Validation des spécifications formelles de la mise à jour dynamique des applications Java Card

Razika Lounas

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ M'HAMED BOUGARA – BOUMERDES

جامعة بومرداس
Université de Boumerdes
University of Boumerdes

FACULTÉ DES SCIENCES

En Cotutelle avec l'Université de Limoges, France

Université de Limoges

Ecole Doctorales n°610 Sciences et Ingénierie des Systèmes, Mathématiques, Informatique-SISMI

THÈSE DE DOCTORAT

Présentée par :

LOUNAS Razika

Filière : Informatique

Option : Informatique

Validation des Spécifications Formelles de la Mise à Jour Dynamique des Applications Java Card

---

Soutenue publiquement le 25/11/2018, devant le jury :

M. AHMED NACER	Mohamed	Prof-USTHB	Président
Mme. BOUKALA-IOUALALEN	Malika	Prof-USTHB	Examinateur
M. AIT AMEUR	Yamine	Prof- INPT-ENSEEIHT/IRIT	Examinateur
M. IMACHE	Rabah	MCA-UMBB	Examinateur
M. MEZGHICHE	Mohamed	Prof-UMBB	Directeur de thèse
M. LANET	Jean-Louis	Prof-INRIA Rennes	Directeur de thèse

Année Universitaire : 2017/2018

ملخّص

التّغيرات الدّيناميكية للبرامج عبارة عن تطبيق تعديلات على البرامج دون إيقافها. تعتبر هذه الخاصّية أساسيّة في الأنظمة الحسّاسة التّي هي في تطوير متواصل و التّي تتطلّب أن تكون في الخدمة باستمرار. الهدف من عملنا هو التّأكّد الصّوري لصحّة تطبيق التعديلات بطريقة دّيناميكية على برامج Java Card في إطار النّظام EmbedDSU.

لهذا الغرض، تحقّقنا أوّلا من صحّة التّعديلات على مستوى نصّ البرنامج وذلك بتعريف المعاني الصّوريّة لعمليّات التّعديلات على النّص الوسط Java Card و ذلك من أجل التّأكّد من أمان الأنواع في البرنامج المغيّر. إقترحنا بعدها طريقة للتّأكّد من صحّة معاني البرامج المغيّرة من خلال اقتراح تحويلات على المسندات.

إهتممنا بعد ذلك بالتّحقّق من صحّة إكتشاف النّقط الآمنة للتّعديلات. إستعملنا التّأكّد عبر النّماذج. هذا التّأكّد سمح لنا بتصحيح خطأ بخصوص إنغلاق في النّظام و بالتأكدّ من خاصيّات أخرى و هي أمان الفعاليّة و ضمان التّغيير.

يتمّ التّعديل على المعطيات بواسطة دوال تغيير الظّرف. في هذا الجانب، اقترحنا حلاّ يسمح بتطبيق هذه الدّوال بطريقة تضمن تناسق الذّاكرة المخصّصة للأشياء في الآلة الإفتراضيّة Java Card، كما تسمح بتعبيريّة جيّدة.

الكلمات المفاتيح: التّعديلات الدّيناميكيّة للبرامج، الطّرق الصّوريّة، Java Card ،صحّة التّعديلات الدّيناميكيّة للبرامج.

ا

Résumé

$\mathcal{L}$a mise à jour dynamique des programmes consiste en la modification de ceux-ci sans en arrêter l'exécution. Cette caractéristique est primordiale pour les applications critiques en continuelles évolutions et nécessitant une haute disponibilité. Le but de notre travail est d'effectuer la vérification formelle de la correction de la mise à jour dynamique d'applications Java Card à travers l'étude du système EmbedDSU.

Pour ce faire, nous avons premièrement établi la correction de la mise à jour du code en définissant une sémantique formelle des opérations de mise à jour sur le code intermédiaire Java Card en vue d'établir la sûreté de typage des mises à jour. Nous avons ensuite proposé une approche pour vérifier la sémantique du code mis à jour à travers la définition d'une transformation de prédicats.

Nous nous sommes ensuite intéressés à la vérification de la correction concernant la détection de points sûrs de la mise à jour. Nous avons utilisé la vérification de modèles. Cette vérification nous a permis de corriger d'abord un problème d'inter blocage dans le système avant d'établir d'autres propriétés de correction : la sûreté d'activation et la garantie de mise à jour.

La mise à jour des données est effectuée à travers les fonctions de transfert d'état. Pour cet aspect, nous avons proposé une solution permettant d'appliquer les fonctions de transfert d'état tout en préservant la consistance du tas de la machine virtuelle Java Card et en permettant une forte expressivité dans leurs écritures.

Mots clefs : Mise à jour dynamique des programmes, méthodes formelles, Java Card, correction de la mise à jour dynamique des programmes.

Abstract

$\mathcal{D}$ynamic Software Updating (DSU) consists in updating running programs on the fly without any downtime. This feature is interesting in critical applications that are in continual evolution and that require high availability. The aim of our work is to perform formal verification the correctness of dynamic software updating in Java Card applications by studying the system EmbedDSU.

To do so, we first established the correctness of code update. We achieved this by defining formal semantics for update operations on java Card bytecode in order to ensure type safety. Then, we proposed an approach to verify the semantics of updated programs by defining a predicate transformation.

Afterward, we were interested in the verification of correction concerning the safe update point detection. We used model checking. This verification allowed us first to fix a deadlock situation in the system and then to establish other correctness properties : activeness safety and updatability.

Data update is performed through the application of state transfer functions. For this aspect, we proposed a solution to apply state transfer functions with the preservation of the Java Card virtual machine heap consistency and by allowing a high expressiveness when writing state transfer functions.

Keywords : Dynamic software update, formal methods, Java Card, Dynamic software update correctness.

Remerciements

J'adresse mes remerciements en premier lieu au Ministère Algérien de l'Enseignement Supérieur et de la Recherche Scientifique pour m'avoir permis de finaliser mes travaux de thèse à travers l'octroi d'une bourse pour l'année universitaire 2015/2016 dans le cadre du programme PROFAS B+.

Mes remerciements les plus sincères et ma gratitude la plus profonde s'adressent à mes directeurs de thèse : Monsieur Mohamed MEZGHICHE, professeur à l'Université de Boumerdes et Monsieur Jean-Louis LANET, professeur à l'INRIA de Rennes, France.

Je remercie Monsieur Mohamed MEZGHICHE de m'avoir encadrée et formée depuis mon accès à la post graduation. Je le remercie pour sa patience, sa disponibilité, et ses conseils précieux et avisés. Ce travail n'aurait jamais aboutit sans son précieux soutien et ses encouragement qui m'ont toujours redonné confiance et volonté. Je tiens également à le remercier en sa qualité de directeur du laboratoire LIMOSE pour les efforts qu'il fait pour les enseignants chercheurs et les doctorants.

Je remercie Monsieur Jean-Louis LANET de m'avoir accueillie dans son équipe à Limoges puis à l'INRIA, Rennes, Bretagne Atlantique. Son accueil chaleureux, sa disponibilité, ses précieux conseils et encouragements m'ont toujours aidée à avancer et ont grandement contribué l'aboutissement de ce travail. Je le remercie de m'avoir permis d'avoir une expérience au sein de l'INRIA qui constituera sans doute un acquis pour ma carrière.

Travailler avec les professeurs Mohamed MEZGHICHE et Jean-Louis LANET est une chance inouïe pour ma carrière. Qu'ils trouvent ici le témoignage de ma sincère reconnaissance et de mon profond respect pour leurs grandes qualités scientifiques et humaines.

J'adresse mes remerciements les plus chaleureux aux membres du jury. Je remercie Monsieur Mohamed AHMED NACER, professeur à l'USTHB pour m'avoir fait l'honneur d'accepter de présider mon jury. Je remercie Madame Malika BOUKALA-IOUALALEN, professeur à l'USTHB, Monsieur Yamine AIT AMEUR, professeur à l'INPT-ENSEEIHT/IRIT et Monsieur Rabah IMACHE, Maître de conférences à l'UMBB pour m'avoir fait l'honneur d'examiner mes travaux et d'être membre du jury de ma soutenance.

Que Madame, Messieurs les membres du jury trouvent en ces mots l'expression de mon profond respect en espérant que mon travail aura été à la hauteur de leurs exigences scientifiques.

J'adresse mes remerciements à Monsieur Mohamed CHAABANI, Chef du département d'Informatique pour ses efforts et son aide.

Je remercie Monsieur Yamine AIT AMEUR pour son aide et le temps qu'il m'avait accordé au début de cette thèse pour discuter de la problématique.

Je remercie Monsieur IMACHE Rabah en sa qualité de chef d'équipe au laboratoire LIMOSE pour ses efforts, son aide et ses nombreux conseils aussi bien sur le plan pédagogique que scientifique.

Je remercie Rabéa Aimeur-Boulifa pour son aide, notamment d'avoir relu mon papier et ses précieuses remarques et suggestions.

Je remercie Hélène Le Bouder pour sa gentillesse et pour son aide concernant le modèle Latex de ce manuscrit.

Je remercie Axel Legay et Nisrine Jafri, respectivement chef de l'équipe TAMIS, INRIA et doctorante à l'INRIA pour avoir travaillé avec moi sur la deuxième partie des contributions de ces

travaux de thèse.

Durant ces années de thèse et mon travail au sein du département d'Informatique à l'UMBB et au laboratoire LIMOSE, j'ai eu le plaisir de côtoyer des collègues enseignants, doctorant et personnel administratif que je remercie pour leur gentillesse. Je remercie particulièrement mes collègues et amis : Saadia Kedjar, Samiya Hamadouche et Hocine Mokrani pour leurs encouragements et leur précieuse aide en relisant certains chapitres de ce document de thèse. Je remercie Zahira Chouiref et Selma Djeddai et pour leurs aides et encouragements.

Je remercie mes collègues : Hamadouche Samiya, Mokrani Hocine, Mesbah Abdelhak, Salhi Dhia Eddine et Djerbi Rachid pour leur précieuse aide concernant la préparation de la soutenance.

Je remercie le personnel au niveau de l'Ecole Doctorale, du Département d'Informatique et du laboratoire Xlim de l'Université de Limoges, pour leur aide et leur disponibilité le long de mon inscription en cotutelle. Plus particulièrement : Mathilde Lecompte, Annie Nicolas, Sabrina Brugier, Claire Buisson. Je remercie également Cécile Bouton et Sophie Viaud de l'INRIA de Rennes pour leurs gentillesse et disponibilité.

J'exprime mes sincères remerciements à mes sœurs et mes frères pour leur soutien et leurs encouragements.

Aucune page de remerciements ne suffira pour exprimer ma gratitude infinie et ma profonde reconnaissance aux deux personnes qui me sont le plus chers au monde : merci à mes parents à qui je dois tous et qui sont un trésor dans ma vie.

Table des matières

Table des figures

Liste des tableaux

Chapitre I

Introduction

"We shall not cease from exploration, and the end of all exploring will be to arrive where we started and to know the place for the first time" T. S. Eliot.

Les logiciels en évolution est un fait de la vie. Les développeurs sont amenés à mettre à jour les systèmes en vue d'en corriger les erreurs, ajouter des fonctionnalités ou effectuer des optimisations. Le processus classique pour effectuer les mises à jour consiste à arrêter le système, effectuer les modifications puis redémarrer l'application. Cependant, dans certaines applications, ce processus qui implique un temps d'arrêt du système engendre un coût qui peut s'avérer prohibitif. En effet, certains domaines d'applications sont dits critiques, il s'agit par exemple d'applications concernant les transactions bancaires, les logiciels de contrôle médical ou les contrôleurs de circulation aérienne. Des faits historiques ont établi que dans ce type d'applications, un arrêt et une perte d'état du système peut avoir des conséquences économiques ou humaines sévères. L'alternative souhaitée à ce type de mise à jour serait une technique qui permettrait d'effectuer les modifications nécessaires tout en gardant le système en exécution : il s'agit de la mise à jour dynamique des programmes (DSU : Dynamic Software Updating).

I.1 Contexte

La DSU désigne l'ensemble des techniques pour mettre à jour un système logiciel ou une application durant son exécution sans interruption de ce dernier et sans perte de l'état d'exécution du système. La DSU permet de mettre à jour le système pour corriger des erreurs, faire face à de nouvelles exigences en ajoutant de nouveaux services à une application, supprimer des services devenus inutiles ou bien en vue de l'optimiser. C'était il y'a quarante ans, Fabry s'exprimait sur le concept de DSU en ces termes [1] :

"Dans un système composé de plusieurs modules, il est souvent nécessaire de mettre à jour l'un d'eux afin de fournir de nouvelles caractéristiques ou une amélioration dans l'organisation interne ... Si les modules gèrent des structures de données permanentes qui doivent être modifiées et que le système est supposé pouvoir continuer les opérations lors de la mise à jour, le problème est plus difficile mais peut être résolu."

Ce fut dans l'introduction au premier système de mise à jour dynamique. Depuis, le concept de la DSU a connu une large expansion à travers le développement de plusieurs techniques et approches pour différents domaines d'application tels que les systèmes d'exploitation, les logiciels de transports ou les systèmes embarquées. Fournir un mécanisme de DSU est une tâche qui pose plusieurs défis aux chercheurs et développeurs. Ces défis sont relatifs à différents aspects et représentent les principaux problèmes scientifiques de la DSU. Le choix du moment opportun pour appliquer la mise à jour représente l'un de ces défis. En effet, plusieurs techniques sont proposées pour spécifier les moments

acceptables pour la DSU et les méthodes permettant de les calculer ou de ramener les systèmes vers ces points. Les chercheurs proposent ensuite des techniques permettant la mise à jour du code et des données. Ces techniques permettent de définir comment implémenter et accéder aux nouvelles versions du code (classes, fonctions ...) et d'effectuer les transferts d'états du programme de l'ancienne vers la nouvelle version.

L'application de la mise à jour dynamique est un processus composé de plusieurs étapes. A chacune d'elles, des erreurs sont susceptibles de se produire. Des erreurs peuvent survenir par exemple lors du choix du moment opportun, de la spécification des parties à mettre à jour ou bien lors de l'écriture des fonctions permettant de transformer les états des programmes. Une erreur au niveau de ces étapes peut se traduire par un arrêt brutal du système, des comportements erronés ou même des brèches dans sa sécurité. La correction de la DSU est un des défis majeur et constitue une problématique orthogonale aux autres. Cette problématique conduit au développement de techniques permettant de garantir la correction à différents niveaux : code, données et choix des points de mise à jour.

I.2 Motivations et objectifs

Les systèmes de DSU s'immiscent de plus en plus dans des applications critiques. Dans ce genre d'applications, une erreur logicielle peut avoir des conséquences dramatiques (pertes de vies humaines ou catastrophes économiques). Par conséquence, en parallèle à l'expansion de l'utilisation de la DSU, se pose de manière de plus en plus accrue la problématique de sa correction. Les développeurs des systèmes de DSU doivent s'assurer que les apports de la DSU en termes d'évolution et de haute disponibilité ne soient pas contre balancés par l'introduction d'erreurs lors de l'application de celle-ci.

Notre travail s'inscrit dans la correction formelle de la DSU. En effet, la réponse aux exigences en matière de garantie de correction des programmes a été apportée par les méthodes formelles. Ces méthodes proposent généralement d'abord un cadre formel pour spécifier les systèmes de DSU et les applications à mettre à jour. Ensuite, une démarche est proposée pour établir par un raisonnement mathématique la conformité du système de DSU aux besoins de l'utilisateur, nous parlons alors de correction de la mise à jour dynamique. Dans ce cadre, des travaux de recherches pionniers ont établit l'indécidabilité du problème de la correction de la DSU. Ceci signifie qu'on ne peut pas développer un algorithme qui permet à partir d'un programme en exécution et d'une mise à jour, de développer un algorithme permettant de répondre si une mise à jour est correcte. De ce fait, les travaux de recherche se basent sur la définition d'un ensemble de critères liés aux différents aspects de la DSU permettant d'établir la correction.

Nous nous intéressons à la correction formelle de la DSU pour les applications des cartes à puces Java (Java Card). A l'instar des différentes applications critiques, la carte à puce peut aussi requérir le besoin de la DSU en vue de corriger des défauts dans une application préalablement chargée, ou peut nécessiter d'adapter le système à un nouvel environnement. Des erreurs de protocoles cryptographiques peuvent être décelées voire même des erreurs de conception. La DSU permet donc de corriger ces bogues sur ces applications après leurs déploiements sur la carte.

L'objectif de cette thèse est de proposer des approches pour la vérification formelle de la correction de la mise à jour dynamique pour les applications Java Card avec une étude de cas du système EmbedDSU. Ces approches permettent d'énoncer et de vérifier des propriétés de correction de la mise à jour vis-à-vis des différents aspects de la DSU :

– Au niveau de la mise à jour du code : l'objectif est de proposer des approches formelles permettant de garantir que le programme mis-à-jour est bien typé. Nous nous intéressons également à garantir que ce programme réponde aux spécifications définies par l'utilisateur.

