuelle passe en mode de détection. Dans ce mode, le module de détection introspecte la pile dans le but d'obtenir le nombre de méthodes restreintes présentes. S'il en existe, la mise à jour est reportée. A la fin d'une méthode restreinte, c'est-à-dire, après la libération de l'espace mémoire du frame associé à ladite méthode, le nombre de méthodes restreintes préalablement calculé est décrémenté et lorsque ce dernier atteint zéro, un point 'sûr' est atteint et la machine virtuelle peut donc passer dans le mode de mise à jour.

Déterminer ce point sûr passe inévitablement par l'introspection des structures de données de la VM. Dans la figure 2, nous présentons les structures de données de SimpleRTJ. Entre autres, la table de référence contient toutes les références aux instances d'objets présentes dans le tas de la machine virtuelle qui ont été crées à partir de l'opérateur New et les références aux objets systèmes. La table de thread contient les adresses de début des piles de thread créés dans la pile Java. Et chaque pile de thread est une liste doublement chainée de frames. Ainsi, à partir de la table de thread, on peut accéder à la pile de thread correspondant. Ensuite, parcourir la pile pour déterminer les frames associés aux méthodes restreintes rélatives aux méthodes modifiées listées dans le fichier de DIFF, ou à celles ayant une référence sur des instances d'objets de la classe à mettre à jour.

Mode de mise à jour
Dans ce mode, la mise à jour est réalisée de façon atomique. L'objectif est de modifier l'état d'exécution du composant pour obtenir un état correspondant à la nouvelle version.

1. Mise à jour du code
   Le processus de mise à jour à l'intérieur de la carte commence par la mise à jour du code du composant avec une granularité qui est la classe. Dans la figure 3, la structure d'un composant Java est présentée. Mettre à jour une classe donnée consiste ainsi à modifier à partir du contenu du fichier de DIFF :
   – Les métadonnées du composant ou de l'application,
   – Le constant pool de la classe, et ses métadonnées (offset dans la table de méthode, table de champs),
   – L'entête de méthode (offset, nombre et type des variables locales) et le bytecode de la liste des méthodes modifiées.
   Cette mise à jour consiste à lire les structures de données initialisées lors de l'interprétation de la DIFF, ensuite copier en modifiant l'ancienne version de la classe vers un nouvel espace mémoire dans l'EEPROM réservé à cet effet, dans le but d'obtenir la nouvelle version de la classe.
2. Mise à jour des pools de constantes des classes affectées
   Après la mise à jour du code de la classe, le processus de mise à jour continue avec la mise à

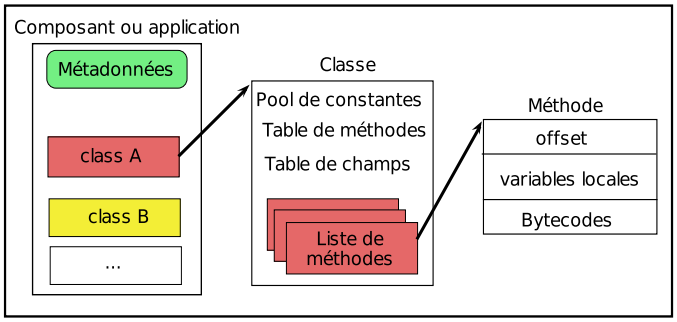

FIGURE 3 – Structure d'un composant Java dans SimpleRTJ

jour des pools de constantes des autres classes affectées. Il s'agit principalement, de la mise à jour des offsets des méthodes, des champs dans la table de constantes, la table de méthodes et la table des champs des classes affectées. Si le bytecode des méthodes d'une de ces classes affectées a été modifié, alors il sera en effet mis à jour, lors de la mise à jour du code de la classe relative.

Ensuite, il est nécessaire de mettre à jour les instances de l'ancienne version de la classe.

