MajecSTIC 2009
Avignon, France, du 16 au 18 novembre 2009

Carte à puce, vers une durée de vie infinie

Agnès Cristèle Noubissi, Julien Iguchi-Cartigny et Jean-Louis Lanet

Université de Limoges, Laboratoire XLIM, Equipe SSD, 83 rue d'Isle, 87000 Limoges – France

{agnes.noubissi, julien.cartigny,jean-louis.lanet}@xlim.fr

Résumé
Nous présentons dans cet article nos travaux de recherche concernant la mise à jour dynamique des applications pour la Java Card 3.0 *connected edition*. Par dynamique, nous entendons le fait de *patcher* des classes de l'application en cours d'exécution sans arrêter ni la machine virtuelle Java Card, ni l'application elle même. De plus, cette opération doit se dérouler de manière transparente pour les autres applications de la carte. Nous montrons que cette approche nécessite l'ajout de plusieurs mécanismes *off-card* et *on-card* afin de pouvoir garantir la fiabilité et la sécurité de l'opération (atomicité de la mise à jour, cohérence du système de type, etc...). Nous présentons une architecture pour étendre le *Card Manager* de la carte afin de pouvoir intégrer la processus de mise à jour dynamique dans le cycle de vie d'un module d'application dans le cadre de la norme Java Card 3.0 *connected edition*.

Abstract
We present in this paper our research on dynamic update of applications for Java Card 3.0 connected edition. By dynamic, we mean the patch of classes of the running application without stopping nor the virtual machine, nor the application itself. Moreover, this operation must be done in transparent manner for the others cards applications and for the user's card. We show that this approach requires the addition of several mechanisms off-card and on-card to ensure the reliability and security of transaction (atomicity of update, consistency of system type, …). We present an architecture to expand the Card manager of the card to be able to integrate the process of dynamic update in the life cycle of an application module as part of the Java Card 3.0 connected edition standard.

Mots-clés : Mise à jour dynamique, Java Card 3.0, Carte à puce.

Keywords : Dynamic upgrade, Java Card 3.0, Smart card.

---

1. Introduction

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. En effet, la carte à puce aujourd'hui a une durée de vie de plus en plus élevée. On peut citer par exemple, le cas du passeport électronique dont la durée de validité est de dix ans. De même, la carte bancaire est limitée par le nombre de transactions de la carte, et une carte peut théoriquement avoir une validité plus importante (même si dans la réalité, le système bancaire limite de manière logicielle cette validité).
Il est difficile d'imaginer qu'une carte à puce puisse résister à de nouvelles attaques (physiques, logiques ou cryptographiques) sur une longue période, d'où la nécessité de mettre à jour les applications embarquées. Cependant, il existe un obstacle important au support d'un tel mécanisme dans les cartes multi-applicatives: une application peut offrir des services utilisées par d'autres applications dans la carte. De plus, certaines applications ne s'arrêtent jamais. Il faut donc pouvoir mettre à jour le code dynamiquement (*Dynamic Software Upgrade* ou DSU) et de façon transparente pour l'utilisateur et les autres applications. Dans cet article, nous présenterons une approche de DSU pour des applications Java Card dans le cas de la nouvelle spécification Java Card 3.0 *connected edition* [6].

2. Présentation de la Java Card 3.0

La Java Card est une carte à puce possédant une plateforme basée sur la technologie Java. Elle est dite « ouverte » car elle permet de charger, d'exécuter et de supprimer des applications après l'émission de la carte. Dans la version minimale de Java Card 3.0 (*classic edition*), la puce possède une machine virtuelle permettant l'exécution d'applications (*applets*) écrites en langage Java Card (sous ensemble de Java) . Ces applets sont chargées dans un format de fichier appelé CAP (*Converted APplet*) et la communication avec le lecteur est effectuée grâce au protocole APDU (*Application Protocol Data Unit*) [9]. La version plus évoluée (Java Card 3.0 *connected edition*) possède en plus un serveur web embarqué et la possibilité de charger les applications web Java Card sous format de fichier *jar* en utilisant le protocole HTTP pour la communication. Dans la suite de cet article, nous parlerons uniquement de cette dernière version.