- Au niveau de la recherche des points sûrs : nous proposons une vérification formelle pour garantir la correction du module de la recherche de points sûrs du système EmbedDSU.
- Au niveau de la mise à jour des données : nous proposons une méthode et des algorithmes pour effectuer la mise à jour des états des programmes pour obtenir la nouvelle version de l'application et le nouveau contexte du système à partir de l'ancienne version. Les algorithmes proposés permettent de prendre en considération les fonctions de transfert d'état définies par le programmeur ainsi que les différents liens entre les objets crées par les applications. La solution proposée permet de garantir la consistence des états des applications.

L'association de ces trois objectifs permet de garantir la correction du système.

I.3 Organisation

Ce document est composé des parties suivantes :

- **La première partie** représente un état de l'art sur la mise à jour dynamique est l'application des méthodes formelles à la correction de celle-ci. Cette partie est composée de deux chapitres. Le chapitre I : ***La mise à jour dynamique des programmes*** est consacrée à la présentation de la mise à jour dynamique des programme à travers sa définition, ses caractéristiques, les problèmes scientifiques soulevés ainsi que la présentation de quelques systèmes de DSU. Ce chapitre se termine par l'évocation de l'intérêt de la DSU et notamment de l'utilisation des méthodes formelles, introduisant ainsi le chapitre suivant. Le chapitre II dont le titre est : ***Correction formelle de la DSU***, est consacré à la présentation d'un état de l'art sur la correction formelle de la DSU. Nous y détaillons d'abords les différents critères de corrections, puis nous présentons les différentes approches formelles utilisées pour établir les critères de correction.

- **La deuxième partie** est consacrée à la présentation du contexte d'étude : il s'agit d'un système de mise à jour dynamique pour les applications des cartes à puce Java. Cette partie est composée de deux chapitres : le chapitre III : ***Les cartes à puce et la technologie Java Card*** est consacré à la présentation de la plateforme Java Card à travers la présentation de différentes notions relatives aux cartes à puces, caractéristiques de la plateforme Java Card ainsi qu'une introduction au langage intermédiaire de Java, le bytecode. Le chapitre IV : ***Le système de mise à jour dynamique EmbedDSU*** est consacré à la présentation du système sur lequel porte notre étude : le système EmbedDSU. Nous y détaillons l'architecture du système, ses différents modules et le principe de fonctionnement. Nous motivons aussi l'intérêt de l'étude d'EmbedDSU.

- **La troisième partie** est consacrée à nos contributions. Elle est composée de trois chapitres. Le chapitre VI : ***Correction de la mise à jour du code***, présente d'abord les critères de corrections que nous définissons pour la correction de la mise à jour du code : la sûreté de typage et la correction comportementale. Nous présentons ensuite notre formalisation de la sémantique des opérations de mise à jour et l'énoncé du critère de correction de la sûreté de typage. Nous présentons ensuite notre proposition du calcul de transformation de prédicats pour la correction des spécifications comportementales des programmes mis à jour. Le chapitre VII : ***Correction de la recherche des points sûrs*** est consacré à notre contribution sur la correction du module de la détection des moments opportuns pour effectuer la mise à jour. Nous présentons une démarche de vérification de modèles qui nous a permis d'établir la correction à travers trois critères de correction. Le chapitre VIII : ***La mise à jour des données*** est consacré à la présentation d'une approche permettant d'effectuer la mise à jour des données de façon à garantir la consistance des états des applications mises à jour.

Ce document termine par une conclusion générale et la présentation de quelques perspectives.

Première partie

Mise à jour dynamique et méthodes formelles

Chapitre II

La mise à jour dynamique des programmes

Sommaire

*M*ettre à jour des programmes dans le but d'en corriger des erreurs, ajouter des fonctionnalités ou améliorer les performances est indispensable. Cependant, c'est une activité qui peut perturber le fonctionnement d'un système par un arrêt et une perte des données. La mise à jour dynamique, qui consiste en la possibilité de mettre à jour des programmes sans en arrêter l'exécution, est de ce fait d'une importance majeure pour des applications où l'interruption des services et la perte d'état représentent des pertes significatives. Dans ce chapitre, nous présentons la mise à jour dynamique en définissant le concept, son intérêt ainsi que les différents problèmes scientifiques soulevés. Ces problèmes ont mené au développement de plusieurs techniques. Nous présenterons des systèmes issus de différents domaines et appartenant à différents styles d'applications.

II.1 Introduction à la DSU

II.1.1 Définition

Mettre à jour un programme consiste à déployer une nouvelle version de celui-ci de manière à remplacer l'ancienne version qui est déployée sur un ensemble d'infrastructures. La mise à jour dynamique des programmes (DSU pour Dynamic Software Updating) [2], appelée aussi mise à jour en ligne des programmes (on-line software update) [3], ou encore mise à jour à chaud des programmes (HotSwUp : Hot Software Update) [4], est définie comme la possibilité de pouvoir modifier un système logiciel ou une application durant son exécution sans interruption de ce dernier. La DSU s'effectue sans perte de l'état d'exécution du système contrairement à la mise à jour statique ou traditionnelle qui implique un arrêt et un redémarrage du système.

D'autres concepts sont proches de celui de la mise à jour dynamique notamment l'adaptation dynamique et la reconfiguration dynamique [5] :

- L'adaptation dynamique [6, 7] représente un changement du système en réponse à une modification du contexte dans lequel il se trouve. Une adaptation dynamique peut par exemple se 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 principale différence entre ce concept et celui de la mise à jour dynamique est que dans l'adaptation, le système détecte et gère par lui même les changements requis par son contexte.
- La reconfiguration dynamique [8, 9], peut se définir comme l'ensemble des changements apportés à une application en cours d'exécution. Ces changements pouvant être la modification de l'architecture de l'application (ajout, retrait des modules et de leurs liaisons) ; la modification de la distribution géographique de l'application ; la modification de la mise en oeuvre d'un composant ; la modification des interfaces de services, etc. Ce concept représente un cas particulier de la mise à jour dynamique des programmes.

Dans la suite du document, nous utiliserons l'abréviation anglophone de la mise à jour dynamique des programmes (DSU).

II.1.2 Intérêts de la DSU

L'intérêt de la mise à jour dynamique se decline selon les deux aspects suivants :

- ***Disponibilité :*** Certaines applications peuvent être mises à jour par le chargement de la nouvelle version et par le redémarrage des applications afin de refléter la mise à jour. Cependant, certaines applications critiques qui requièrent une haute disponibilité, tels que la gestion des transactions financières, le contrôle du trafic aérien, l'infrastructure réseau ou les applications médicales, un arrêt planifié et organisé pour la mise à jour peut entraîner des pertes importantes. La mise à jour dynamique est une approche qui évite cette interruption temporaire en permettant aux systèmes d'être opérationnels sans interruption.
- ***Evolution :*** L'évolution des systèmes est inévitable. En effet, les logiciels sont amenés à être mis à jour plusieurs fois durant leurs cycles de vie. La nature des changements varie entre l'introduction des corrections, l'amélioration des performances comme les optimisations, les protections comme les correctifs de sécurité, l'adaptation des programmes à de nouveaux besoins des utilisateurs via le déploiement de nouveaux services ou l'adaptation aux nouvelles contraintes de l'environnement. La mise à jour dynamique est considérée comme une méthode pour l'évolution des systèmes.

II.1.3 Critères pour la DSU

Un système pratique pour la mise à jour dynamique devrait satisfaire quatre critères : la flexibilité, la robustesse, l'efficacité et la facilité d'utilisation [2, 10].

1. ***Flexibilité*** : Ce critère se réfère à la capacité d'un système à prendre en charge les mises à jour des différents composants systèmes. Idéalement, un système de mise à jour devrait être suffisamment souple pour que n'importe quelle partie d'un programme au cours d'exécution puisse être mise à jour sans nécessiter un temps d'arrêt.
2. ***Robustesse*** : Les applications doivent constamment fournir un service correct et continu. En particulier, plus la probabilité que le système puisse se bloquer, perdre des données, effectuer des opérations incorrectes, ou échouer autrement à cause d'une mise à jour, plus le risque augmente sur l'application qui l'utilise. Quatre propriétés importantes de robustesse que les systèmes de mise à jour devraient chercher à atteindre sont répertoriées :
   (a) *Sécurité :* Une mise à jour ne doit pas effectuer des opérations illégales qui conduisent à une brèche de sécurité, telle que le déréférencement d'un pointeur NULL ou l'indexation d'un tableau en dehors de ses limites.
   (b) *Exhaustivité* : Une mise à jour dynamique doit être complète, ce qui signifie que la mise à jour aborde les changements qu'une nouvelle version a fait à l'ancienne.
   (c) *Temps* : Le choix de temps d'exécution de la mise à jour dynamique ne doit pas entraîner des erreurs.
   (d) *Restauration activé* : Au cours de la mise à jour, des erreurs peuvent se glisser en passant les tests et les procédures de vérification. Par conséquent, un moyen est requit pour restaurer le système à sa forme originale après avoir découvert qu'une mise à jour appliquée a bugué.
3. ***Efficacité*** : Les systèmes nécessitants un service sans arrêt sont des systèmes à haute performance, par exemple les serveurs web, les processeurs de transactions, etc. Par conséquent, un système de mise à jour dynamique doit garantir qu'il n'y a pas de dégradation dans la performance due au processus de la mise à jour.
4. ***Facilité d'utilisation*** : L'applicabilité d'un système est souvent déterminée par sa facilité d'utilisation. beaucoup d'outils et produits ont été ignorés parce qu'ils ne sont pas faciles à utiliser. Un système est facile à utiliser s'il réduit la charge de travail sur le programmateur, et / ou réduit la complexité des tâches à effectuer. Une façon de rendre un système de mise à jour facile à utiliser est de séparer le processus de développement de la mise à jour du développement du logiciel. Par exemple, après qu' une nouvelle version du logiciel est développée, un correctif est développé pour mettre à jour dynamiquement le système de fonctionnement de la nouvelle version. De cette façon, les développeurs construisent et testent leurs logiciels sans avoir besoin de penser à des mises à jour, rendant effectivement la construction de logiciels et la construction de correctifs deux composants séparés du processus de développement.

II.2 Problèmes scientifiques de la DSU

II.2.1 La mise à jour du code

La mise à jour du code est la fonctionnalité de base que chaque système de mise à jour dynamique doit implémenter. Cet aspect a pour rôle de soulever les questions relatives à la mise à jour des méthodes, des corps des méthodes, signatures, les champs d'une classe , suppression et ajouts des méthodes en vue de permettre au système de faire exécuter la nouvelle version de celui ci.

Le problème de mise à jour du code réside principalement dans la modification du code binaire et 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.

II.2.2 La mise à jour des données

L'exécution d'un programme est basée généralement sur la notion de transformations d'états. Les états d'un programmes sont définis par un ensemble de données pouvant être des variables globales, des objets dans le tas, des piles d'exécution ou des tables d'une base de données. Une mise à jour dynamique d'un programme peut modifier la structure des données et la façon dont le programme les manipule. La mise à jour dynamique du code entraîne une situation où l'on a un nouveau code et une ancienne version des données. Ceci mène le système à un dysfonctionnement et des comportements erronés. Pour éviter ces situations, les systèmes de mise à jour dynamique doivent proposer des solutions pour mettre à jour les données de telle sorte à les rendre compatibles avec la nouvelle version de l'application. Il s'agit de techniques pour résoudre des problèmes liés notamment à la coexistence des données de l'ancienne et de la nouvelle version après la mise à jour et la méthode de transfert des anciennes instances vers les nouvelles et la préservation de la cohérence du système.

II.2.3 Détection du moment opportun

Les deux problématiques précédentes sont reliées à la façon dont le système est mis à jour. Cependant, appliquer la mise à jour à un moment hasardeux peut laisser le système dans un état incohérent voir même peut provoquer une défaillance du système en dépit de la définition de techniques pour la mise à jour du code et des données. Ceci peut arriver par exemple, si après la mise à jour, une méthode en cours d'exécution accède à un champ qui a été supprimé ou dont le type a été modifié ou fait appel à une méthode dont la signature a été modifiée. Il est nécessaire donc de déterminer à quels moments la mise à jour (du code et des données) peut être réalisée en conservant la cohérence du système. Ces moments sont appelés des points sûrs du système. Cette problématique est reliée à la définition de techniques pour déterminer les points sûrs et ramener le système à atteindre ceux-ci.

II.2.4 Correction de la mise à jour dynamique

La problématique de la correction de la mise à jour dynamique est orthogonale aux autres problématiques. En effet, l'étude des trois premières problématiques mène au développement de techniques pour la mise à jour du code et des données et le choix du moment opportun pour l'application de la mise à jour. Cependant, le défi se situe dans le fait d'appliquer les différentes techniques en garantissant que le système ne défaille pas au moment de la mise à jour, ne corrompt les données et n'introduit pas d'erreurs sémantiques lors de la mise à jour et après celle-ci. Il s'agit donc de garantir la correction de la mise à jour dynamique. La correction de la mise à jour dynamique se présente comme la capacité à vérifier de façon exhaustive ou à tester que le système de mise à jour satisfait des spécifications d'un ensemble de comportements attendus après la mise à jour. Ces spécifications définissent des critères de correction. La problématique de la correction de la mise à jour dynamique est alors scindée en deux sous problématiques : définir les critères de correction et définir des techniques pour les établir.

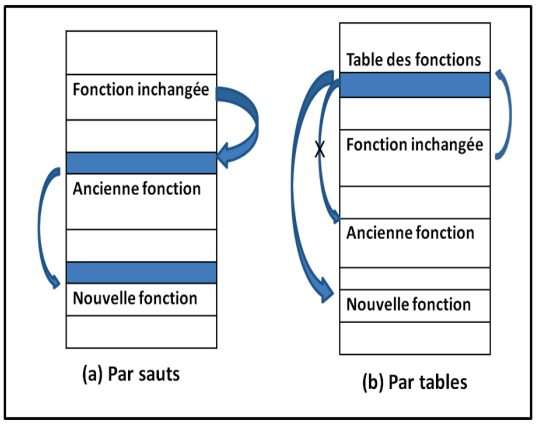

Figure II.1 : Approches d'indirection

II.3 Techniques pour la DSU

II.3.1 Techniques pour la mise à jour du code

II.3.1.1 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 des versions 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. Cette technique a été adoptée par plusieurs systèmes [11, 12, 13, 14]. Cette technique nécessite la définition de conditions pour la consistence du système. Dans [14], l'auteur présente une étude de compatibilité sur des programmes orientés objet pour classifier des objets selon qu'il soient compatibles avec l'ancienne ou la nouvelle version d'un code et définit plusieurs modes de mise à jour possibles. Dans [13], les auteurs proposent un système de renommage des classes pour éviter les cohésions entre nouvelles et anciennes versions ainsi qu'une interface pour l'invocation des méthodes et objets pour garder la cohérence du système.

II.3.1.2 Techniques 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. 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. Cette techniques est utilisée dans des système tels que Ginseng [15] et K42 [16]. Une autre solution pour effectuer l'indirection consiste à remplacer les premières instructions de code d'une méthode modifiée par un appel vers la nouvelle version de la méthode [17]. 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. La Figure II.1 [10] illustre les techniques d'indirection par tables et sauts.

Une autre forme d'indirection est l'utilisation des classes proxy pour les langages orientés objet [18, 19]. La Figure II.2 illustre l'utilisation de cette technique. Chaque classe à mettre à jour est

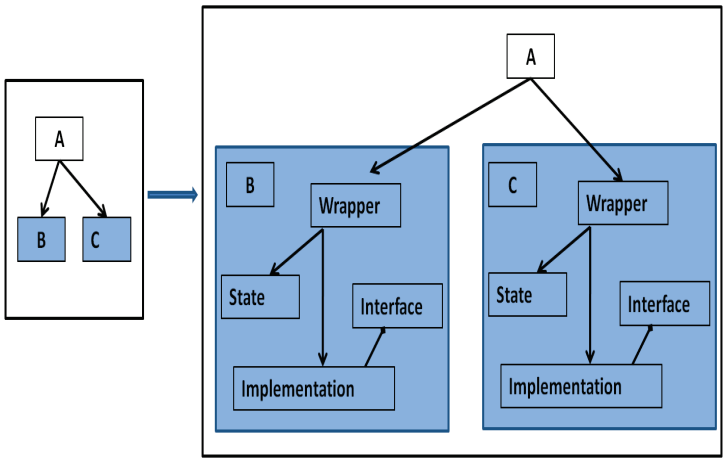

Figure II.2 : Exemple d'utilisation des classes proxy

remplacée par un ensemble de classes. Ceci est effectué notamment dans le but de séparer l'interface des différentes versions de l'implémentation. Toute utilisation de la classe modifiée passe par une classe (wrapper) qui permet d'orienter les appels vers la bonne implementation.

II.3.1.3 Instrumentation binaire

L'instrumentation est l'usage ou l'application d'instruments pour des fins d'observation, de mesure ou de contrôle. Instrumenter un code revient à le modifier par ajout ou suppression de fragments de code pour des raisons d'observation de comportement du programme ou de mesure de propriétés pendant l'exécution, ou de prise de contrôle de l'exécution [20]. L'instrumentation dynamique du binaire ou du bytecode (pour des programmes Java) est le type d'instrumentation exploité pour la mise à jour dynamique. Il s'agit d'un processus d'ajout de nouvelles propriétés à un programme en modifiant le binaire ou le bytecode d'un ensemble de classes. Le code d'instrumentation injecté habituellement lors de l'exécution du code à des fins d'analyse est exploité à des fin de modification du comportement du code. Dans [5], le bytecode est manipulé par une opération de recopie en modification. Le corps d'une méthode est recopié vers un autre emplacement physique. Au fur et à mesure, les nouvelles instructions sont insérées et les instructions supprimées sont ignorées. Le système Ksplice [17] permet d'utiliser la technique du saut en remplaçant, dans le binaire, les premières instructions d'une méthode par un saut.