3. Mise à jour des instances

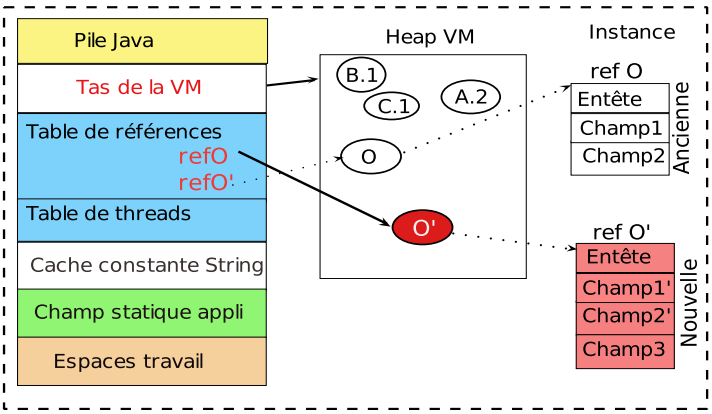

FIGURE 4 – Mise à jour des instances dans le tas

Pour la mise à jour des instances, l'approche du ramasseur de miettes Java est utilisée. Il s'agit de parcourir le tas de la machine virtuelle à partir de la racine de persistance (table de référence, variables locales dans les frames, champs statiques, etc.) afin de rechercher toutes les instances de la classe à mettre à jour.

Comme indiqué sur la figure 4, pour chaque instance d'objet trouvé O appartenant à la classe C, il s'agit d'allouer un nouvel espace mémoire pour O' de taille appropriée à une instance de la nouvelle classe C'. Ensuite, le processus continue par :

– l'initialisation de l'entête de l'objet O' pour qu'elle corresponde à une instance de C',
– l'initalisation des champs non modifiés avec les valeurs correspondantes de O,
– l'intialisation des champs modifiés avec les valeurs obtenues en off-card et contenues dans le fichier de DIFF,
– et enfin la sauvegarde de l'ancienne et la nouvelle référence dans la table de référence sachant que l'ancienne référence pourra être utilisée dans le processus de roll-back.

4. Mise à jour des frames

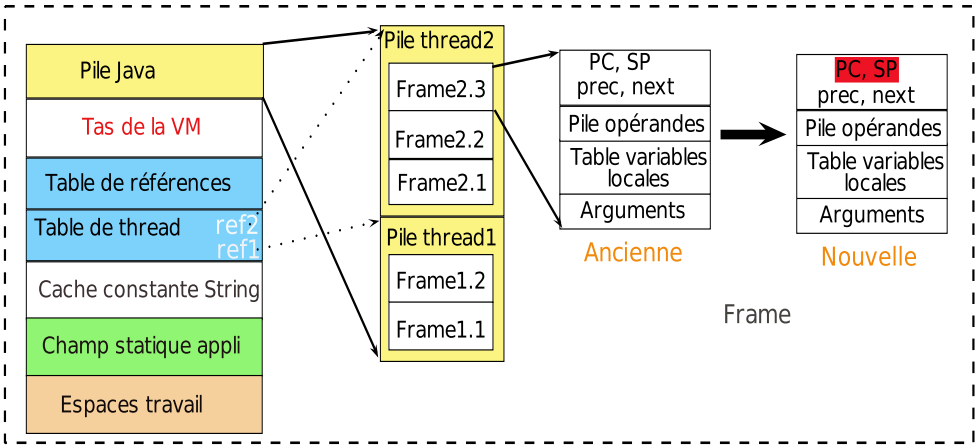

FIGURE 5 – Mise à jour des frames

La figure 5 présente le processus de mise à jour des frames, qui consiste principalement à parcourir la table de threads et déterminer la pile de threads concernée. Ensuite, il s'agit de parcourir la pile de threads et pour chaque frame associé à une méthode de la classe, de modifier les adresses de retour de la méthode pour qu'elle pointe sur l'offset correspondant à l'instruction dans le bytecode de la méthode correspondante située dans la nouvelle zone mémoire de la classe.