Composants matériels

Une carte à puce Java possède les caractéristiques suivantes:

- Une mémoire ROM (*Read Only Memory*) de 512 kilo-octets sur laquelle est gravée le système d'exploitation et la machine virtuelle Java Card de la carte.
- Une mémoire EEPROM (*Electrical Erasable Programmable Read Only Memory*) de 128 kilo-octets contenant les applications et les données de la carte.
- Une mémoire RAM (*Random Access Memory*) qui est un espace de 24 kilo-octets contenant la pile d'exécution de la machine virtuelle et une zone de stockage temporaire.
- Un processeur 32-bits basé sur une architecture CISC.

Communication avec l'extérieur

La Java Card 3.0 est définie dans la norme [6] comme une carte à puce pouvant être *combi* ou hybride car elle peut posséder une ou plusieurs interfaces, à la fois sans contact ou avec contact.

Architecture Java Card

Au cœur de la plateforme Java Card se trouve la machine virtuelle Java Card (JCVM) , sous ensemble de la machine virtuelle Java classique. Elle fonctionne généralement sur un système d'exploitation hôte minimal permettant de gérer les ressources du système. Au dessus de la JCVM se trouve l'environnement d'exécution Java Card (JCRE) responsable de la gestion sécurisée du cycle de vie des applications qui peuvent être des applets classiques, étendues ou des applications web. Ces applications sont généralement appelées modules d'application.

3. Travaux relatifs

Pour faire face au problème de mise à jour dynamique des applications Java, plusieurs travaux ont été effectués [10, 13, 14].

JVOLVES [10] est un système implémentant la DSU grâce à Jikes RVM, une machine virtuelle étendue intégrant des services de mises à jour des applications Java.

DVM [13] (Dynamic Virtual Machine) implémente la DSU en modifiant la machine virtuelle. Les auteurs proposent d'étendre le chargeur de classe Java pour supporter le remplacement d'une définition de classe et la mise à jour des objets instanciés. Ils définissent deux nouvelles méthodes : *reloadClass* et *replaceClass* permettant de charger la nouvelle version de l'application et de mettre à jour l'instance de l'application en cours.

Orso et An. [14] proposent d'implémenter la DSU sans modifier la machine virtuelle en utilisant un *proxy* pour chaque classe à mettre à jour.

Généralement, ces implémentations étaient 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). Or la Java Card ne possède que 24 kilo-octets de RAM, 128 kilo-octets de mémoire persistante et une vitesse de traitement limitée (moins de 50 Mhz de processeur). Au vu du fossé qui sépare ces deux types de plateformes, ces travaux ne sont à priori pas directement adaptables pour la Java Card.
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 applications ne peut encore être effectuée de façon dynamique.

4. Mise à jour dynamique des applications Java Card

Dans une Java Card 3.0, un module d'application possède un cycle de vie bien défini. 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 ou une classe d'une application est demandé, une instance de l'application est créée. Dès que l'exécution est terminée ou si une demande explicite de destruction est effectuée, l'instance de l'application peut être supprimée. Enfin, une application peut être tout simplement déchargée de la Java Card. On peut donc résumer le cycle de vie d'une application comme suit : (1) Chargée ; (2) Instanciée ou active ; (3) Supprimée (suppression de l'instance) ; (4) Déchargée (suppression de l'application).

On remarque plusieurs défauts avec ce cycle de vie lors de la *DSU*. Premièrement, la mise à jour d'une application ou d'un module d'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 la Java Card, cela suppose un arrêt des autres applications utilisant ce service. Cette obligation peut être coûteuse en temps processeur mais peut aussi impacter l'utilisateur de la carte. L'ampleur de ce problème est encore plus grande à une échelle plus importante, où plusieurs millions d'utilisateurs possèdent la Java Card avec des applications à mettre à jour appartenant à des entités différentes. D'où la nécessité d'une mise à jour dynamique des applications Java Card.

4.1. Une solution spontanée

Une première solution de mise à jour est de remplacer dans la mémoire persistante l'ancienne version d'un module d'application par la nouvelle version sans décharger l'application. Cependant cette solution n'a rien de dynamique car elle nécessite l'arrêt du service offert. De plus, cette solution se heurte à d'autres problèmes :

1. Les contraintes matérielles de la carte: taille de la mémoire persistante, données stockées

de manière séquentielle, il n'existe pas de mécanisme de défragmentation.