II.3.1.4 Chargement et édition de liens dynamiques

De très nombreuses applications sont construites sur la base de plusieurs modules. L'édition des liens est le processus qui permet de combiner plusieurs modules sous la forme de fichiers objets en un seul fichier exécutable. Le processus du chargement des programmes (ou parties de programmes) permet de mettre ceux-ci en mémoire pour être exécutés.

A l'origine, ces processus sont statiques : ils sont effectués avant l'exécution des programmes. Cependant, plusieurs applications requièrent chargement et edition des liens dynamiques. De nombreux systèmes de DSU ont exploité ces mécanismes. Dans [2], l'auteur développe un système d'édition de liens dynamique dédié pour la DSU dans des systèmes VNC (Verifiable Native Code). Dans [21, 22], les auteurs présentent des extensions au chargeur Java pour effectuer la DSU. La machine virtuelle Java permet à l'utilisateur d'écrire son propre *classloader* qui hérite du *classloader* Java de telle sorte à supporter la mise à jour dynamique et gérer les différentes versions des classes.

II.3.2 Techniques pour la mise à jour des données

II.3.2.1 Coexistence des données

De manière analogue à la coexistence des versions de codes, certains systèmes de DSU adoptent la technique de coexistence des données, comme par exemple dans [23, 4, 24, 25]. Les anciennes versions des données 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. Ceci nécessite la prise en compte de conditions qui garantissent la cohérence du système. Dans [25, 4], les auteurs définissent des transformateurs de données à double sens : les données peuvent être transformées de l'ancienne vers la nouvelle version et vice versa selon les différentes situations pour maintenir la cohérence. Dans [23], les auteurs proposent dans le cas de la présence de plusieurs versions, un ordonnancement dans le transfert vers la nouvelle version des données.

II.3.2.2 Les fonction de transfert d'états

La modification des données représente la modification de l'état du système de telle sorte à l'adapter à la nouvelle version. L'ajustement apporté à l'état du système par l'application de la mise à jour est généralement spécifié dans les Fonctions de Transfert d'États (FTE ou bien STF : State Transfer Functions). Il existe deux types de fonctions de transfert d'états :

- les fonctions de transfert fournies par le programmeur : elles sont basées sur la sémantique de l'ancienne version de l'application. Elles 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.
- les fonctions de transfert générées automatiquement : certains systèmes de DSU fournissent la fonctionnalité de génération de fonctions de transfert lors de la mise à jour. Généralement, ces fonctions de transfert générées automatiquement sont dans la plupart des systèmes 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, par défaut, les entiers sont initialisés à zéro et les références à NULL. Ainsi, si les membres donnés 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 auteurs dans [26] se démarquent en proposant une approche permettant la génération de FTEs expressives automatiquement en se basant sur les sémantiques de l'ancienne et de la nouvelle version de l'application en utilisant la technique de correspondance entre les objets crées par l'ancienne version et les objets crées par la nouvelle application.

Deux modes sont définis pour l'application des FTE : le mode paresseux et le mode immédiat.

II.3.2.3 Les modes de transfert d'états

La technique de transfert d'état en mode paresseux s'applique après la mise à jour de telle sorte que pour chaque accès à un objet le système vérifie si celui-ci appartient à une classe modifiée, ou à une application modifiée. 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 de cette technique 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

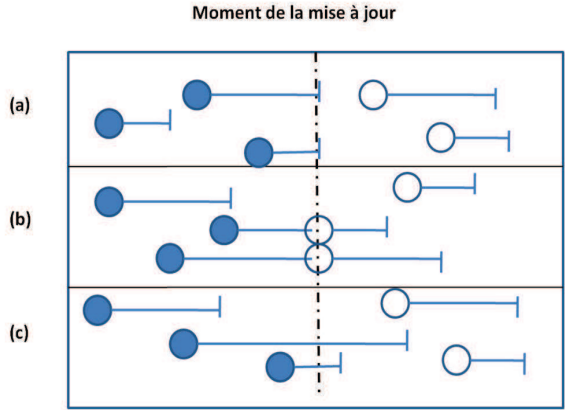

Figure II.3 : Modes pour la mise à jour des données

déterminer le numéro de version de ce dernier. Le mode transfert immédiat consiste à 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.

La Figure II.3 [21] illustre les différentes situations selon les modes de transfert : dans le cas *(a)*, le système attend que toutes les anciennes instances des données finissent, et à partir de là, toutes les instances créées appartiennent à la nouvelle version de l'application. Le cas *(b)* illustre la solution qui propose de transférer immédiatement les anciennes instances pour les rendre compatibles avec la nouvelle version. Le cas *(c)* illustre la situation ou les anciennes instances continuent d'exister tandis que la nouvelle version crée des nouvelles instances de données.

II.3.3 La détection des points de mise à jour

La détection des moments opportuns, appelés aussi points sûrs, pour la mise à jour repose sur deux approches majeures : une approche predictive et une méthode introspective comme le montre la Figure II.4 :

- ***L'approche par prédiction :*** dans cette approche, les points sûrs sont calculés préalablement à la mise à jour. Les travaux adoptant cette approche introduisent les points sûrs sous forme d'annotation ou d'appels à des fonctions. Ces points sont déterminés par des analyses statiques ou/et dynamiques des applications. Dans [27], les auteurs définissent une analyse permettant d'étiqueter le programme par des expressions *update* à chaque point ou une mise à jour est possible. Une expression *update* sera annotée avec les types qui ne doivent pas être modifiés par celle-ci. Dans [14], l'auteur définit des algorithmes permettant le calcul des points sûr pour des programmes multi thread en se basant sur des représentations graphiques des programmes, comme les graphes de flux de contrôle et les graphes inter procédures. Quand un de ces points est atteint, le système vérifie si une mise à jour a été notifiée et le processus est lancé le cas échéant.
- ***L'approche introspective :*** dans cette approche, lorsqu'une mise à jour est signalée, le système est ramené vers un état dit quiescent. Dans les travaux adoptant cette approche (comme par exemple [17, 16]), un tel état est atteint quand aucune méthode à mettre à jour n'est active (en cours d'exécution). Ces systèmes se basent sur des fonctions permettant l'introspection de l'état de l'application pour retourner des informations concernant les méthodes en cours d'exécution et les objets créés ainsi que des mécanismes permettant d'atteindre l'état quiescent qui correspond à un point sûr où la mise à jour peut être lancée.

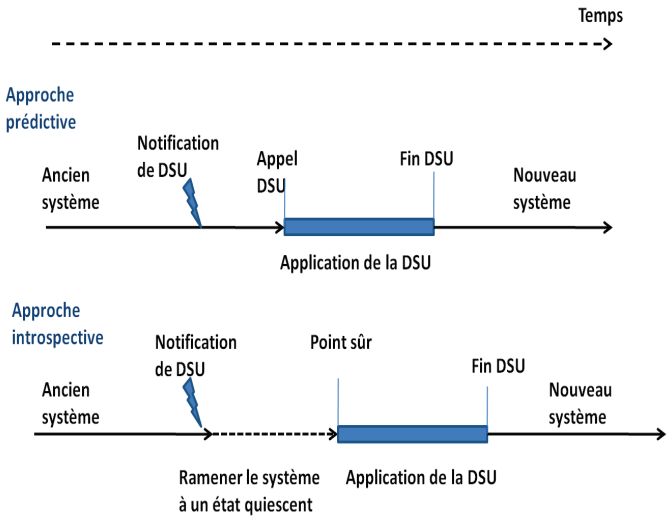

Figure II.4 : Approches pour les moments opportuns de la DSU

L'application de ces approches fait appel à des techniques telles que : les méthodes restreintes, les méthodes actives, l'extraction des boucles et la reconstruction des piles.

II.3.3.1 Méthodes restreintes et méthodes actives

Une méthode est dite restreinte si c'est une méthode concernée par la mise à jour et en cours d'exécution. 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.

La technique de méthodes actives propose de mettre à jour des méthodes actives. Cependant, il faut vérifier que :

- elles ne font pas des appels à des méthodes dont la signature a changé ou à des méthodes supprimées ;
- 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 : 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 ; 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.

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).

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 de déterminer l'instruction où il va falloir continuer l'exécution après la mise à jour du code de f.

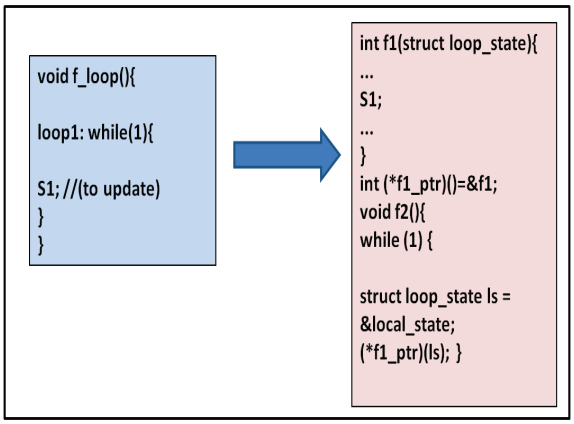

Figure II.5 : Exemple d'extraction de boucle

La technique de recherche de point sûr basée sur les méthodes restreintes peuvent rencontrer un problème de boucle infinie également. 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 les problèmes cités : la technique d'extraction de la boucle et la technique de reconstruction des piles.

II.3.3.2 Extraction des boucles

La technique d'extraction de boucle consiste à associer une autre méthode à l'ensemble d'instructions constituant la boucle. Le programmeur peut marquer les boucles (notamment les boucles à longues itérations) et un compilateur extrait la boucle vers sa propre fonction. On obtient ainsi au moins deux méthodes f1 et f2 (voir Figure II.5 [28]) en extrayant les instructions de la boucle dans une méthode 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.

II.3.3.3 Reconstruction des piles

Cette technique est utilisée dans le but d'apporter des mises à jour à des méthodes actives. Elle 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 ([29]). 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.

Une autre solution propose le remplacement dans la pile (On-Stack Replacement : OSR) [30] qui est utilisée pour les méthodes ayant le même bytecode pour l'ancienne et la nouvelle version. La pile n'est pas reconstruite mais la solution 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.

II.4 Systèmes de DSU

Le but de cette section est de présenter quelques systèmes de DSU. A travers la littérature, plusieurs classifications sont présentées. Certaines classifications adoptent les techniques utilisées comme critère de catégorisation [31]. D'autres proposent de classifier les systèmes selon leurs déclinaisons, par exemple en une catégorie considérant les systèmes de DSU comme des extensions à des systèmes existant tels que certains langages de programmation ou bien certaines machines virtuelles [10]. Nous présentons une classification selon les paradigmes des langages de programmation sur lesquelles sont basées les applications auxquelles sont destinés les systèmes considérés.

II.4.1 Les languages de programmation séquentielle

II.4.1.1 Ginseng

Ginseng [15] est un système de mise à jour dynamique pour les applications serveurs en C. Il est composé de trois parties : un compilateur, un générateur de correctifs et un système d'exécution :

- *Le compilateur :* Ce composant compile le code de telle sorte qu'il supporte les mises à jour dynamique en effectuant des redirections vers les versions les plus récentes pour les fonctions et en faisant en sorte que si un type est modifié, toutes les données de ce type seront transformées pour avoir la récente présentation du type. Ceci est effectué en introduisant une donnée pour garder la version et des fonctions pour noter l'utilisation concrète de chaque donnée d'un type. Le deuxième rôle du compilateur est d'effectuer une analyse statique du programme qui permet de s'assurer que les mises à jour sont sûres. Cette analyse permet de générer des contraintes sur les types qui peuvent être modifiés en un point d'un code.
- *Le générateur de correctifs :* Cette partie de Ginseng permet de comparer les deux versions d'un programme pour produire un fichier qui contient la différence entre elles (les nouvelles fonctions, les nouveaux types, les données modifiées, ...). Elle produit également des fonctions de transformations de types.
- *Le système d'exécution :* Pour effectuer une mise à jour, l'utilisateur envoie un signal au programme. Le système d'exécution charge le correctif, vérifie que les mises à jour qu'il contient sont compatibles avec le point du lancement de la mise à jour : si les contraintes sont satisfaites (contraintes fournies par une analyse statique), les changements nécessaires sont apportés dans le programme, sinon, le système attend un autre point pour effectuer la mise à jour.

Pour la mise à jour du code, Ginseng utilise la technique d'indirection des appels de méthodes pour accéder aux nouvelles versions. Pour chaque fonction, un pointeur est généré pour pointer sur cette fonction. Au début, cette variable pointe sur la version initiale. Ensuite, chaque appel à la fonction est remplacé dans le code par un appel via le pointeur.

Le transfert d'états est assuré en introduisant des fonctions de transformation de types. L'application d'une fonction de transformation d'état est effectuée de la façon suivante : Pour supporter la mise à jour, un champ pour stocker le numéro de version est ajouté à chaque type. Une fonction appelée concrétisation du type est utilisée à chaque fois que le type est utilisé concrètement dans le code (une utilisation est concrète si elle repose sur la structure du type). A chaque appel de cette fonction, une comparaison est effectuée entre le numéro de la version n du type et la dernière version m, si $n < m$, une fonction de transformation d'état est appelée.

Pour éviter les mauvais points pour les mises à jour, Ginseng utilise l'analyse statique appelée analyse de capacité de mise à jour (*updateability analysis*). Celle-ci est basée sur l'annotation des différentes fonctions avec des ensembles contenant les types qui sont utilisés concrètement par une fonction. Cette condition assure qu'à un point de mise à jour aucune utilisation des types concernés par la mise à jour n'est concrète dans la suite du programme.

II.4.1.2 DSU pour systèmes d'exploitation

Le système Ksplice [17] est développé pour la mise à jour dynamique du noyau Linux. Pour ce faire, Ksplice repose sur trois phases : premièrement un correctif est généré en comparant les fichier ELF (Executable and Linkable Object) de l'ancienne et de la nouvelle version. Après, un code de remplacement est généré, ce code contient la résolution des adresses et des symboles. Le système se base ensuite sur la réécriture du code binaire pour insérer les changement dans le noyau en cours d'exécution.

Pour la mise à jour du code, 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.

Le système K42 [32] est basé sur la programmation orientée objet. L'application à mettre à jour est donc modélisée comme un ensemble d'objets où chaque objet exporte une interface en déclarant une classe virtuelle de base. Chaque ressource (la mémoire virtuelle, une connexion réseau, fichiers ouverts, ou un processus) est gérée par un ensemble d'instances d'objets. Chaque objet encapsule les méta-données nécessaires à la gestion de la ressource ainsi que les verrouillages nécessaires pour manipuler les méta-données. Dans K42, l'ancien code est remplacé par le nouveau code de façon globale. 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. Chaque objet est invoqué à l'aide d'indirection d'appel à travers une table de translation d'objets (OTT). 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. K42 utilise la technique de clusterisation pour les instances. Ce mécanisme permet à un objet de contrôler sa propre distribution entre les processeurs.

II.4.1.3 DSU pour systèmes embarqués

Les auteurs présentent dans [33] un système de mise à jour dynamique pour des systèmes embarqués basés sur le langage C. Le système présenté est basé sur FreeRTOS : un système d'exploitation temps réel (RTOS) faible empreinte, portable, préemptif et Open Source pour microcontrôleurs. C'est l'un des systèmes d'exploitation les plus utilisés actuellement. Il est utilisé principalement pour les systèmes embarqués qui ont des contraintes d'espace pour le code, mais aussi pour des systèmes de traitement vidéo et des applications réseaux qui ont des contraintes temps réel. Le système de mise à jour est une extension du système FreeRTOS. Dans cette approche, la mise à jour du code est basée sur la résolution dynamique des liens de fichiers en format binaire.

La mise à jour du code est basée sur une séparation des tâches FreeRTOS sous la forme de fichiers binaires ELF (Executable and Linkable Format). Ceci offre la possibilité de refaire l'édition des liens durant l'exécution et de ce fait, ajouter et supprimer des fichiers binaires durant l'exécution. L'édition des liens dynamiques permet de compiler les tâches puis de les insérer dans le système en cours d'exécution.

Le choix des points de mise à jour est basé sur l'insertion d'annotations par le programmeur. L'état d'une tâche est dit sûr pour la mise à jour si des événements externes telle que la communication inter-tâches ne perturbent pas l'état de la tâche. Le transfert d'état (variables et structures de données) est effectué en stockant, après la détection d'un point sûr, temporairement le contexte dans un segment prédéfini et dédié de la mémoire. Ceci permet à la nouvelle version de continuer avec le même état (les mêmes addresses) que l'ancienne version. Ceci est dû à la limite des systèmes embarqués en matière d'espace mémoire. Dés que le nouveau code est disponible (par l'édition dy-

namique des liens), l'état est transféré de l'emplacement temporaire vers son emplacement définitif.