Roll-Back

Le module de roll-back permet de retrouver le moment où s'arrête la mise à jour, les données telles que :
- les anciennes versions des instances, à partir des anciennes références sauvegardées,
- l'ancienne version du code en changeant toutes les références dans les pools de constantes des classes affectées pour qu'elles pointent sur les anciennes adresses,
- les anciennes versions des références et des adresses de retour dans la pile de threads,
- et les autres structures de données de la machine virtuelle ayant été modifiées.

4. BENCHMARKS PRÉLIMINAIRES

Dans le but de déterminer le coût d'exécution du processus de mise à jour des instances, nous avons comparé le coût d'un ramasse-miettes normal au coût de la mise à jour des instances. L'objectif étant de déterminer et comparer le coût durant la mise à jour des instances lors de l'ajout, de la suppression ou de la modification d'un ensemble de champs dans la classe à mettre à jour. La table 1 décrit les différentes versions d'instances ou configurations utilisées pour les benchmarks préliminaires.
La mise en place des *benchmarks* est réalisée en se basant sur 4 versions de la classe : une version initiale, une version avec les champs ré-ordonnés, une version avec des champs ajoutés et une version avec des champs supprimés. La classe d'origine contient 4 champs (deux entiers, un short et une référence à un objet initialisé à *null*). La mesure s'effectue en faisant varier le nombre d'objets pour chaque configuration. Initialement, 400 objets de la classe à mettre à jour sont crées, et pour chaque fraction entre 0% et 100% ou chaque série de 40 objets, 25 exécutions pour chaque configuration est réalisée et la moyenne des temps obtenus est utilisée. Le même processus est effectué pour le ramasse-miettes (GC) mais sans objets *morts*. La figure 2 présente quelques résultats obtenus.

Initialement	Champs re-ordonnés	Ajout champs	Suppression champs
Class C { int var1 ; int var2 ; short var3 ; Object var4 ; }	class C' { int var2 ; short var3 ; int var1 ; Object var4 ; }	class C' { *short var5 ;* int var1 ; *int var6 ;* int var2 ; short var3 ; Object var4 ; }	class C' { int var1 ; short var3 ; }

TABLE 1 – Les versions d'instances utilisées

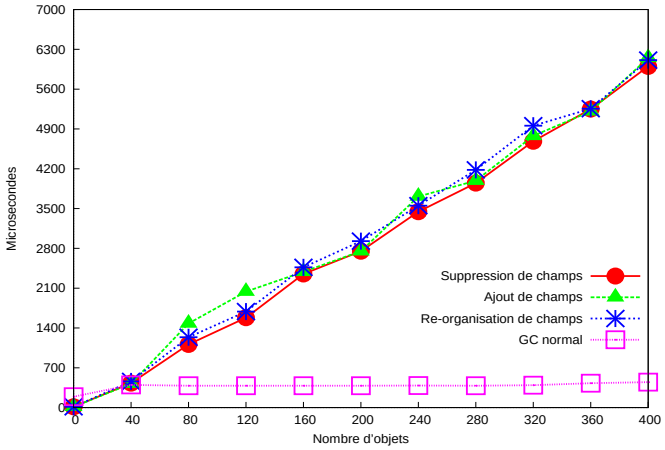

FIGURE 6 – Résultats de mise à jour obtenus en modifiant les champs d'une classe, comparé au GC