2. L'architecture orientée service de la Java Card: les applications accédant à un service modifié doivent être mises à jour, ce qui n'est pas effectuée avec cette solution.
3. L'aspect orienté objet des applications Java Card: Le changement de code d'une classe peut avoir un impact sur les autres classes, ce qui n'est pas vérifié dans cette première solution.

Remplacer en mémoire persistante un bloc d'instructions par un autre ne suffit plus dès lors que l'application correspondante est basée sur une approche orientée objet et que les applications du système peuvent échanger des données entre elles par l'intermédiaire de services. La mise à jour dynamique va donc bien au delà d'un simple remplacement statique de code. Dans cet article, nous nous intéressons qu'à un sous-ensemble du problème: la mise à jour dynamique d'une classe active d'un module d'application Java Card dont le cycle de vie est associé à celui du module d'application.

4.2. Mise à jour dynamique d'une classe

Remplacer un bloc d'instructions en mémoire persistante par une autre ne suffit plus dès lors qu'il s'agit de mise à jour dynamique. En effet, la classe étant active, il s'agit de prendre en compte les instances de la classe, le contenu de la pile d'exécution et toutes les références à cette classe. Une solution simple serait d'attendre que la classe soit inactive, mais la machine virtuelle Java Card possède un cycle de vie identique à celui de la carte (*i.e.* les objets persistants de la machine virtuelle Java Card sont préservées en dehors des sessions de communication avec l'extérieur, mais aussi lors de l'arrêt de l'alimentation de la carte).

Une classe Java Card contient en général des membres données et des membres méthodes. Elle peut hériter d'une autre classe, implémenter une interface, ou être instanciée par une autre classe. Elle peut donc être en relation avec une ou plusieurs autres classes. Le but d'une mise à jour est de pouvoir modifier un membre de la classe, une signature de méthode, des variables locales d'une méthode,etc... en sachant qu'une ou plusieurs instances sont en cours d'exécution dans la JCVM. Dans notre approche, cette mise à jour se déroule en deux étapes: *off-card* et *on-card*.

Partie *off-card*

Dans un premier temps, une application en dehors de la carte calcule la différence entre l'ancienne version de la classe (présente dans la carte) et la nouvelle version. Cette différence (appelée DIFF) permet de répondre à la question : « Quels sont les éléments qui ont réellement changés entre les deux versions de la classe ? ». Cette DIFF est ensuite envoyée sur la Java Card pour être *wrappée* à l'ancienne classe active afin d'obtenir la nouvelle version. L'application *off-card* de génération de la DIFF est donc capable de ressortir les modifications effectuées sur (1) les attributs propres de classe (*package*, extensions, implémentations); (2) les variables globales de la classe (*i.e.* les membres données de la classe); (3) la signature d'une méthode, ses propriétés d'accès, ses variables locales et son code (*bytecode*).

Cette DIFF est représentée dans un langage de spécification DSL (*Domain Specific Language*) défini à cet effet. Elle est transférée dans la Java Card et interprétée afin d'obtenir la nouvelle version de la classe à partir de l'ancienne. L'utilisation de la DIFF présente plusieurs avantages:

1. Minimiser l'*overhead* en taille, c'est-à-dire réduire le coût réseau de transfert de la mise à jour.
2. Éviter le problème de conflit de nom au sein de la JCVM. Si la nouvelle version de la classe est chargée alors qu'une instance de l'ancienne version est en cours, il est possible d'avoir un conflit de nom entre les deux versions (*name clashes* [11]). La machine virtuelle refuserait alors de charger la nouvelle version de l'application tant que l'ancienne version est encore présente dans la Java Card.

Partie *on-card*

Après avoir obtenu la DIFF, il faut envoyer cette DIFF sur la carte pour effectuer la mise à jour dynamique. La question à résoudre est: « Comment utiliser la DIFF pour obtenir la nouvelle version de la classe en préservant la syntaxe, la sémantique du langage, la cohérence des types entre les classes et surtout veiller à ne pas introduire de nouvelles failles de sécurité ?». Étant donné que notre approche consiste à ajouter un nouvel état au cycle de vie d'un module d'une application, il est nécessaire de modifier le gestionnaire de la Java Card (*Card manager)* en lui ajoutant le module *updateAppCard.* Le rôle de ce nouveau module est d'étendre le cycle de vie d'un module d'application pour ajouter la gestion de la mise à jour dynamique tout en restant cohérent avec l'architecture Java Card.