Dans le même contexte, les travaux [34] proposent un système de DSU appelé DSSUS (Dynamic Satellite Software Updating System) pour la mise à jour de systèmes pour satellites basés sur la plateforme Vxwork. Il est constitué principalement de module de mise à jour dynamique et un vérifieur de possibilité de mise à jour basé sur une analyse des dépendances entre les données et les modules de l'application. Le système adopte l'approche des méthodes restreintes pour les points sûrs.

II.4.1.4 MUC

le système MUC (Multi-version execution for Updating of Cloud), présenté dans [35], est un système de mise à jour dynamique pour les applications cloud écrites en langage C. L'approche est basée sur quatre parties. La première consiste en une analyse statique permettant de regrouper des informations nécessaires à la mise à jour : un fichier de mise à jour indiquant les modifications à effectuer, un fichier des systèmes d'appel récapitulant l'ensemble des appels contenus dans l'application et un fichier des communications inter-process.

A la réception d'une commande de mise à jour, la deuxième partie du système lance, par une commande fork, un processus qui est une copie de l'ancienne version. La troisième étape est consacrée à la synchronisation. Cette partie garantit que les deux process communiquent avec l'environnement comme étant un seul process (la synchronisation s'effectue au niveau des appels systèmes). Cette étape utilise le fichier d'appels généré par la première partie.

II.4.2 Les langages orientés objets

II.4.2.1 DVM

DVM (Dynamic Virtual Machine) [21] 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 étend 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). Pour la mise à jour des données, DVM utilise un algorithme similaire à l'algorithme de ramasse-miettes 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. La recherche de points sûrs dans 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.

II.4.2.2 Jvolve

JVolve [30] est un système de mise à jour dynamique implémenté sur Jikes RVM (Jikes Research Virtual Machine). JVolve intègre et étend le chargeur de classes dynamique de Jikes RVM, le compilateur JIT (Just In Time) , le planificateur de threads, le ramasse-miettes et le support de remplacement sur pile (On-Stack Replacement : OSR) et utilise un module *return-barrier* pour implémenter la mise à jour dynamique. Le processus de la mise à jour dans JVolve est constitué en deux étapes distinctes : la première consiste en la génération de la spécification de la mise à jour par l'outil de préparation de mise à jour UPT (Update Preparation Tool). La seconde consiste en l'application de la mise à jour à la détection du point sûr.

La préparation de la mise à jour permet de déterminer la différence entre l'ancienne version et la nouvelle version de l'application : les fichiers classes modifiés et les fichiers classes supprimés et ceux ajoutés. Ceci constitue la première partie de la spécification de la mise à jour. La deuxième partie, informe la machine virtuelle de la façon de traiter les modifications apportées aux données.

En donnant une spécification de mise à jour, l'utilisateur signale à la machine virtuelle la présence d'une mise à jour à appliquer. Celle-ci charge alors les nouvelles classes dans le programme en cours d'exécution par l'intermédiaire du *classloader* standard et planifie la mise à jour. JVolve vérifie ensuite si la machine virtuelle est à un point sûr. Un point sûr exige qu'aucune pile de thread ne contienne une méthode restreinte. Les méthodes restreintes sont de trois catégories : les méthodes changées par la mise à jour ; les méthodes qui font référence à des classes mises à jour et les méthodes spécifiées par l'utilisateur en mesure de sécurité. Si des méthodes restreintes sont sur la pile, la machine virtuelle installe le *return-barrier* pour la première et la troisième catégorie et effectue le OSR pour la deuxième catégorie pour atteindre un point sûr.

Cette technique vise à recompiler les méthodes dans la seconde catégorie même si elles s'exécutent, à condition qu'elles ne contiennent pas des appels à des méthodes mises à jour. Une fois que tous les threads de l'application sont synchronisés au point sûr, JVolve applique la mise à jour . Il annule d'abord les versions compilées de toutes les méthodes changées. Ces méthodes sont recompilées en fonction des besoins. La machine virtuelle initie ensuite le ramasse-miettes (Garbage Collector) pour détecter tous les objets existants dont les classes sont mises à jour. Il alloue des objets qui sont conformes aux définitions des nouvelles classes. À la fin de l'application du ramasse-miettes, JVolve exécute les transformateurs de classe ainsi que les transformateurs d'objets, le premier pour initier les champs statiques et le second pour initier les champs d'instance.

II.4.2.3 DUSC

DUSC (Dynamic Update through Swapping of Classes) [18] est un système de mise à jour dynamique des applications Java qui est complètement défini au niveau du programme. Par conséquent, cette technique n'exige aucun appui de la JVM. Cette technique fonctionne en premier temps en modifiant statiquement l'application afin de permettre la mise à jour dynamique puis en effectuant le remplacement dynamique des classes au moment de l'exécution, lorsqu'une nouvelle version de l'une ou plusieurs classes est disponible. DUSC est basée sur l'utilisation des classes proxy. Cette technique permet de mettre à jour une application Java en cours d'exécution par substitution, l'ajout et la suppression des classes.

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 méthodes actives 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.

DUSC présente quelques limitations, notamment il ne permet pas la mise à jour des signatures des méthodes et les interfaces des classes.

II.4.2.4 Dynamic C++

Dans [36], les auteurs présentent un système de mise à jour dynamique pour la gestion des connections dans un réseau de télécommunications. Le système est basé sur les classes dynamiques permettant d'introduire de nouvelles fonctionnalités dans les classes C++. La solution est implémentée en utilisant les classes proxy. Chaque classe est écrite en deux parties : une interface abstraite et une ou plusieurs implémentations qui héritent de l'interface.

Une implémentation correspond à chaque version de la classe. La résolution indirecte est utilisée pour faire appel à l'implémentation actuelle d'une classe. Une table permet d'associer chaque nom de classe à la version en cours de l'implémentation. La mise à jour des instances d'objets considère que chaque objet créé correspond à la nouvelle version. Les objets de l'ancienne version continuent à utiliser la version qui les a créés. Ces derniers sont détruits à la fin de leurs tâches. Des méthodes spécifiques sont introduites pour permettre à un objet de determiner si sa version est la plus récente et offrent aux programmeurs la possibilité de transférer l'état des objets de manière explicite.

II.4.3 Les langages fonctionnels

II.4.3.1 ReCaml

Le système ReCaml [37] représente un nouveau langage de programmation fonctionnel, conçu pour la manipulation des états d'exécution d'un programme d'une manière sûre. Le système est basé sur un lambda-calcul simplement typé. Il est considéré comme une extension de l'interpréteur de code octets Caml.

ReCaml permet la mise à jour des fonctions actives. Le programmeur annote le code à des points permettant la mise à jour dynamique. Quand une mise à jour est signalée, l'exécution est restaurée à l'annotation la plus proche pour appliquer la mise à jour. Le programmeur fournit également des fonctions de transfert d'états à exécuter au moment de la mise à jour. Les états d'exécution sont modélisés en utilisant le concept de continuations. Une continuation est une représentation abstraite des états d'un programme. Elle permet de représenter le déroulement d'un programme à un point donné et d'avoir une perspective sur ce qui serait dérivé à partir d'un point de l'exécution d'un programme.

La mise à jour est effectuée par la manipulation des continuations en définissant un nouveau constructeur propre à ce langage, appelé *match_cont*. Cet opérateur permet d'implémenter plusieurs politiques de mise à jours des états (finir d'abord l'ancienne version, combiner des résultats de l'ancienne version avec le nouveau programme ou recalculer entièrement selon la nouvelle version). Il fait correspondre une continuation avec des appels décomposés de la pile d'activation de méthodes. Le programmeur de la mise à jour spécifie les opérations à exécuter sur les piles. Le typage permet d'assurer la correction des opérations.

II.4.3.2 DynamicML

Dans leurs travaux [38], les auteurs présentent DynamicML, un système de mise à jour dynamique pour les programmes ML. Le système se présente comme une extension au langage ML. Ce dernier étant un langage avec typage statique fort et polymorphe : le transtypage (cast en C) implicite est strictement interdit, ce qui supprime un grand nombre de bogues possibles. Le système de types évolué offert par le langage permet de définir précisément les types et les opérations autorisées sur les types et les structures de données. Il est possible de définir des fonctions génériques et d'écrire des fonctions qui prennent d'autres fonctions en paramètres (fonctions dites d'ordre supérieur)

Le système permet le remplacement dynamique des modules ML. La nouvelle version d'un module doit avoir la même signature que l'ancienne ou bien en être un sous type. La mise à jour est restreinte aux types de données abstraits. Le système DynamicML ne permet pas une coexistence de versions. La mise à jour est effectuée à un état quiescent du module : aucune de ses fonctions ne doit être dans la pile des appels. Des fonctions de transfert d'états, appelées fonctions d'installation de la mise à jour, sont fournies par le programmeur en vue d'automatiser le processus de mise à jour. L'étape de la compilation vérifie la présence de ces fonctions et de ce fait garantit la cohérence de la mise à jour. Le système dispose d'un mécanisme de rollback : dans le cas où une exception est

levée durant le processus de mise à jour, le processus de mise à jour est interrompu et le système est ramené par ce mécanisme à l'état initial.

Ce système pose des restrictions sur l'expressivité des mises à jour mais permet d'assurer la correction de la mise à jour par l'utilisation de concepts inhérents à la programmation fonctionnelle (comme la notion des fonctors : fonctions qui prennent des modules comme arguments). L'utilisation d'un algorithme de garbage collection pour la mise à jour permet également de standardiser l'implementation de la mise à jour.

II.4.4 Les programmes multi thread

II.4.4.1 POLUS

POLUS [25] supporte la mise à jour dynamique pour des systèmes multi threads. La mise à jour du code dans POLUS utilise un module de générateur de correctifs. Les correctifs sont compilés par un compilateur standard. Il utilise aussi un injecteur de correctifs qui a pour but d'insérer les mises à jour 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. 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). Pour la 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.

II.4.4.2 UpStare

Upstare [29, 39] un système de DSU qui a été développé dans le cadre des travaux de recherche sur la tolérance aux fautes. Il est dédié à la mise à jour dynamique des applications C multi threads. Il est composé de plusieurs modules tels que le compilateur, le générateur de correctifs, et un module de mise à jour proprement dit. Pour la mise à jour du code, Upstare compile en instrumentant une application en y ajoutant notamment des informations nécessaires à son processus de mise à jour. Il utilise la technique d'indirection pour les appels de fonctions. Pour la 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. Le système instrumente les points d'entrées et de sorties des fonctions et des boucles, ceci permettant de déterminer les points sûrs. Après détection d'un point sûr, il suspend tous les threads associés aux applications et applique une mise à jour atomique à l'aide d'un thread dit *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 multi threads et qui sont le plus souvent en attente de requêtes.

II.4.5 Les programmes par composants

II.4.5.1 DSU pour systèmes d'automatique

Dans [40], les auteurs présentent un système de mise à jour dynamique pour les systèmes automatiques. La solution présentée est basée sur l'architecture FASA (Future Automation System Architecture). C'est une architecture à base de composants pour les applications temps réel. Premièrement, le code du nouveau composant est chargé en mémoire. Le système crée alors une nouvelle configuration, qui est d'abord une copie de l'actuelle configuration. Un mécanisme de synchronisation d'états permet de transférer l'état de l'ancien composant vers le nouveau. Ce mécanisme est basé sur une trace des changements d'états.

Le système implémente une mécanisme "switchover" qui garantit que toutes les mises à jour sont activées en même temps sur tous les contrôleurs impliqués. La sûreté est assurée par un mécanisme de restauration du contexte initial si le composant mis à jour se comporte d'une façon imprévue.

II.4.5.2 DSU pour les systèmes OSGi

Le système OSGi (Open Services Gateway initiative) spécifie une plateforme de services fondée sur le langage Java. Le framework implémente un modèle de composants dynamique et complet où les applications et composants sont sous la forme de bundles : archives Java représentant l'unité de déploiement.

Dans [41], les auteurs présentent une approche de mise à jour dynamique basée sur l'analyse des bundles. La préparation de la mise à jour est effectuée en analysant les bundles de l'ancienne et la nouvelle version de l'application pour détecter les différences (classes ajoutées ou modifiées par exemple). Des fonctions de transfert d'états sont ajoutées au fichier de mise à jour pour spécifier principalement deux niveaux : l'adaptation de l'architecture et l'adaptation d'interface. Le système est décomposé en deux parties : un module de mise à jour structuré en bundles représentant un service de mise à jour et une machine virtuelle qui implémente principalement des fonctionnalités d'introspection et un système de detection de point sûr.

Dans [42], les auteurs présentent deux techniques pour la mise à jour. Dans la première, le nouveau service, portant le même nom que l'ancienne version est enregistrée, avant la mise à jour, avec une priorité plus élevée. De ce fait, il est automatiquement sélectionné par le client. Un point de mise à jour est alors signalé pour effectuer un transfert d'état. La deuxième technique est basée sur une combinaison de rechargement dynamique et compilation en temps réel. Un chargeur dédié est défini pour les nouveaux services. Quand un changement dans l'implémentation est détecté, le code source est recompilé et rechargé. Un système de classes proxy est utilisé pour permettre de rediriger les appels vers les services adéquats.

II.4.5.3 DSU pour les systèmes distribués

Dans [43], les auteurs décrivent une infrastructure de mise à jour qui permet la re-programmation des noeuds en se basant sur le code binaire des applications. La préparation de la mise à jour est effectuée par un noeud dédié. Le système construit une modularisation de l'application pour détecter les différentes dépendances qui se trouvent entre ses éléments.

Quand une fonction est modifiée, le système permet d'identifier les fonctions s'y référant et construit un graphe de dépendances permettant de montrer les liens et leurs localisations sur le réseau. Ces informations servent de base à un processus semi automatique basé sur plusieurs politiques pour determiner la possibilité de la mise à jour. Le cas échéant, le système crée un modèle d'image mémoire représentant l'utilisation de la mémoire dans le noeud. Le modèle est initialisé avec l'état initial du noeud et utilisé pour garder trace de l'agencement de l'application. Ces informations permettent au système de localiser les emplacements où mettre le code modifié.

II.4.5.4 DSU pour l'Internet des objets

L'Internet des objets (Internet Of Things : IOT) représente une évolution d'Internet vers les échanges d'informations et de données provenant des dispositifs présents dans le monde réel vers le réseau Internet. Ces dispositifs proviennent de domaines variés (santé, transport...). Cette évolution permet d'améliorer la capacité à rassembler, analyser et restituer des données pouvant être transformées en informations et connaissances. Dans [44], les auteurs présentent un cadre pour la mise à jour dynamique pour l'IOT. Le but est d'assurer une haute disponibilité des capteurs de transmission de données et les services de coopération dans IOT.

La solution suppose une exécution périodique des mises à jour et définit un cadre de gestion de mise à jour des capteurs (*update management framework*) selon leurs contextes. Un pack de service de mise à jour est défini entre un coordinateur de mises à jour et les capteurs. La solution propose plusieurs algorithmes pour le service de mises à jour selon le nombres de capteurs à gérer et le nombre de capteurs impliqués dans une mise à jour. Le système utilise la réplication de la version originale du programme. Les replicas deviennent invalides après la mise à jour. Chaque capteur contient une table d'informations concernant les moments de la mise à jour. Ces informations sont utilisées pour le routage et la synchronisation de la mise à jour et de ce fait, permettant d'assurer la consistance du système.

II.5 Correction de la mise à jour

II.5.1 Correction basée sur le test

Le test du logiciel est une approche de vérification destinée à s'assurer que ce logiciel possède effectivement les caractéristiques requises pour son contexte d'utilisation. Ceci est effectué en décrivant le contexte du logiciel, les fonctionnalités attendues ou les situations dangereuses à considérer à chacune des étapes du développement. Le test a pour objectif de détecter les éventuels écarts entre le comportement attendu et le comportement observé au cours des tests, et ainsi éliminer un grand nombre de fautes présentes dans le système. Le deuxième objectif est d'obtenir la confiance nécessaire avant l'utilisation opérationnelle du système. Dans le cadre de la mise à jour dynamique, le test logiciel est utilisé pour la vérification d'un programme avant, durant et après la mise à jour.

L'évaluation des systèmes s'effectue généralement sur la base de métriques relatives aux critères de DSU (flexibilité, efficacité, robustesse et facilité d'utilisation). Les critères sont évalués sur un ensemble d'applications jugées, par les chercheurs convaincantes.

Certains travaux comme [18] se basent sur des plateformes de tests existantes. Dans [28, 45], les auteurs utilisent une plateforme de tests existante mais proposent leur propre méthodologie baptisée *DSUtest*. Elle consiste à enregistrer dans un premier temps tous les points de mise à jour atteints durant l'exécution d'un test. Ensuite, tous ces points sont testés vis-à-vis de propriétés définies par l'utilisateur exprimés sous forme de spécifications. Les points de mises à jour pouvant atteindre un grand nombre, les auteurs proposent une minimisation, prouvée correcte, de ceux ci vers des classes d'équivalences.

Dans leurs travaux [10, 46], les auteurs définissent le premier cadre général de test systématique dédié à la mise à jour dynamique des programmes. Ce cadre, baptisé *Tedsuto*, est basé sur l'idée de vérifier les mises à jour pouvant aboutir à des erreurs en se basant sur l'exécution de plusieurs tests et explorer l'évolution de l'application durant l'exécution du test. Chaque évolution vers une mise à jour correcte est appelée "opportunité de mise à jour". Ce cadre est centré sur les erreurs relatives au moment opportun de l'application de la mise à jour.