Le GC normal est exécuté sans objets *morts* dans le tas de la machine virtuelle, donc sans surcoût supplémentaire dû par la libération de la mémoire. Dans ce cas, le GC est optimisé tandis que la mise à jour des instances ont des surcoûts supplémentaires dûs aux fonctions tels que la création et l'initialisation des nouvelles instances de la classe avec les valeurs fournies en off-card et présentes dans le fichier de DIFF. Ces fonctions rendent le processus plus coûteux par rapport à ce GC normal.
D'autre part, on constate que l'ajout des champs est l'opération la plus coûteuse comparée aux autres types de modifications, ceci dû au coût de la recherche d'un espace mémoire approprié de taille correspondant à celle d'une instance de la nouvelle version de la classe, à cela s'ajoute le coût de la copie des anciens champs et l'initialisation des nouveaux champs par les valeurs initiales fournies provenant du fichier de DIFF.
La réorganisation des champs est également coûteux mais moins coûteux par rapport à l'opération d'ajout de champs, ceci dû au fait que la lecture des valeurs initiales fournies par le fichier de DIFF n'est pas réalisée.
La suppression des champs reste l'opération la moins coûteuse, les champs non supprimés sont copiés dans le nouvel espace mémoire réservé à la nouvelle instance, et cette recopie est réalisée en omettant les champs supprimés, en plus l'initialisation des champs avec les valeurs fournies par le fichier de DIFF n'est effectuée que dans certains cas.

5. TRAVAUX RELATIFS

De nombreux systèmes de mise à jour dynamique ont été réalisés. Notamment, dans le domaine Java, on peut citer entre autres : JVOLVE [9], JDRUMS [10], DVM [11] et les travaux d'Orso et Al. [10].
Orso et An. [12] proposent d'implémenter la DSU sans modifier la machine virtuelle en utilisant un

proxy pour chaque classe à mettre à jour. Pour chaque classe C, une classe proxy CP est proposée. Tous les appels vers les méthodes de C passent un appel dans CP qui lui se charge d'appeler la méthode de C. Ainsi, la mise à jour d'une classe passe par la mise à jour de la classe proxy et donc un appel d'une méthode de C, passera par CP qui appellera à son tour, la méthode correspondante dans C'. Et de nouveaux objets de C' sont créés et mis à jour grâce aux anciennes instances de C. Cette approche nécessite à la classe d'exporter les mêmes interfaces publiques et restreint la mise à jour. En effet, on ne peut ajouter d'appels à des méthodes publiques qui ne sont pas présentes dans les interfaces importées dans l'ancienne version.

DVM [11] et JDRUMS [10] implémentent la mise à jour dynamique en modifiant la machine virtuelle mais pas de façon similaire à EmbedDSU. En effet, les auteurs de DVM proposent d'étendre le chargeur de classe Java pour permettre le remplacement d'une définition de la classe et la mise à jour des objets instanciés. Pour cela, ils définissent deux méthodes principales : reloadClass () et replaceClass () permettant respectivement de charger la nouvelle version de la classe et de mettre à jour les instances de la classe en cours. JDRUMS, quant à lui, utilise les tables d'indirections pour remplacer les anciennes versions des classes par les nouvelles en modifiant les adresses de location des classes. Le principal inconvénient de ces techniques est qu'on observe un surcoût même durant le mode standard de la machine virtuelle. Dans EmbedDSU, en mode normal de la machine virtuelle, aucun coût supplémentaire n'est observé.

JVOLVES [9] est un système implémentant la DSU grâce à Jikes RVM, une machine virtuelle étendue intégrant des services de mises à jour des applications Java. Il présente une interface similaire à EmbedDSU, en ce sens où, il possède un outil appelé UPT (*Update Preparation Tool*) permettant de déterminer entre l'ancienne version et la nouvelle version d'une application :
– les fichiers classes modifiés,
– les fichiers classes supprimés,
– et ceux ajoutés.
Et à partir de là, la mise à jour n'est effectuée qu'avec les fichiers classes modifiés et ajoutés. En plus, elle associe un fichier permettant la transformation d'état pour la mise à jour des instances. EmbedDSU est proche de JVOLVE par contre au lieu de travailler au niveau fichier, nous avons travaillé au niveau code avec pour granularité, la classe. Ainsi, au lieu de comparer deux applications pour détecter les fichiers modifiés, nous avons comparé deux classes pour déterminer les modifications internes plus précisément les modifications syntaxiques. L'objectif étant de transférer une quantité réduite d'information au vue des contraintes de ressources en terme de stockage sur la carte. En plus, JVOLVE ajoute un compilateur JIT (Just-In-Time), ce qui augmente les coûts de mise à jour.
Généralement, ces implémentations sont destinées à des plateformes possédant les caractéristiques minimales d'un poste de travail (128 Méga-octets de Ram, 50 Giga-octets de mémoire persistante, et 500 Mhz de processeur) qui ne peuvent se comparer aux ressources restreintes des cartes à puce.
Actuellement, dans le domaine des cartes à puce en général, la mise à jour d'une application consiste à un retrait et à un chargement de la nouvelle version de l'application. Aucune mise à jour des composants systèmes ne peut encore être effectuée.