Ce module comporte :

- *wrapper* capable de prendre en entrée la DIFF, de l'interpréter et d'envoyer des instructions au *patcher.*
- *patcher* qui se charge de modifier l'ancienne version de la classe sélectionnée sur la carte pour qu'elle corresponde à la nouvelle version.
- *security manager* dont le rôle est de s'assurer que de nouvelles failles de sécurité ne seront pas introduites.
- *roll-backer* qui ré-initialise de manière atomique à l'état antérieur à l'application de la DIFF en cas d'échec de l'opération de mise à jour.
- Un détecteur de *safe update point* qui détermine le moment opportun pour appliquer la DIFF à partir des informations délivrées par le *patcher*.

La figure 1 présente les composants du module *updateAppCard* et les relations entre eux. On notera sur ce schéma la présence de deux espaces d'instauration différents pour les objets persistants et les objets transients [5, 6].

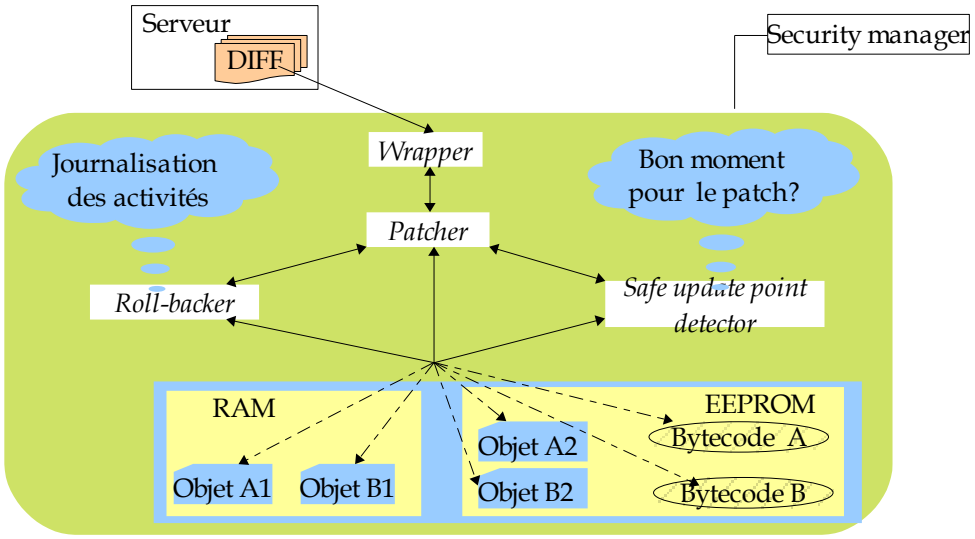

FIG. 1 –Composants du module *updateAppCard*

5. Défis rencontrés lors des DSU

Plusieurs problèmes sont rencontrés durant une DSU dus au fait que la classe peut être instanciée par d'autres classes appartenant à d'autres applications. Dans ce cas, les références à cette classe rendent sa mise à jour compliquée. De plus, il faut s'assurer que la sémantique du langage est préservée et qu'il n'y a donc pas d'incohérence de type.

5.1. Préserver la cohérence du système de type

La modification d'une classe peut avoir un très grand impact sur la cohérence des types. Soit par exemple trois classes simples A, B, et C définies par la figure 2.

```
public Interface I {
        public int crediter (int m);
        public int debiter (int m);
}
public class A implements I {
        private int somme;

        public int créditer (int montant) {
                this.somme = this.somme + montant;
                return somme;
        }

        public int débiter (int montant) {
                if (this.somme – montant > 0) {
                        this.somme = this.somme – montant;
                        return somme;
                } else return -1;
        }
  }
```

```
public class B {
    int amount;
    public void banque (int i, int montant)  {
        I  aux = new A();
        if (i==0)
            amount = A.crediter(montant);
        else   amount = A.debiter (montant);
    }
}
```

```
public class C {
    int prix;
    public void user ( int i)  {
        B aux = new B();
        B.banque (i, prix);
    }
}
```