Le système est basé sur la séparation de la partie mise à jour (le système de mise à jour et le

programme à mettre à jour) et la partie test. La partie test est composée d'un système de test et d'un observateur de tests. Les deux parties communiquent par une IPC (Inter Process Communication).

Le système de test envoie des requêtes à l'application à mettre à jour. Le déroulement de celle ci aboutit à plusieurs opportunités de mise à jour. A chaque opportunité détectée, le système d'observation est sollicité pour décider d'appliquer ou non la mise à jour. Le système d'observation notifie le système de test sur la présence de l'opportunité. En se basant sur les informations concernant le test en cours d'exécution, le système de test envoie une réponse au système observation qui notifie alors le système de mise à jour dynamique avant de retourner à l'application à mettre à jour qui répond au système de tests. Le système supporte plusieurs catégories de tests, par exemple, les tests orientés mise à jour, orientés opérations et les tests de synchronisation.

II.5.2 Correction basée sur les méthodes formelles

Cette catégorie désigne l'utilisation des concepts et techniques issues des mathématiques ou de la logique formelle pour spécifier, développer et raisonner sur les systèmes de mises à jour dynamique et leurs propriétés. L'utilisation des méthodes formelles pour la correction de la mise à jour dynamique passe par :

- La définition des critères de correction ;
- le choix d'un paradigme formel pour la spécification et la vérification ;
- la spécification des systèmes de mise à jour, les applications à mettre à jour et des propriétés de correction ;
- établir les propriétés par des techniques formelles.

Un état de l'art sur les travaux appartenant à cette catégorie sera présenté au chapitre suivant.

II.6 Conclusion

Nous avons présenté dans ce chapitre un état de l'art sur la mise à jour dynamique des programmes. La mise à jour dynamique est une technique qui répond aux besoins d'évolution et de haute disponibilité requis par certains types d'applications. Nous avons expliqué les différents problèmes scientifiques soulevés par la mise à jour dynamique au niveau des codes, des données, du choix des moments opportuns et de la correction des changements. L'intérêt de la DSU est reflété par les nombreux travaux de recherches qui sont consacrés à l'étude des différentes problématiques. Ceci a donné lieu à plusieurs techniques et au développement de plusieurs systèmes. L'étude des problématiques posées par la DSU nous a mené à aborder la correction de la DSU qui a fait objet de nombreuses recherches et qui sera abordée en détail dans le prochain chapitre.

Chapitre III

Correction formelle de la DSU

Sommaire

$\mathcal{L}$a mise à jour dynamique des programmes est utilisée dans des domaines requérant une haute disponibilité des applications. Ces applications sont souvent critiques : une erreur peut mener à des pertes matérielles, voir humaines, considérables. Dans ce type d'applications, l'apport de la DSU en terme de haute disponibilité ne doit pas être contre balancé par des erreurs que celle-ci est susceptible d'introduire. L'utilisation des méthodes formelles pour établir la correction de la DSU apporte le degré de rigueur requis par les applications critiques. Dans ce chapitre, nous allons présenter les différents critères sur lesquels est basée la correction de la DSU. Ensuite, nous présentons un état de l'art sur les différentes techniques et méthodes formelles utilisées dans le but d'établir ces critères. Une classification sur la base des paradigmes formels est proposée pour l'étude de cet état de l'art.

III.1 Critères de correction de la DSU

III.1.1 Indécidabilité de la correction de la DSU

Idéalement, afin d'établir la correction de la mise à jour dynamique, le but est d'assurer que le comportement de l'application après la fin de la procédure de la mise à jour correspond au comportement qui aurait été obtenu en installant la nouvelle version de l'application directement.

Dans [3, 47], les auteurs démontrent l'indécidabilité de la validité de la mise à jour. Cela veut dire qu' on ne peut pas développer un algorithme qui, étant donné les codes P et $P1$, la fonction

de transfert S et un état s du programme, répond si la mise à jour avec ces paramètres est valide ou non. Par conséquent, la validité de la mise à jour est assurée par la définition d'un ensemble de conditions qui portent sur :

- L'état à partir duquel est effectuée la mise à jour : définir des conditions que doit satisfaire un état d'un process pour installer la mise à jour ;
- Le calcul d'un certain nombre de points de contrôle dans un programme et une correspondance entre points de contrôles dans les deux versions ;
- La notion d'état dépend du modèle du langage choisi. Par exemple, pour un langage simple (impératif, sans fonctions) on prend le contenu des variables et le compteur ordinal, pour un langage orienté objet on considère le tas où sont créées les instances.

Cet ensemble de conditions implique que la modification est une extension fonctionnelle de l'ancienne version modulo un ensemble de fonctions de transfert d'états. Une procédure $p1$ est une extension fonctionnelle d'une procédure $p0$ si le process est dans le même état dans les cas suivants : la procédure $p0$ est exécutée dans un état s puis le process est mis à jour ou, le process est mis à jour à l'état s et après la procédure $p1$ est exécutée.

Pour plusieurs auteurs ([48, 14, 49]), ces conditions sont très restrictives vis-à-vis de l'expressivité de la DSU. Plusieurs recherches ont donné lieu à la définition de plusieurs critères de correction de la DSU. La correction de la DSU ne repose donc pas sur une définition unique mais sur la définition d'un ensemble de critères. Ces critères sont exprimés sous formes de propriétés portant sur les différents aspects de la DSU : les codes, les données et les points sûrs.

L'étude des critères de correction de la mise à jour dynamique fait ressortir deux catégories. La première catégorie regroupe les critères qui sont communs à toutes les mises à jour considérées comme par exemple la sûreté du typage. La deuxième catégorie désigne des propriétés relatives à la sémantique des programmes mis à jour et sont donc spécifiques à chaque programme.

III.1.2 Les critères de correction communs

III.1.2.1 L'atteignabilité

Ce critère a été présenté dans [3, 47]. Les auteurs proposent un cadre formel pour la modélisation de la DSU et définissent des conditions permettant de garantir sa validité. Ce travail est basé sur la notion d'atteignabilité (*reachability*). Les auteurs définissent un process comme étant l'exécution d'un code P. Un process est caractérisé par le code P et un état s qui change selon une fonction de transition (modèle d'exécution). Un état s, pour un programme P, est dit atteignable si et seulement si l'exécution de P à partir d'un état initial peut atteindre, après un certain temps, l'état s. Lors de la mise à jour dynamique, le comportement d'un process est changé de P vers P1. Cette mise à jour peut nécessiter la modélisation du changement d'état pour par exemple initialiser une variable qui a été ajoutée. Ce passage est modélisé par la définition d'une fonction de transfert d'états. La définition de cette fonction est basée sur la connaissance des sémantiques des deux version P et P1. Cette fonction est supposée donnée par le programmeur. Une mise à jour dynamique d'un process à partir d'un code P vers P1 à un instant t en utilisant la fonction de transfert S est valide si après le changement, le process arrive à un état atteignable de P1 après une période de temps finie. C'est-à-dire qu'après une certaine période, le process commence à se comporter comme s'il exécute la deuxième version du code (P1) à partir d'un état initial.

III.1.2.2 La con-freeness

Dans [27], les auteurs ont défini la correction de la DSU sur la base de la notion de con-freeness. Le critère stipule qu'une mise à jour dans un programme doit s'effectuer lorsque celui-ci est dans une configuration dite de *con-freeness*. Cette configuration assure qu'aucune utilisation des types

concernés par la mise à jour n'est concrète (d'où le terme) dans la suite du programme, après le point de DSU.

Les auteurs distinguent deux types d'utilisations pour une donnée d'un type t : une utilisation concrète qui repose sur la structure du type et une utilisation abstraite ou la structure du type n'est pas considérée. Les expressions dites de coersion permettent alors d'identifier, dans un programme, quand une donnée est utilisée concrètement et quand elle est utilisée abstraitement. Ces informations sont utilisées comme ceci : on ne peut pas effectuer une mise à jour sur un type alors qu'il est utilisé concrètement dans une autre partie du programme (qui est en cours de s'exécution).

Des fonctions de transformation et de vérification de types permettent d'assurer la cohérence des types. Cette notion est relative à la transformation des types. Elle exprime qu'on ne doit à un aucun moment avoir des parties du code qui manipulent des représentations différentes pour un même type.

Le principe de la méthode consiste à étiqueter le programme par des expressions *update* à chaque point ou une mise à jour est possible avec les types qui ne doivent pas être mis à jour pour garantir la notion de con-freeness et assurer ainsi que le programme est bien formé après le processus de la mise à jour.

III.1.2.3 La sûreté de typage

La sûreté de typage [25, 37, 49] est un critère clé dans la définition des programmes bien formés. En effet, la sûreté de typage permet de garantir que les types de données dans un programme pour les constantes, les variables, et les méthodes sont conformes au système de typage du langage de l'application. Cela permet de garantir par exemple que : aucune erreur due à une opération incorrecte en raison du type des données ne peut se produire ou que le résultat de l'évaluation d'une expression est une valeur d'un type compatible avec le type de l'expression. L'utilisation de ce critère pour la correction de la DSU [49] permet de garantir par exemple que pour une fonction f définie comme $f(A\ a)\ldots$ dans la première version d'un programme, si le nouveau programme change le type de l'argument A, alors après la mise à jour, un appel à la fonction f doit être effectué avec la nouvelle version de A. Dans [50, 51], dans un contexte de MPI (Message Passing Interface), ce critère permet de garantir qu'un thread ne recevra jamais une valeur ayant un type différent de ce qui est attendu.

III.1.2.4 La consistence

Le problème de la consistence de la mise à jour est posé dans le cas ou il y'a une coexistence des codes ou des données. C'est un critère qui se réfère à l'utilisation des bonnes versions des codes et des données lors de la mise à jour et après celle-ci. L'inconsistence du code peut arriver par exemple [52] si on considère une boucle principale d'un programme qui fait l'encodage/décodage de messages. La fonction est invoquée au début et à la fin de chaque interaction. La mise à jour d'une telle fonction peut entraîner qu'un message soit codé avec un algorithme de chiffrement et décodé avec un autre, ce qui cause une erreur dans le système. De même, l'utilisation de deux versions différentes d'une donnée cause des erreurs dans le système dans le cas où une fonction accède à deux versions différentes d'un même objet par exemple. Plusieurs techniques sont utilisées pour assurer la consistence comme les fonctions d'indirection [53], la synchronisation [54] et le mécanisme des transactions [55].

III.1.2.5 L'absence d'arrêt brutal

Cette propriété [49] permet de garantir qu'une mise à jour correcte ne causera pas une fin brutale du système cible durant et après la mise à jour. Un arrêt brutal peut être causé par des erreurs

dans les correctifs ou les procédures de mise à jour. Des vérifications systèmes [49] et de correctifs [5] sont utilisées pour palier à ces erreurs.

III.1.2.6 L'absence d'inter-blocage

Un ensemble de processus est en inter-blocage si chaque processus attend la libération d'une ressource qui est allouée à un autre processus de l'ensemble. Comme tous les processus sont en attente, aucun ne pourra s'exécuter et donc libérer les ressources demandées par les autres, ils attendront tous indéfiniment. Lors de l'application de la DSU, le programmeur doit s'assurer qu'il n'introduit pas des appels aux fonctions de mise à jour qui peuvent introduire des inter-blocages dans le système [50, 56].

III.1.2.7 Sûreté d'activation

Cette propriété permet de garantir, dans certains systèmes de DSU, que la mise à jour s'applique uniquement si les fonctions (ou les composants [57]) qui sont concernées par la mise à jour ne sont pas en cours d'exécution. Ceci est basé sur des fonctions d'introspection de l'environnement d'exécution pour définir des points sûrs pour la mise à jour [58, 34, 28]. Ces points permettent de définir les états quiescents caractérisés par l'absence de méthodes à mettre à jour dans l'environnement d'exécution.

III.1.2.8 Garantie de la mise à jour

Cette propriété [59] signifie qu'une fois une requête de mise à jour effectuée, le système en cours d'exécution parviendra à un état permettant d'effectuer la mise à jour. C'est une propriété temporelle qui permet de s'assurer que le mécanisme de mise à jour ramène le système en cours d'exécution à des points sûrs qui satisfont les propriétés définies par l'utilisateur.

III.1.3 Les critères de correction spécifiques

Les propriétés présentées dans la section précédente sont communes aux différentes mises à jour apportées par le système. Il s'agit d'établir par exemple que le système n'introduit pas d'erreurs de typage ou garantit la consistence des programmes mis à jour. Un autre type de correction consiste à établir des propriétés sur la sémantique ou comportements des programmes mis à jour. Nous parlons alors de propriétés spécifiques car elles sont propres à chaque programme mis à jour. Ceci requière l'écriture des spécifications formelles des programmes et du changement de leurs comportements.

Ces critères reposent sur des techniques permettant par exemple d'écrire les spécifications au coeur même du code source du programme à mettre à jour [60] pour générer ensuite un ensemble de formules logiques qu'il faut démontrer pour établir la correction du programme. Une autre technique permet aux utilisateurs d'écrire des spécifications dans les correctifs sur la base de plusieurs modèles qui permettent de guider le processus de la mise à jour [14]. Dans [51], une autre technique est utilisée : il s'agit de l'extension d'un système de type avec les effets de bord des programmes.

Le Tableau III.1 propose une présentation des différents critères de correction en montrant les paradigmes de langages de programmation où les critères sont considérés. Le tableau met en évidence les domaines d'applications quand ceci est mentionné dans les travaux de recherches. Il montre ainsi l'intérêt porté aux différents critères dans des types d'applications générales ou critiques dans les différents styles de programmation.

Paradigmes	Propriétés	Domaines d'application
Séquentiel	- **Sûreté de typage :** [15, 2, 49, 53, 61]. - **Consistance :** [62, 11, 49, 53, 63, 35]. - **propriétés spécifiques :** [48, 64, 63]. - **Atteignabilité :** [3, 47, 27]. - **Absence d'arrêt brutal :** [49, 53]. - **Sûreté d'activation :** [17, 28]. -**Con-freeness :** [27].	- **Général :** [64, 11, 2, 15, 3, 48, 49, 47, 27, 53, 28, 61, 63]. - **Systèmes distribués :** [62]. - **Systèmes d'exploitation :** [17]. - **Applications Cloud :** [35].
Orienté objet	- **Sûreté de typage :** [21, 36, 18, 30, 23]. - **Consistance :** [58, 16, 36, 65, 56, 66]. - **con-freeness :** [34]. - **Absence d'inter-blocage :** [56]. - **Sûreté d'activation :** [58, 34, 30].	- **Général :** [21, 30, 56, 23]. - **Systèmes d'exploitation :** [16, 65]. - **Systèmes embarqués :** [58, 34]. - **Réseaux et télécommunication :** [36].
Fonctionnel	-**Sûreté de typage :** [38, 4, 37]. - **Consistance :** [38].	- **Général :** [38, 4, 37].
Multi thread	- **Sûreté de typage :** [39, 54, 51]. - **Consistance :** [39, 25, 54, 56]. - **Absence d'arrêt brutal :** [25]. - **Absence d'inter-blocage :** [51, 56, 50]. - **Propriétés spécifiques :** [51, 50, 67].	- **Général :** [39, 25, 54, 50, 51, 56].
Orienté composants	- **Sûreté de typage :** [42, 68]. - **Consistance :** [41, 69, 70, 40]. - **Propriétés spécifiques :** [70, 68, 71]. - **Absence d'arrêt brutal :** [40]. - **Sûreté d'activation :** [43].	- **Général :** [42, 71]. - **Applications orientées services :** [68]. - **Systèmes distribués embarqués :** [43]. - **Systèmes de télécommunication :** [69]. - **Systèmes automatiques :** [40]. - **Systèmes distribués :** [70].

Tableau III.1 : Critères de correction dans les différents paradigmes et domaines d'application

III.2 Application des méthodes formelles à la DSU

III.2.1 Définition des méthodes formelles

L'expression *méthodes formelles* désigne l'utilisation de concepts et techniques issus de la logique formelle ou des mathématiques pour spécifier, développer et raisonner sur des systèmes informatiques et leurs propriétés [72]. Comme la définition l'indique, les méthodes formelles peuvent être utilisées lors des différentes étapes du processus de développement d'un système. Une méthode formelle doit fournir :

- Un langage formel permettant de décrire ou modéliser le système et ses propriétés ;
- Une démarche articulée autour d'un ensemble d'outils pour raisonner sur des éléments de ce langage.

Á la base des méthodes formelles, se trouve la notion de spécification formelle [73]. Spécifier un programme consiste à donner sous une forme explicite son comportement, c'est-à-dire décrire ce qu'il fait, mais pas comment il le fait. Il s'agit d'exprimer quelles sont ses données et quelles propriétés elles vérifient. Il s'agit aussi de spécifier quels sont les résultats (sorties) du programme et quelles propriétés ils doivent vérifier. Une spécification est dite formelle si :

- Elle est écrite en suivant une syntaxe bien définie, comme celle des langages de programmation ;
- La syntaxe est accompagnée d'une sémantique rigoureuse qui définit des modèles mathématiques représentant les réalisations acceptables de chaque spécification syntaxiquement correcte ;
- La syntaxe et la sémantique sont accompagnées de règles de déduction qui permettent de démontrer des propriétés d'une spécification.