6. TRAVAUX FUTURS

6.1. Transfert d'état

Actuellement, dans EmbedDSU, la stratégie d'initialisation des champs ajoutés des instances consiste à récupérer la valeur (constante) calculée off-card et fournie dans le fichier de DIFF et de le copier dans l'espace correspondant au champ à mettre à jour. Cependant, il peut exister des cas où la valeur initiale est le résultat de l'exécution d'une méthode donnée ou d'une expression à évaluer durant l'exécution. Et donc, il est nécessaire de fournir une initialisation dynamique des champs des nouvelles instances durant le processus de mise à jour. Pour cela, l'idée est de fournir des fonctions de transfert en entrée du processus de mise à jour.
La mise en œuvre de cette fonctionnalité passe par deux fichiers : un fichier de DIFF et un fichier de transfert d'état fourni par les développeurs. Sachant que les développeurs ont la meilleure compréhen-

sion de l'évolution des changements de leurs applications ou composants applicatifs.

6.2. Méthodes proxies

EmbedDSU permet une mise à jour d'une classe et les classes affectées de façon atomique. S'il y a un nombre important de classes à mettre à jour, cette approche peut rendre indisponible la carte durant un certain moment (le temps d'attente maximale pour un utilisateur de carte étant environ de 3 secondes). Par conséquent, l'idée est de se pencher sur les classes affectées, de faire une mise à jour totale des classes affectées possédant des méthodes supprimées et effectuer une mise à jour partielle de celles contenant uniquement des méthodes modifiées, en particulier celles dont la signature a été modifiée. Et pour cela, fournir des méthodes proxies permettant de passer d'un appel de méthode dont la signature a changé vers la nouvelle version de la méthode sans modifier le code de l'appel au niveau de la classe affectée. Les configurations suivantes de modification de signatures de méthodes sont celles concernées et les méthodes proxies associées :

– Modification de l'ordre des paramètres : → RT m(T2 p2, T1 p1), on a :

```
RT m (T1 p1, T2 p2) {
  return m(p2, p1);
}
```

– Suppression de paramètres : → RT m(T2 p2), on a :

```
RT m (T1 p1, T2 p2) {
  return m(p2);
}
```

– Ajout de paramètres : → RT m(T1 p1, T2 p2, T3 p3)
Ce dernier cas est le plus difficile à traiter, puisqu'une valeur initiale ne peut pas toujours être fournie pour les paramètres ajoutés. En effet, pour les paramètres comme un mot de passe, un code secret ou un code PIN, si ce type de paramètre est ajouté, il est difficile de prédire la valeur par défaut à affecter. Mais dans certains cas où l'initialisation est possible, nous pouvons avoir une méthode proxy comme celle-ci :

```
RT m (T1 P1, T2 P2) {
  T3 p3 = null;
  return m(p1, p2, p3);
}
```