FIG. 2 – Classes A, B et C

Supposons que dans la nouvelle version de A (figure 3) n'implémente plus l'interface I et qu'elle offre uniquement la méthode *créditer* avec des paramètres de type **Long**.

```
public class A {
        private Long somme;
        public Long créditer (Long  montant, float TVA) {
            this.somme = this.somme + montant - TVA;
            return somme;
        }
  }
```

FIG. 3 – Nouvelle version de la classe A

La modification de la classe A va générer des violations de type. Il peut s'agir :

- d'une référence à un champ ou une méthode qui a été supprimée. Dans notre exemple la méthode *débiter* a été supprimée dans la classe A mais est référencée par la classe B. Pour résoudre ce problème, notre approche consiste à mettre en place un diagramme de flot de donnée associé aux différentes classes [8] pour déterminer celles qui utilisent la méthode et/ou la variable supprimée afin de les mettre également à jour;

- Ou d'une référence à une variable dont le type a changé, à une méthode dont la signature a été modifiée, etc... Dans cet exemple, l'instanciation de la classe A par la classe B pour obtenir une Interface I a été supprimée car la méthode *créditer* de la classe A a

changé de signature et de type de retour. Par conséquent les classes B et C deviennent incohérentes et inutilisables.

Préserver la cohérence des types revient donc à corriger ces violations des types lors de la modification des classes. Pour cela il faut définir deux propriétés très importantes : le *safe update point* et l'atomicité de la transaction de l'opération de mise à jour.

Atomicité de la mise à jour
Elle est assurée par le *roll-backer* du module *updateAppCard*. il doit maintenir un journal des événements de mises à jour effectuées pour une classe donnée, ceci en tenant compte des contraintes d'espace mémoire liées à la carte à puce en général.

Safe update point
Un point crucial de la mise à jour dynamique est le moment auquel il faut effectuer le *patch* pour éviter des modifications de code exécuté au même instant par d'autres instances. Supposons dans notre exemple qu'une instance de la classe B et C soient actives dans la JCVM et que la classe A soit mise à jour avant que la classe C appelle la méthode *banque* de B. Une défaillance du système peut alors apparaître pour plusieurs raisons: (1) la méthode *débiter* a été supprimée; (2) la signature et le type de retour de la méthode *créditer* ont été modifiés; (3) l'interface I a été supprimée; (4) le résultat récupéré par la variable *amount* dans B est de type **Long**. Pour éviter toutes ces incohérences, il faut pouvoir détecter le bon moment durant lequel la mise à jour peut être effectuée. Notre postulat pour résoudre ce problème est que pour supprimer/modifier une méthode, une interface, ou une variable, il faut vérifier qu'il n'existe aucune classe active dans la machine virtuelle qui référence celle-ci. Cette tâche est effectuée par le détecteur du *safe update point* du module *updateAppCard*.

5.2. Assurer la sécurité des applications de la carte

Le module *updateAppCard intégré* dans le C*ard manager* de la Java Card doit pouvoir implémenter un modèle de sécurité qui assure que la mise à jour s'applique uniquement aux classes auxquelles l'autorisation est fournie (sous forme de certificat associé à la DIFF par exemple). Notre approche consiste à mettre en place un *security manager* dans ce nouveau module qui gère les accès aux classes à mettre à jour grâce à la signature associée à la DIFF.

6. Approche de mise à jour en *on Card*

6.1. Modification de la définition de la classe

C'est le rôle du *patcher,* il s'occupe de la ré-localisation du code suivi de l'édition des liens. Donc, Il s'occupe de recopier l'ancienne définition de la classe tout en la modifiant grâce aux instructions fournies par le *wrapper* vers une autre adresse en mémoire persistante, de recopier tout en modifiant les objets de la classe puis d'effectuer les modifications de certaines références nécessaires pour qu'elles pointent vers la nouvelle définition de la classe et/ou vers les nouveaux objets de la classe. De plus, le *patcher* veille à libérer les espaces associés à l'ancienne définition et aux anciens objets de la classe.