III.2.2 Les méthodes formelles dans la DSU

Selon le niveau de formalisation [72], nous distinguons trois niveaux pour l'application des méthodes formelles dans la DSU. Cette catégorisation correspond à la classification des méthodes formelles selon les niveaux de rigueur des formalisations :

- Le premier niveau : Ce niveau correspond à l'utilisation de concepts et notations des logiques et des mathématiques comme la théorie des ensembles, les fonctions et les systèmes de types. les outils fournissent un cadre formel à la démonstration sans pour autant en fournir un support. Les démonstrations y sont faites à la main et une preuve est validée lorsqu'elle convainc les évaluateurs.
- Le deuxième niveau : Les méthodes de ce niveau utilisent des languages de spécifications formelles avec support et outils comme les model checkers et les vérifieurs de types. Les preuves peuvent être menées manuellement mais le langage naturel possible au niveau précédent disparaît.
- Le troisième niveau : Ce niveau correspond à des systèmes qui utilisent des languages de spécification dans le cadre de méthodes intégrant la formulation d'une démonstration en leur cadre : les systèmes de preuves (des prouveurs, des vérifieurs de preuves ou des assistants de preuves)

L'interaction des méthodes formelles avec la mise à jour dynamique des programmes a donné lieu à des travaux de recherches issus de différents domaines d'applications et utilisant diverses techniques et approches formelles. Nous allons donner dans ce qui suit, un aperçu sur les techniques formelles. Ensuite, nous proposons une classification des travaux de recherche selon les paradigmes des approches formelles utilisées. Une approche formelle pouvant utiliser différentes techniques.

III.2.3 Techniques formelles dans la DSU

III.2.3.1 La preuve de théorèmes

Cette technique [74] consiste à montrer par un raisonnement mathématique qu'un programme est conforme au comportement induit par sa spécification. La spécification est faite dans une logique de premier ordre ou d'ordre supérieur. Le comportement d'un dispositif peut se modéliser à l'aide de prédicats. La spécification et l'implémentation sont toutes deux décrites dans la même logique. Ceci permet d'associer immédiatement à cette description une interprétation fondée sur la sémantique de la logique. Cependant, une description dans une logique demande de la part de l'utilisateur une très grande discipline, et une compréhension très claire de son système. Cette technique utilise des système de preuves : des prouveurs de théorèmes, des vérifieurs de preuves ou des assistants de preuves. Un système de preuve dans une logique donnée est un ensemble d'axiomes et de règles d'inférence. Les axiomes sont des formules de la logique et sont élémentaires dans le sens où ils représentent les propriétés de base des opérateurs de la logique. Les règles d'inférence permettent d'établir des propriétés en utilisant les axiomes et les règles logiques du système. Parmi les méthodes utilisant cette techniques : Coq, HOL.

III.2.3.2 La vérification de modèles

La vérification de modèles ou model checking [75] est une technique qui repose sur la modélisation du système à étudier sous la forme d'un automate qui décrit quels sont les états possibles du système, ses évolutions (transitions entre états) et les propriétés atomiques que vérifie chaque état. La spécification est donnée sous forme de propriétés (éventuellement temporelle) du système. L'outil se base sur des algorithmes qui permettent de vérifier que le modèle satisfait sa spécification. Le model checking couvre l'ensemble de tous les états du système, ce qui en fait une technique de vérification formelle : toutes les exécutions du système sont considérées. Quand la propriété n'est

pas vérifiée, l'outil vérifieur de modèles montre un contre exemple. Parmi les outils du model checking, nous citons : SPIN, DIVINE.

III.2.3.3 L'analyse statique des programmes

L'analyse statique [76] consiste à analyser le texte d'un programme pour en extraire de l'information. Cette analyse est effectuée sans exécuter le programme dans le but d'analyser toutes les propriétés observables sur celui-ci. Le type d'informations extraites ainsi que les techniques choisies dépendent de l'utilisation à laquelle l'analyse est destinée. L'analyse statique est utilisée pour repérer des erreurs de programmation ou de conception, mais aussi pour déterminer la facilité ou la difficulté à maintenir le code. Ces informations sont utilisées par exemple pour transformer le code d'un programme, corriger des erreurs, ou analyser les performances. L'analyse statique représente la pièce maîtresse de quelques algorithmes de vérification. Il existe plusieurs formes d'analyse statique par exemple, l'analyse des flots de données, analyse du typage et l'analyse à base de contraintes.

III.2.3.4 L'annotation des programmes

Cette technique consiste à insérer dans le code source des énoncés logiques décrivant son comportement : les annotations. Il s'agit du modèle de la logique de Hoare [77]. L'annotation d'une fonction ou d'un fragment de code indique ses invariants en cours d'exécution, sa précondition et sa postcondition. La précondition représente les conditions préalables à l'exécution du code. La postcondition représente l'état à la fin de l'exécution. Ces spécifications sont écrites souvent dans un langage basé sur la logique du premier ordre et dont la sémantique est totalement liée au langage de programmation avec lequel est écrit le code qu'on veut prouver. Les programmes et les annotations sont traités par des outils qui génèrent des conditions de vérification qu'il faut prouver pour se convaincre de la correction du programme. Dans ce cadre, plusieurs outils ont été développés, comme le système d'annotation JML (Java Modeling Language) [78] pour les programmes Java.

III.2.3.5 Le raffinement

Le raffinement [79] est issu des travaux d'axiomatisation des langages de programmation de Morgan et Dijkstra [80, 81]. Cette technique permet la synthèse pas-à-pas d'un programme à partir d'une spécification. Chaque degré de précision supplémentaire représente un choix d'implémentation, par exemple, le choix de l'algorithme implémentant une fonction, ou encore le choix de la représentation concrète d'un type de données. Chaque étape du raffinement engendre des formules logiques qu'il faut démontrer. Cela permet de garantir que l'étape n+1 satisfait bien ce qui est requis à l'étape n. Ceci permet d'obtenir un programme qui satisfait la spécification originale. Le principal système faisant appel au raffinement est la méthode B.

III.2.3.6 La réécriture

La réécriture [82] est une technique pour la spécification des actions à effectuer sur les objets et l'utilisation de règles de réécriture qui agissent sur l'état du système, lui-même défini dans une logique équationnelle sous-jacente à la logique de réécriture. Les systèmes de réécriture sont des ensembles de règles de réécriture. Les règles d'un système de réécriture spécifient explicitement la forme des objets réductibles et la façon dont ils sont réécrits. Dans la logique de réécriture, le calcul consiste en l'application répétée des règles d'un systems de réécriture à un terme jusqu'a ce qu'aucune règle ne puisse plus être appliquée : le terme est alors dit en forme normale. Cette technique permet de spécifier des systèmes, de prouver des propriétés du système à spécifier, et de raisonner sur ses changements. Un programme satisfait ses propriétés si elles sont déductibles par la spécification du programme en appliquant les règles de réécriture.

III.2.3.7 La bisimulation

La bisimulation [83] est une technique permettant de caractériser l'équivalence comportementale de process spécifiés sous la forme de systèmes d'états - transitions étiquetés (LTS : Labelled State Transition). Une relation binaire R, (pRq) sur le système est une bisimulation si :

- pour tout état p' tel que $p \rightarrow^u p'$, il existe q' tel que $q \rightarrow^u q'$ et $(p'Rq')$ et si
- pour chaque état q' tel que $q \rightarrow^u q'$, il existe p' tel que $p \rightarrow^u p'$ and $(q'Rp')$,

avec u représente une action qui permet de passer de p à p' (Resp. de q à q'). La bisimilarité est l'union de toutes les bisimulations. Cette technique offre le moyen de prouver l'equivalence des process de deux manières : établir sur les états la relation R et démontrer que c'est une bisimulation ou construire R pour établir la bisimulation entre deux process (par exemple, une spécification et une implémentation). La bisimulation est utilisée pour les systèmes concurrents et réactifs.

III.3 Les paradigmes de formalisation de la DSU

Dans le but d'assurer la correction de la mise à jour dynamique, les techniques formelles sont utilisées dans le cadre d'approches basées sur différents paradigmes. Un paradigme désigne le concept de base sur lequel se base une approche formelle. Celui-ci détermine les concepts de base et le style de formulation des spécifications des différentes parties : les programmes à mettre à jour, les modifications, les processus de mise à jour ainsi que les critères de correction. Les approches offrent des démarches utilisant différentes techniques pour établir la correction vis-à-vis des critères. Nous proposons de présenter les différents travaux de recherches relatifs à la correction formelle de la DSU en les classifiant selon les paradigmes utilisés.

III.3.1 Le paradigme algébrique

Le paradigme algébrique (ou par algèbre multi - sortée) représente un formalisme où les spécifications se présentent sous la forme d'une collection d'ensembles de données, d'opérations et d'axiomes. Les noms des ensembles des valeurs sont appelés sortes. Par exemple, la sorte nat des nombres naturels. A chaque ensemble de données correspond un ensemble d'opérations. Chaque opération est typée par la donnée de son domaine et de son co-domaine. Les opérations sont spécifiées par un ensemble d'axiomes. Les axiomes sont soit des équations, soit des axiomes conditionnels. L'ensemble des sortes et des noms d'opérations d'une spécification algébrique est appelé signature. Le raisonnement s'effectuent en utilisant des techniques telles que la réécriture ou le model checking.

III.3.1.1 Les travaux de Zhang et al.

Dans leurs travaux [49, 84, 85], les auteurs ont présenté un cadre algébrique pour la spécification et la vérification de systèmes de DSU basés sur le mécanisme POLUS [25]. L'idée principale de la formalisation est de représenter le système de DSU comme un système de réécriture sur lequel les propriétés d'intérêt sont vérifiées. La formalisation comporte trois parties :

- le programme sous forme de sortes (sorte pour les programmes, l'ensemble des fonctions et instructions utilisées, des expressions) et d'opérations (permettant des construire des programmes et des éléments de ceux ci) ;
- le mécanisme de mise à jour sous forme d'un système de réécriture ;
- le correctif qui contient les variables (resp. les fonctions) ajoutées et/ou supprimées, ainsi que les fonctions d'indirection et de synchronisation d'états.

Ces travaux s'intéressent aux propriétés communes telles que les propriétés suivantes : la sûreté de typage, l'absence d'arrêt brutal, la consistance ainsi que des propriétés spécifiques. Ces propriétés sont sous forme de prédicats déductibles à partir de la spécification du système en applicant les règles

de réécriture de celui-ci. Le processus de vérification est basé sur trois étapes : choisir une configuration initiale, formaliser les propriétés puis vérifier. La formalisation est effectué dans le cadre de la méthode CafeObj [86]. Cette méthode permet de spécifier d'abord un système sous formes de systèmes transitions. Ces systèmes de transitions sont ensuite adaptés pour être représentés dans un formalisme algébrique. Cette représentation engendre des formules qui sont vérifiées en utilisant la preuve de théorèmes ou le model checking.

III.3.1.2 Les travaux de Chen et al.

Dans leurs travaux [68], les auteurs présentent un cadre pour l'implémentation de la DSU pour des applications orientées services basées sur OSGi. Le mécanisme de mise à jour est formalisé en utilisant le langage de l'algèbre des process FSP (Finite State Process). Les opérateurs FSP utilisés pour la formalisation sont : l'opérateur de préfixe d'action, la récursion, le choix, la composition parallèle et la priorité.

Les auteurs définissent les concepts de base du système : les services, enregistrements des services et accès à ceux-ci ainsi que les liens pouvant exister entre eux. Les entités relatives au mécanisme de mise à jour sont : le client, le serveur, le gestionnaire de mise à jour, les fonctions de transfert d'états ainsi qu'un mécanisme de délégation (assurant l'indirection de l'ancienne version d'un service vers la nouvelle) et et les interfaces des services. Les interactions entre clients et serveurs sont ensuite modélisées en utilisant ces opérateurs. Les clients et les serveurs sont modélisés comme étant des process et leurs actions à l'aide d'opérateurs FSP. De même, la spécification du mécanisme de mise à jour définit les parties suivantes comme étant des process : l'ancienne version d'un service, la nouvelle version, l'interface, le gestionnaire de mise à jour, le mécanisme de délégation et les fonctions de transfert d'états. Les actions relatives au mécanisme sont définies avec les opérateurs FSP (lancement de la mise à jour, redirection des appels, obtenir l'état d'un service ...).

Une fois le modèle construit, les propriétés à vérifier (absence d'inter-blocage, correction du serveur et sûreté de typage) sont exprimées en utilisant le langage FSP puis vérifiées en utilisant le model checker LTSA (Labelled Transition System Analyzer).

III.3.2 Le paradigme fonctionnel et systèmes de types

Le paradigme fonctionnel propose un cadre de spécification dans lequel la notion de fonction est centrale. Il se rapproche donc des langages de programmation fonctionnels comme Ocaml. Le système modélisé est exprimé par une série de définitions de fonctions, éventuellement récursives, sans effets de bord. Les fonctions sont représentées dans des formalismes dérivés du lambda-calcul qui est le fondement de tous les formalismes fonctionnels. Le lambda-calcul peut être enrichi avec des types pour les variables et les fonctions et offre un modèle de spécification très robuste.

III.3.2.1 Les travaux de Bierman et al.

Dans leurs travaux [61], les auteurs proposent un cadre formel pour la spécification et le raisonnement sur la DSU en se basant sur le lambda-calcul. Les auteurs introduisent un *update-calculus*, avec une sémantique formelle précise vue comme une extension du lambda-calcul typé. L'utilisation de ce calcul offre la garantie d'un fondement rigoureux pour le raisonnement sur la DSU.

La syntaxe est inspirée du lambda calcul avec les extensions suivantes : une primitive appelée *update* qui permet de charger une nouvelle version d'un module, mise à jour basée sur la notion de modules.

Une sémantique opérationnelle définit un ensemble de règles de réduction pour l'évaluation des expressions du calcul dans un contexte donné. Un ensemble de règles de typage est également donné pour vérifier qu'une mise à jour n'altère pas le bon typage d'un programme. Le formalisme proposé

est simple, flexible et extensible. Parmi les extensions envisagées, nous citons la possibilité d'assurer la correction sémantique d'une mise à jour étudiée dans [60].

III.3.2.2 Les travaux de Stoyle et al.

Les auteurs dans [27, 53] proposent de développer une technique de DSU qui permet de calculer par une analyse statique les points où une mise à jour dynamique peut être effectuée.

Le cadre baptisé PROTEUS permet d'exprimer la mise à jour dynamique et raisonner sur la consistance des représentations dans la DSU pour des programmes C. En plus des constructeurs habituels pour les langages de programmation : déclarations de types, expressions (conditionnelle, référence, affectation, let..in etc), les valeurs, et les programmes, PROTEUS fournit des expressions spéciales pour la mise à jour : updater, abs t e et con t e pour désigner respectivement la mise à jour, l'utilisation abstraite et concrète d'un type de données. La propriété de base dans ce travail est la con-freeness. L'étude est effectuée sur le code dans le but de déterminer les points de mise à jour. Le principe est d'étiqueter le programme par des expressions *update* à chaque point ou une mise à jour est possible. Une expression *update* sera annotée avec les types qui ne doivent pas être modifiés par celle-ci.

Le travail se base sur une analyse statique. Cette partie a le but suivant : étant donné un programme PROTEUS, inférer tous les point ou une mise à jour peut être appliquée, c'est-à-dire des points con-free par rapport aux types du programme puis insérer des expressions update en ces points. Cette analyse est présentée sous forme d'un système d'inférence et de types. Sa correction est démontrée ensuite. Le système d'inférence permet de calculer la capacité (*capability*) qui représente les types pouvant être modifiés à chaque point d'un programme. Dans ce système, la notion de type est améliorée pour inclure la notion de capacité. Pour chaque expression e, une règle est donnée pour déterminer, étant donné un ensemble capacité en entrée, l'ensemble capacité après l'exécution de l'expression. Deux propriétés liées à l'analyse sont démontrées : 1)- une mise à jour valide (effectuée sous la précondition nécessaire) préserve le bon typage des programmes et 2)- l'application de la mise à jour sur les expressions du code, l'environnement des types et le tas préserve le bon typage des programmes.

Le mécanisme est étendu notamment dans [53] pour définir différents lambda calculs dans le but d'établir des propriétés telles que la consistence et l'absence d'arrêt brutal.

III.3.2.3 Les travaux de Anderson et al.

Dans [51, 67], les auteurs présentent un cadre pour la spécification et le raisonnement pour la DSU dans les programmes multi threads. Le cadre est basé sur un système de types étendu avec effets. Le principe de ce travail est que la sûreté de la mise à jour dépend des états du programme caractérisés par le code et les ressources partagées (appelés snapshots). Ce travail part donc de l'idée suivante : un accès à une ressource partagée peut modifier celle-ci et une mise à jour du code peut modifier l'effet du code.

Le problème considéré est le suivant : étant donné un snapshot d'un système et des mises à jour dynamiques, est ce que le système modifié aura l'effet désiré et respectera la sûreté de typage ? L'approche proposée va typer le snapshot à chaque mise à jour introduite en traçant l'effet des mises à jour sur les expressions. Ceci est basé sur la proposition d'un système de type et d'effet pour garder trace des effets causés par la mise à jour sur les expressions, et pour garder trace de la différence entre l'effet de l'expression mise à jour et l'effet souhaité. Le language considéré est un lambda-calcul avec récursivité et threads. Les propriétés établies sont la sûreté de typage, la consistence de la mise à jour et l'absence d'inter-blocage.