7. CONCLUSION

Dans cet article, nous présentons le framework EmbedDSU, une technique de mise à jour dynamique de code dans le contexte de carte à puce Java. Nous présentons son architecture, son implémentation, les benchmarks et les travaux futurs. EmbedDSU est un framework de HotSwUp basé sur la modification de la machine virtuelle SimpleRTJ, et est subdivisé en plusieurs parties : off-card et on-card. Après implémentation, les benchmarks présentés ont été réalisés sur un poste de travail.
Cependant, afin d'avoir des benchmarks plus réaliste, actuellement, des travaux de transferts de la machine virtuelle sur une board d'évaluation AT91 EB40 [16] sont en cours. L'idée étant d'approfondir les benchmarks pour obtenir les temps d'exécution à des niveaux plus fin, tels que le temps de transfert du fichier de DIFF, le temps d'interprétation du fichier de DIFF, les coût en consommation électrique et surtout les quantités de mémoires consommées tant au niveau de la RAM qu'au niveau de l'EEPROM.

Bibliographie

1. Agnes C. Noubissi, Jean-Louis Lanet et Julien Iguchi-Cartigny – Incremental Dynamic Update For Java Smart Cards Applications – ICONS'10, France, April 2010.
2. Agnes C. Noubissi, Julien Cartigny et Jean-Louis Lanet – Hot Updates for Java Based Smart Cards – Third Workshop on Hot Topics in Software Upgrades HotSwUp'11, Hannover - Germany, April 2011
3. Agnès C. Noubissi, Julien Iguchi-Cartigny et Jean-Louis Lanet – Carte à puce, vers une durée de vie infinie – MajecStic, Avignon, 2009.

4. Jonathan T. Moore, Michael Hicks et Scott Nettles – Dynamic software Updating, Programming Language Design and Implementation – PLDI, ACM, 2001.
5. Semiconductors Austria GmbH Styria – http ://www.mifare.net/.
6. Zhiqun Chen – Java Card Technology for Smart Cards – Addison, Wesley, 2000
7. The Java Card 3.0 specification : http ://java.sun.com/javacard/
8. Milan Fort – Smart card application development using Java Card Technology – SeWeS 2006.
9. Suriya Subramanian, Michael Hicks et Kathryn S. McKinley – Dynamic Software Updates : A VM-Centric Approach – PLDI, June 2009.
10. Jesper Andersson et Tobias Ritzau – Dynamic deployment of Java applications, Java for Embedded Systems Workshop – London, May 2000.
11. Earl Barr, J. Fritz Barnes, Jeff Gragg, Raju Pandey et Scott Malabarba – Runtime support for type-safe dynamic java classes – ECOOP, 2000.
12. Alessandro Orso, Anup Rao, et Mary Jean Harrold – A technique for dynamic updating of Java Software – ICSM, 2002.
13. Karsten Nohl, David Evans, Starbug et Henryk Pltâz – Reverse-Engineering a Cryptographic RFID Tag – USENIX Security Symposium, San Jose, CA. July 2008.
14. Karsten Nohl et Henryk Pltz – MIFARE, Little Security, Despite Obscurity – Presentation on the 24th Congress of the Chaos Computer Club in Berlin, December 2007.
15. G. de Koning Gans, Jaap-Henk Hoepman et Flavio D. Garcia – A Pratical Attack on MIFARE Classic – CARDIS, 2008.
16. ATMEL Corporation : http ://www.atmel.com/products/AT91/
17. Apache : http ://jakarta.apache.org/bcel/
18. Global Platform : http ://www.globalplatform.org/
19. SimpleRTJ : http ://www.rtjcom.com/
20. Arie van Deursen, Paul Klint et Joost Visser – Domain-specific languages : an annotated bibliography – June 2000.
21. G. Thomas, N. Geoffray, C. Clement, and B. Folliot – Designing highly flexible virtual machines : the jnjvm experience. Software Practice Expererience, 2008.
22. Bertil Folliot, Ian Piumarta, Fabio Riccardi – A Dynamically Configurable, Multi-Language Execution Platform. Proc. of 8th ACM SIGOPS European Workshop, Sintra, Portugal, September 1998.