6.2. Modification des instances de la classe

La modification des instances de la classe s'apparente au problème du *garbage collection* [2, 7] de la JCVM. Notre approche consiste à rechercher dans le tas de la machine virtuelle toutes les instances de la classe à modifier, à mémoriser leurs références dans un tableau d'indicateur des objets marqués. Puis à parcourir ce dernier pour mettre à jour les objets associés.

6.3. Modification des classes dépendantes

Il s'agit tout d'abord d'identifier non seulement toutes les classes qui dépendent de la classe à mettre à jour mais aussi du type de dépendance (héritage, utilisation de méthode, utilisation d'une variable, etc). Ainsi, il est possible de procéder au changement des références nécessaires.

6.4. Coût de la modification d'une classe

Étant donné que l'implémentation n'est pas totalement effectuée, on ne peut réellement se prononcer sur le coût d'une modification. Toutefois, le coût dépendra du type de modification apportée par la nouvelle version et du nombre de dépendances à la classe à mettre à jour.

7. Conclusion

Le problème d'une mise à jour dynamique (sans arrêter les autres applications et services de la carte) est un problème relativement complexe. La recherche d'un *safe update point* par exemple nécessite de disposer de mécanismes capable d'effectuer une introspection de l'organisation des objets dans la carte. Les limitations sévères de ce type d'environnement (bande passante, mémoire, CPU) nous obligent à trouver des stratégies pour limiter l'information transmise (utilisation d'une DIFF) et d'un mécanisme léger pour les transformations *on-card* (réécriture du code lors de la copie des classes). Enfin, la plateforme Java Card 3.0 est proche de la technologie Java standard (API, utilisation du format jar pour transmettre les applications à la carte, présence d'un *byte code verifier* dans la carte). Il est donc nécessaire d'intégrer nos modifications sans remettre en question le cycle de vie d'une application ou les mécanismes de sûreté.

Actuellement, le premier objectif est d'implémenter complètement et de tester la partie o*n Card* sur une Java Card afin d'analyser les performances du processus et de pouvoir étudier concrètement les nouvelles failles de sécurité possible.

8. Bibliographie

1. Jonathan T. Moore, Michael Hicks, et Scott Nettles. *Dynamic software Updating.* In Programming Language Design and Implementation. ACM, 2001.
2. Robert Henriksson. *Scheduling garbage collection in embedded systems.* Thèse de doctorat, Département d'informatique, Université de Lund, Juillet 1998.
3. Jean-Louis Lanet. *introduction à la carte à puce et problématique de la sécurité.* Crypto'puces, Porquerolles, Avril 2007.
4. Jean-Louis Lanet. *Sécurité des systèmes ouverts pour cartes à puces.* Workshop ISYPAR, 2000.
5. Z. Chen. *Java Card Technology for Smart Cards*. Addison, Wesley, 2000.
6. Sun Microsystem. *The Java Card 3.0 specification.* http://java.sun.com/javacard/, March 2008.
7. Paul R. Wilson. *Uniprocessor Garbage Collection Techniques.* International workshop on memory management, September 1992.
8. Marc Roper, Murray Wood, and Neil Walkinshaw. *The Java System Dependance Graph.* International Workshop on Source Code Analysis and Manipulation, 2003.
9. Milan Fort. *Smart card application development using Java Card Technology*. SeWeS 2006.
10. Kathryn S. McKinley, Michael Hicks, et Suriya Subramanian. *Dynamic Software Updates : A VM-Centric Approach.* In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* , June 2009.
11. Petr Hnětynka, Petr Tůma. *Fighting Class Name Clashes in Java Component Systems.* JMLC : joint modular languages conference, Klagenfurt, Autriche, 2003.
12. Jesper Andersson et Tobias Ritzau. *Dynamic deployment of Java applications.* Java for Embedded Systems Workshop, London, May 2000.
13. Earl Barr, J. Fritz Barnes, Jeff Gragg, Raju Pandey et Scott Malabarba. *Runtime support for type-safe dynamic java classes.* Ecoop, 2000.
14. Alessandro Orso, Anup Rao, et Mary Jean Harrold. *A technique for dynamic updating of Java Software*. ICSM, 2002.