Le principe est utilisé dans [50] pour établir la correction de la mise à jour dans un contexte MPI (Message Passing Interface) et dans [55] par l'extension d'un système de type avec des effets

permettant d'exprimer les effet d'un calcul déjà effectué (effet antérieur) et les effets d'un calcul à venir (effet futur). Ces effets sont utilisés pour assurer la consistence dans les programmes multi threads.

III.3.2.4 Les travaux de Hashimoto

Dans [63], l'auteur présente une méthode pour assurer la correction sémantique basée sur la définition d'un ensemble de points sûrs. La formalisation est basée sur un langage de premier ordre. Dans ce langage, un programme est représenté par une lambda-expression. A partir de cette représentation, un arbre libellé est construit pour le programme. Une notion de correspondance (code mapping) est introduite pour identifier des points de correspondance entre un programme P et sa nouvelle version $P1$ dans le but d'extraire les noeuds representant des modifications (code inséré, modifié ou supprimé) dans l'arbre représentant P.

Les effets de la mise à jour sont tracés en définissant un modèle exact qui utilise les flots d'informations et les dépendances extraites des arbres représentant les programmes.

Les informations sur les noeuds affectés par la mise à jour, les valeurs utilisées et les opérations de correspondance entre les états pour calculer des points sûrs de mise à jour. Un point sûr est tel que on est pas en présence avec du code mis à jour et les valeurs modifiées n'ont pas une utilisations critiques ultérieures.

Une approximation du modèle est construite par une interpretation abstraite de la sémantique pour déduire un ensemble de points sûrs réels. Le programme lui même est utilisé pour obtenir les transformateurs d'états en réutilisant les calculs du programme initial P.

III.3.2.5 Les travaux de Buisson et al.

Dans [57], les auteurs proposent un cadre formel (Pycots) basé sur l'assistant de preuve Coq pour établir la correction de la DSU pour un modèle par composants basé sur le langage python. Le cadre offre un modèle abstrait appelé Coqcots qui permet de démontrer des propriétés sur des architectures et leurs manipulations.

Ces modèles sont utilisés dans une approche permettant la construction de mises à jour correctes pour les composants. Premièrement, l'architecture actuelle du système cible est extraite en utilisant la plateforme Pycots. Le résultat est un module Coq contenant une architecture Coqcots. Le concepteur définit la configuration et construit la preuve de sa correction dans Coq. Ceci est basé sur la définition formelle de primitives de reconfiguration comme *create* pour ajouter un nouveau composant et *hotswap* pour modifier le comportement d'un composant existant. Ceci génère des preuves relatives aux opérations et à la préservation des contraintes d'architecture.

Un script de reconfiguration est alors extrait au moyen d'une extension au mécanisme d'extraction de base de Coq pour produire du code python. Le script est envoyé à la partie manager de Pytcots. A la réception de ce script, le manager l'applique au système cible.

III.3.3 Le paradigme états-transitions

Ce paradigme est représenté par les machines d'états abstraites (ASM : Abstract State Machine). Les ASM forment un paradigme basé sur les notions d'états et de transitions. Un système y est modélisé par un ensemble d'états et par un ensemble de règles de transition. Cet ensemble décrit sous quelles conditions (appelées gardes) un ensemble de transformations ont lieu permettant la transition d'un état à un autre. Parmi les outils issus des ASM, citons le langage de programmation ASMGopher qui étend le langage de programmation fonctionnel Gopher.

III.3.3.1 Les travaux de Hayden et al.

Dans[48, 28], les auteurs proposent un cadre formel pour la vérification de la DSU pour des programmes C. Ils proposent d'abord une classification des spécifications des mises à jour en trois catégories :

- Les spécifications compatibles avec l'ancienne version (Backward Compatible Specification) : La mise à jour dynamique modifie les propriétés d'un programme mais peut également en préserver quelques unes. Cette classe exprime les propriétés qui sont préservées par la mise à jour dynamique.
- Les spécification post mise à jour (Post-Update Specification) : Dans cette catégorie, les auteurs classent les spécifications qui expriment l'ajout de nouvelles propriétés aux programmes sans modifier celles qui existent déjà.
- Les spécifications nécessitant adaptation (Conformable Specification) : Dans cette catégorie, on classe les spécifications apportant de nouvelles propriétés qui affectent les propriétés déjà existantes mais d'une façon systématique. Elle exprime le fait que les propriétés existantes doivent être rendues conformes au nouveau code. L'exemple donné pour une mise à jour de ce genre est de rajouter un paramètre (espace nom) à une fonction qui permet de retourner une liste de couples (clé, valeur). L'application d'une telle mise à jour implique que pour les autres valeurs, il faut ajouter une valeur par défaut aux paires existantes.

Les spécifications sont écrites dans le même langage que le programme à mettre à jour. La vérification qu'un programme, une fois modifié, satisfait les propriétés requises, est réalisée selon une démarche appelée fusion de programmes (*program merging*). Elle consiste à transformer l'ancienne version du programme et un correctif en un seul programme. Ce dernier est démontré équivalent au code original plus le correctif. La transformation est réalisée en implémentant une fonction de transformation, pour le langage CIL (C Intermediate Language : une représentation plus simple de C), prenant comme argument l'ancien code et le patch et produisant un programme. Le programme argument est un triplet <code, tas, expression en cours d'exécution>. Le patch contient le nouveau code, un nouveau tas (à concaténer avec l'ancien) et une fonction de transformation d'états.

La transformation est formalisée et prouvée correcte. L'équivalence entre le programme obtenu et les données (code initial plus patch) est prouvée en énonçant un théorème appelé théorème de bisimulation. Ce théorème établit d'abord que le programme obtenu par la fusion simule chaque étape d'exécution dans l'ancien et le nouveau code ainsi que la mise à jour de l'ancien vers le nouveau. Ensuite, pour chaque trace dans le programme obtenu par la fusion il y a une trace qui lui correspond dans l'ancien programme.

III.3.3.2 Les travaux de Wu et al.

Dans [71], les auteurs présentent une machine d'états abstraite pour modéliser la mise à jour dans les systèmes orientés services basés sur les OSGIs. Le formalisme permet la vérification des systèmes pour déterminer s'ils satisfont les demandes de mise à jour pour fournir les principales fonctionnalités. Il fournit également une base pour spécifier de nouveaux composants et contraintes de mises à jour et raisonner sur les propriétés de correction. La formalisation est basée sur un modèle appelé machine d'état abstraite distribuée, un modèle de base pour spécifier des systèmes concurrents et réactifs. Les différents threads de l'application spécifiée sont modélisés par un ensemble d'agents. Deux types d'agents sont définis : les agents fonctionnels responsables des tâches de l'application et un agent coordinateur responsable de la mise à jour.

Le modèle abstrait général représente les fonctionnalités du système et des spécifications de la mise à jour par la définition d'un modèle pour chaque agent. Un ensemble d'états est défini pour les caractériser. L'agent coordinateur est représenté par les états suivants : Initial, en attente de mise à jour, en préparation de mise à jour, en gestion de mise à jour et l'état final. L'agent fonctionnel

est caractérisé par les états suivants : initial, pré-mise à jour, post-mise à jour et final. Chaque état correspond à un comportement spécifique. Le passage d'un état à un autre est spécifié par un ensemble de conditions.

L'analyse formelle est basée sur les concepts d'univers, de signatures et le raffinement du modèle de base. La notion d'univers représente les notions de base pour l'application et la mise à jour (par exemple, les versions, les noms des services, les contraintes des versions ...). Les signatures représentent les actions relatives au système (par exemple, obtenir la version d'un service, vérifier qu'un numéro de version est conforme aux contraintes, indiquer si un service dépend d'un autre ...). L'ensemble des états et des règles régissant les modèles en précisant comment les changements d'états s'effectuent est spécifié en utilisant les différents notions déclarées. Les raffinements spécifient des règles permettant de représenter des propriétés garantissant la correction de la mise à jour. Ces propriétés portent par exemple sur le moment de la mise à jour, l'ordre dans lequel sont effectuées les mises à jour, les différentes dépendances entre les composants et la compatibilité des services. Les auteurs utilisent le vérifieur de modèles SML pour établir les propriétés.

III.3.4 Le paradigme axiomatique

Le paradigme axiomatique est orienté vers la spécification et la vérification des propriétés spécifiques. La base de ce paradigme est la logique de Hoare. La logique de Hoare [77] est une discipline qui consiste à annoter les programmes avec des formules logiques, appelées *assertions*, et à extraire d'autres formules logiques, appelées *obligations de preuve* à partir d'un tel programme annoté. La validité des obligations de preuve entraîne la correction du programme vis-à-vis des annotations : sa spécification formelle. Pour un programme (ou une partie d'un programme) S, on aura le triplet de Hoare suivant : {P} S {Q} où : S est l'action spécifiée ; P et Q sont des formules logiques portant sur les variables du programme et appelées respectivement précondition et postcondition telles que : la précondition indique les conditions logiques qui doivent être satisfaites avant l'exécution de l'action S et la postcondition indique les conditions logiques vérifiées après l'exécution de S.

III.3.4.1 Les travaux de Charlton et al.

Dans [60], les auteurs présentent une extension de la logique de Hoare pour raisonner sur la DSU. Le travail se base sur la définition d'un langage impératif avec des caractéristiques concernant l'allocation de la mémoire. Les auteurs montrent comment les mises à jour dynamiques peuvent être modélisées en utilisant un langage de programmation de haut niveau où les procédures peuvent être écrites sur le tas. Puis ils montrent comment ces mises à jour peuvent être prouvées correctes avec un calcul de Hoare qui permet de garder la trace des spécifications comportementales de telles procédures stockées.

Le code étudié concerne un serveur web. Le serveur web est un système d'un seul thread qui exécute dans une boucle infinie des requêtes d'événements à partir d'une file d'attente, les requêtes sont de type {*get*, *post*, *update*}. Initialement, les modules sont à la version 1, lorsqu'une mise à jour a lieu, les modules mis à jour sont ajoutés à l'ensemble actuel, et on leur attribue le prochain numéro de version. Ensuite, les auteurs définissent un langage d'assertions en enrichissant la logique de Hoare pour raisonner sur des programmes manipulant le tas en permettant de spécifier le comportement du code de manière locale, de sorte que seulement la partie du tas qui est manipulée par le code est spécifiée. Le code supportant la mise à jour dynamique est spécifié avec des assertions sous la forme de triplet de Hoare. Ensuite, des obligations de preuve sont générées à partir du code spécifié, ces obligations de preuve sont démontrées dans l'outil Crowfoot.

III.3.5 Le paradigme à base de graphes

Cette catégorie concerne l'utilisation de concepts issus de la théorie des graphes. Les programmes ainsi que les processus de mise à jour sont modélisés sous forme de graphes. Une théorie est ensuite construite pour définir et établir les critères de correction.

III.3.5.1 Les travaux de Murarka et al.

Dans [14, 56], les auteurs proposent une modélisation permettant d'assurer la correction des mises à jour. Ce travail définit un critère pour la correction de l'exécution de requêtes simultanées à la mise à jour dynamique et proposent une approche de DSU pour garantir ce critère. Cette approche évite les deadlocks et garantit que le processus se trouve dans un état consistent durant et après la mise à jour. Ceci est assuré du fait que la mise à jour est exécutée à des point sûrs obtenus en analysant le correctif et les interdépendances existantes dans le code.

La définition formelle des critères repose d'abord sur la définition des changements qui peuvent être exécutés (classes ajoutées, supprimées, modifiées, inchangées). Une classification est établie pour distinguer les objets qui sont créés par d'anciennes versions mais qui peuvent être utilisés par le nouveau sans être modifiés, les nouveaux objets que l'ancien code ne peut plus utiliser et la définition de fonctions de transferts d'états. Sur la base de ces définitions, des conditions de corrections sont énoncées. Ces conditions doivent être satisfaites par les points de mise à jour.

Le calcul des points sûrs et des séquences de mise à jour est effectué en se basant sur différents types de graphes : le graphe de contrôle de flux (CFG : Control Flow Graph) et le graphe inter procédural (IFG : Inter procedural Flow Graph). Le CFG représente les flux dans les méthodes et les activités de l'application. Le IFG représente le flux de l'application. Ces graphes servent à calculer les points sûrs de mise à jour. Les auteurs utilisent ensuite un autre type de graphe pour représenter les dépendances inter threads (Parallel Execution Graph : PEG) dans le but d'éviter les inter-blocages qui peuvent potentiellement être introduits par les invocations des mises à jour. Un ensemble d'ordonnancement (de séquences) pour effectuer la mise à jour est ensuite calculé.

Ce style de modélisation est utilisé dans [66] dans le but de garantir la consistance. L'idée de base est d'isoler quelques méthodes du processus de mise à jour. Ces méthodes sont dites encapsulées et utilisées pour représenter des actions atomiques. Un graphe de dépendances des mises à jour est utilisé pour définir des séquences de mise à jour (ordre de mise à jour des classes) permettant de préserver cette isolation et donc la consistance. La même méthode est utilisée dans [11] pour établir la consistance.

III.3.5.2 Les travaux de Zhang et Huang

Dans [22], les auteurs présentent une approche basée sur le mécanisme des transactions pour la mise à jour dynamique des programmes Java en montrant certaines propriétés (ACID) permettant d'établir la démonstration formelle de la sûreté de typage.

Les auteurs présentent d'abord des définitions formelles pour les concepts de base : les méthodes, les champs, les dépendances et l'héritage entre les classes, les dépendances relatives aux méthodes et aux champs. Lors de l'exécution, ces relations entre classes se traduisent par des relations entre les objets. Ces relations sont modélisées sous forme de graphe. Il s'agit donc d'identifier les classes affectées par la mise à jour d'une classe C pour garantir l'absence d'erreurs de typage. L'ensemble des mises à jour d'une classe est donc modélisé de telle sorte à inclure d'abord la classe à mettre à jour elle même, puis ajouter des classes selon les différents cas qui se présentent. Par exemple, si des méthodes ou des champs sont supprimés alors inclure dans cet ensemble toutes les classes invoquant les éléments supprimés. La correction de la mise à jour dynamique repose sur la notion de transaction définie par un ensemble de mises à jour de classes à exécuter sans interruption.

Les auteurs proposent ensuite une approche de mise à jour dynamique qui se base sur l'ancienne version et la nouvelle version et sur une extension du *classloader* Java. La JVM permet à l'utilisateur d'écrire son propre *classloader* de telle sorte à supporter la mise à jour dynamique ainsi que des programmes écrits de telle sorte à les mettre à jour. Cette approche est utilisée également dans [21].

Propriétés	Formalismes	Techniques
Sûreté de typage	- **Algébrique :** [49, 68]. - **Fonctionel :** [2, 38, 4, 37, 51, 61, 53, 50]. - **Systèmes de types :** [21, 23, 15, 4, 53, 61, 42].	- **Analyse statique de mise à jour :** [15, 53, 61]. - **Analyse de typage de sessions :** [51, 50]. - **Preuve de théorèmes :** [37]. - **Analyse statique et model cheching :** [38]. - **Réécriture :** [49]. - **Model checking :** [68].
Consistance	- **Algébrique :** [49]. - **États-transitions :** [70]. - **Fonctionnel :** [2, 38, 53, 63, 57, 55]. - **Graphe de dépendances :** [11].	- **Réécriture :** [49]. - **Analyse statique et vérifieur de consistance :** [11, 38, 55]. - **Analyse de capacité de mise à jour :** [53]. - **Analyse statique :** [56, 66]. - **Interprétation abstraite :** [63]. - **Preuves de théorèmes :** [57].
Absence d'arrêt brutal	- **Algébrique :** [49]. - **Fonctionnel :** [53].	- **Réécriture :** [49].
Propriétés spécifiques	- **Algébrique :** [49, 84, 68, 85]. - **Fonctionnel :** [63, 28]. - **Axiomatique :** [60]. - **États-transitions :** [48, 64, 70, 71, 59]. - **Systèmes de types avec effets :** [67].	- **Bisimulation :** [48, 28]. - **Annotation :** [60, 28]. - **Analyse statique :** [64, 67]. - **Raffinement** : [71]. - **Interprétation abstraite :** [63]. - **model checking :** [84, 85, 68, 59, 49].
Con-freeness	- **Fonctionnel :** [27].	- **Analyse statique de capacité de mise à jour :** [27, 34].
Atteignabilité	- **États-transitions :** [3, 87, 47].	- **Analyse statique :** [3, 87, 47].
Absence d'inter-blocage	- **modélisation en graphes :** [14, 56]. - **Fonctionnel :** [51, 50].	- **Analyse de typage de sessions :** [51, 50]. - **Analyse statique :** [14, 56].
sûreté d'activation	- **États-transitions :** [28]. - **Fonctionnel :** [57].	- **Analyseur de dépendances :** [34]. - **Analyse statique :** [34]. - **Preuve de théorèmes :** [57].
Garantie de mise à jour	- **États-transitions :** [59].	- **Model checking :** [59].

Tableau III.2 : Paradigmes formels et techniques pour les critères de correction

III.4 Conclusion

Nous nous sommes intéressés dans ce chapitre à l'utilisation des méthodes formelles pour établir la correction des mises à jour dynamiques. Le concept de correction n'étant pas basé sur une définition unique, ce chapitre présente d'abord les différents critères de corrections de la mise à jour dynamique. La présentation de ces critères met en évidence les différents styles et domaines d'applications où ces critères ont été établis. Ensuite, un aperçu sur les principales techniques utilisées pour la correction de la DSU a été introduit. Nous avons proposé une classification des travaux de recherche relatifs à la correction des systèmes de DSU selon les paradigmes sur lesquels reposent les approches formelles utilisées. La description de chaque travail fait ressortir les concepts de bases utilisés par l'approches, les critères établis et les techniques utilisées.

Nous avons résumé la classification des travaux de recherches selon les paradigmes formels dans le Tableau III.2. Ce tableau permet de :

- Mettre en evidence certains faits comme par exemple que certains formalismes sont exclusivement utilisés pour un type de critère (le formalisme axiomatique est utilisé exclusivement pour la correction du comportement des programmes mis à jour). Il montre également que plusieurs méthodes permettent d'établir le même critère. Par exemple, la consistance est un critère qui a été établi en utilisant différents formalismes (algébrique, fonctionnel, les graphes et les systèmes de transitions).
- Observer d'autres catégorisations pour les critères de correction. Nous pouvons par exemple, classer ces critères selon qu'ils sont établis à un niveau abstrait (ou de conception) ou bien à un niveau code. Les critères tels que la sûreté du typage ou bien la correction du comportement sont relatifs au niveau code. Les critères se situant au niveau conception sont par exemple l'absence de deadlock ou bien la garantie de mise à jour. Ces critères sont relatifs à l'ordre dans les opérations de mise à jour ou bien à l'interaction du programme avec son environnement.
- Le tableau peut suggérer des choix quant à l'approche formelle à utiliser pour un critère donné ou bien le passage à un niveau de rigueur supérieur dans la formalisation et la vérification.

L'étude de l'état de l'art montre l'importance de l'utilisation des méthodes formelles pour établir la correction de la mise à jour dynamique. La diversité des approches montre aussi qu'il n' y' a pas de meilleur formalisme pour établir la correction de la DSU. Nous concluons également que nous ne disposons pas d'un formalisme global permettant de spécifier les systèmes de DSU, leurs propriétés et d'effectuer la vérification. La contribution à la correction d'un système de DSU nécessite donc la proposition de démarches, de formalisations et le choix d'outils pour modéliser le fonctionnement du système, spécifier les critères de correction et les établir.

Deuxième partie

DSU dans Java Card

Chapitre IV

Les cartes à puce et la technologie Java Card

Sommaire

Les cartes à puce jouent aujourd'hui un rôle crucial dans de nombreuses applications que tout un chacun utilise de façon quotidienne. Ceci tient au fait que les cartes à puce se sont imposées comme le moyen privilégié pour assurer la confidentialité et l'intégrité de données personnelles et/ou secrètes. L'évolution des différents champs d'application, tels que le paiement par carte bancaire, la communication par téléphonie mobile, la validation de titre de transport, etc, a généré un déploiement massif de ces cartes. La technologie Java Card fournit une plateforme pour cartes à puce articulée autour du langage Java. Les caractéristiques de cette plateforme, telles que son indépendance du matériel, la sécurité et la facilité de programmation, lui ont permis de s'imposer comme étant la technologie la plus déployée dans le milieu bancaire ou dans la téléphonie mobile. Dans ce chapitre, nous présentons un aperçu sur les cartes à puce, leurs caractéristiques, classifications et utilisations. Nous présentons ensuite la plateforme Java Card à travers ses caractéristiques, son architecture et les bases de son fonctionnement. Nous terminons par la présentation de quelques travaux relatifs l'application des méthodes formelles à la plateforme Java Card.

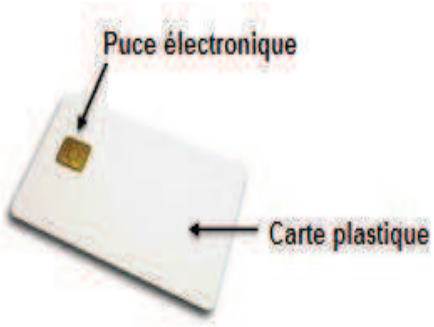

Figure IV.1 : Une carte à puce

IV.1 Les cartes à puce

Une carte à puce (voir la Figure IV.1) est un équipement se présentant sous forme de carte plastique de petite taille, intégrant un circuit électronique qui permet de stocker des données et d'effectuer des opérations de façon sécurisée. C'est donc un ordinateur à part entière possédant sa propre puissance de calcul et sa propre capacité de stockage d'informations, et étant capable, dans ses dernières générations, d'exécuter du code et d'héberger diverses applications. Elle est aussi qualifiée de carte intelligente (smart card en anglais), en opposée à la carte à mémoire qui contient des fusibles destructibles électriquement au fur et à mesure de leur utilisation. Tout ce qui concerne la carte à puce a fait l'objet d'une normalisation, auprès de l'Organisme International de Normalisation (ISO), portant le nom de ISO 7816 (voir [88]).

IV.1.1 Typologie des cartes à puces

Il existe plusieurs types de cartes à puce que'on peut classer soit suivant les technologies utilisées en interne, soit suivant le mode de communication de ces cartes avec le monde extérieur [89]. La Figure IV.2 [90] illustre ces classifications.

IV.1.1.1 Selon le mode de communication

a) La carte avec contact

Pour communiquer avec le monde extérieur, la carte à contact doit être insérée dans un lecteur (appelé aussi CAD pour Card Acceptance Device) afin d'utiliser les points de contact présents sur sa surface. Les contacts sont au nombre de huit (C1-C8). Les caractéristiques sont définies dans la norme ISO 7816-2. La Figure IV.3 illustre les contacts tels que :

- C1 : Vcc, utilisé pour la tension d'alimentation positive de la carte ;
- C2 : RST, utilisé pour la commande de remise à zéro ;
- C3 : CLK, horloge fournie à la carte ;
- C5 : GND, pour la masse électrique ;
- C6 : Vss : n'est plus utilisé, sert à l'USB actuellement ;
- C7 : I/O pour les entrées/sorties des données ;
- C4 et C8 : RFU, initialement réservés pour utilisation future, actuellement ils servent à communiquer en USB.

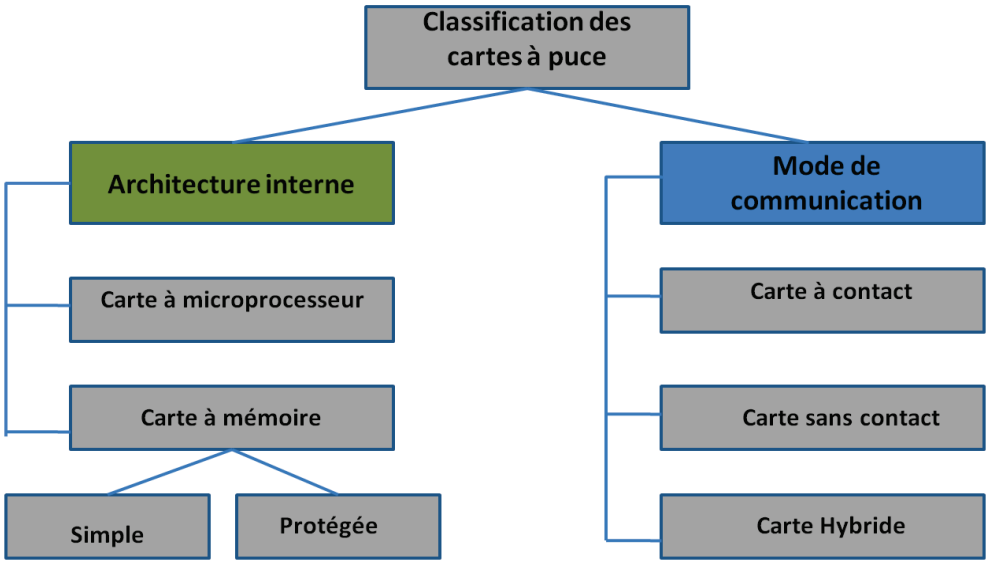

Figure IV.2 : Typologie des cartes à puce

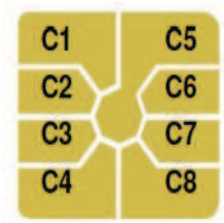

Figure IV.3 : Numérotation et positions des contact selon la norme ISO 7816-2

b) La carte sans contact

Dans ce type de cartes, la communication s'établit à travers une interface radio fonctionnant par induction grâce à une antenne imprimée ou intégrée dans la carte à puce (Figure IV.4). Cette carte n'utilise pas de contact physique avec le lecteur mais doit être assez prés de celui-ci, entre 3 et 10 centimètres. Les cartes sans contact sont privilégiées dans le domaine du transport ainsi que pour le contrôle d'accès aux locaux, domaines dans lesquels les transactions doivent être faites à une vitesse assez élevée.

c) La carte hybride

Les technologies étant différentes, une puce ne peut pas être à la fois avec et sans contact. De ce fait, les cartes hybrides embarquent deux puces : la première est reliée aux contacts et la deuxième à l'antenne. Ces cartes relient les avantages des deux types, cependant, leur prix est plus élevé.

IV.1.1.2 Selon la technologie interne

a) La carte à mémoire

Premiers modèles de cartes à puce, elles représentaient la majorité des cartes vendues dans le monde en 1999. Comme son nom l'indique, une telle carte possède une puce mémoire. Elle peut

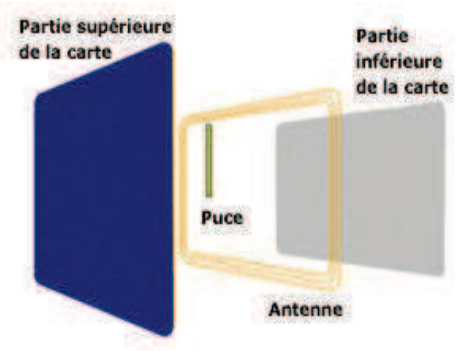

Figure IV.4 : Carte à puce sans contact

aussi comporter une logique câblée non programmable (instruction programmée et non reprogrammable) mais pas de microprocesseur. Autrefois, la taille de sa mémoire était seulement de quelques Kilo-octets alors qu'aujourd'hui certaines peuvent atteindre le Méga-octets. Les avantages de cette carte sont sa technologie simple et son faible coût de revient. Ses principaux inconvénients sont sa dépendance vis-à-vis du lecteur de carte pour le calcul et la possibilité d'être assez facile à dupliquer. Cette catégorie est scindée en deux sous-catégories :

- *La carte à mémoire simple :* Elle ne contient qu'une zone mémoire et un minimum de logique pour pouvoir y accéder via des signaux électriques.
- *La carte à mémoire protégée :* Elle associe de la mémoire, dont certaines zones sont accessibles seulement en lecture ou seulement en phase de personnalisation de la carte, de la logique permettant l'exécution d'automates simples jusqu'à la presentation de mots de passe ou autres succédanés de codes PIN. Il s'agit typiquement des télécartes contenant un compteur d'unités, un numéro de série et une donnée secrète permettant d'authentifier la carte.

b) La carte à microprocesseur

Ce type de carte constitue ce que l'on appelle une smart card. En effet, la puce de cette carte embarque un microprocesseur, ce qui la rend autonome (donc intelligente). Les dimensions de la puce sont au maximum de 25 mm2 pour sa surface et de 200 micromètres pour son épaisseur. C'est l'association en un seul circuit (voir Figure IV.5) de :

- un microprocesseur de 8, 16 ou 32 bits avec des architectures CISC (Complex Instruction Set Computer) ou RISC (Reduced Instruction Set Computer) travaillant à des fréquences internes allant de 5 à 40 MHZ.
- une mémoire morte (ROM) programmée en usine de façon figée. Son contenu constitué du système d'exploitation ainsi que les données permanentes n'est pas modifiable. Sa taille va de 1 Ko jusqu'à 24 Ko (pour les cartes haut de gamme).
- une mémoire EEPROM dont le contenu peut être modifiable. Elle contient les données qui peuvent évoluer dans le temps sans être perdues. Sa taille varie de 16 Ko à 125 Ko.
- une interface d'entrée/sortie série pour les échanges de données.
- de la logique nécessaire au fonctionnement.
- la puce possède éventuellement un coprocesseur cryptographique qui réalise les opérations de chiffrement et déchiffrement de façon matérielle ce qui permet d'améliorer les performances. Ce coprocesseur cryptographique est associé à un générateur de nombres aléatoires RNG (Random Number Generator).

Dans toute la suite du document, le terme *carte à puce* se référera à la carte à microprocesseur.

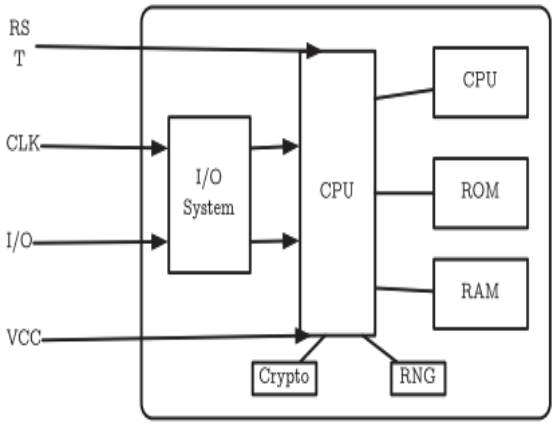

Figure IV.5 : Architecture simplifiée d'une carte à microprocesseur

IV.1.2 Communication de la carte avec son environnement

Pour que la carte à puce communique avec le monde extérieur, elle doit être placée dans un lecteur ou à proximité de ce dernier. Le lecteur est connecté lui même à un autre ordinateur (hôte) fournissant ainsi l'énergie nécessaire au fonctionnement de la carte. La carte communique avec le lecteur en mode semi-duplex, c'est-à-dire qu'à un instant donné, seule une des deux parties peut utiliser la ligne de communication : soit le lecteur, soit la carte, mais jamais les deux en même temps. Les données circulant entre la carte et l'hôte sont des paquets appelés APDUs (Application Protocol Data Units) définis dans la norme ISO 7816-4. Les échanges se font selon un modèle client/serveur où la carte joue le rôle du serveur : elle n'initialise pas la communication mais se contente de répondre aux commandes envoyées par l'hôte.

c) La pile protocolaire

La communication client/serveur se fait au travers d'une pile protocolaire de trois couches, chacune correspond un à protocole de communication [91] :

- **La couche physique :** Elle est normalisée par l'ISO 7816-3. Elle décrit les caractéristiques physiques comme la fréquence d'horloge (entre 1 MHz et 5 MHz), la vitesse des communications (jusqu'à 115200 bauds), etc.
- **La couche transport :** Elle est aussi normalisée dans l'ISO 7816-3. Elle définit les échanges de données pour les protocoles de transmission (TPDU : Transmission Protocol Data Unit) les plus couramment utilisés. Parmi eux, deux sont particulièrement utilisés : le premier appelé T=0 est un protocole orienté octet et le second appelé T=1 est un protocole orienté paquet. C'est cette couche qui définit également le format de l'ATR (Answer To Reset) qui est le message envoyé par la carte au lecteur juste après sa réinitialisation. Il contient la description des paramètres de transmission comme le protocole de transport supporté par la carte et ses paramètres matériels.
- **La couche applicative :** Elle définit le protocole utilisé pour l'échange des données au format APDU. Elle est normalisée dans l'ISO 7816-4.

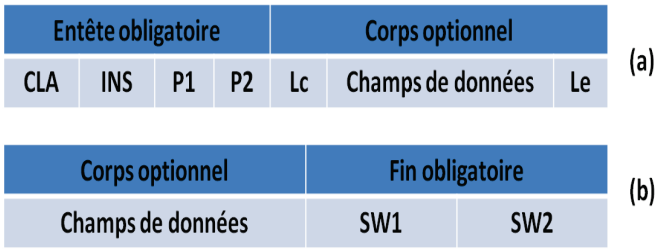

Figure IV.6 : Commande APDU (a) et réponse APDU (b)

d) Les commandes et réponses APDU

Il existe deux types d'APDU : les commandes APDU et les réponses APDU [91] [92]. La carte est toujours en attente d'une commande APDU et une réponse APDU aura toujours lieu en retour d'une commande APDU. Les deux types de structures de communication commande et réponse vont donc de paire pour chaque échange.

- **Commande APDU :** Envoyée par l'hôte vers la carte, sa structure comprend un entête et un corps (voir Figure IV.6, partie (a)). L'entête obligatoire contient quatre champs correspondant à un octet ayant une signification précise :
  - **CLA**, l'octet de classe qui identifie la catégorie de la commande APDU correspondant généralement à un domaine d'application donné (exemple, CLA = A0h pour les commandes des cartes SIM) ;
  - **INS**, l'octet d'instructions qui indique l'instruction à exécuter ;
  - **P1** et **P2**, deux octets contenant les paramètres de l'instruction.

  Le corps de la commande est variable et peut être omis :
  - **Lc** est l'octet qui spécifie la taille du champ de données ;
  - Le **Champ de données** qui contient les données à envoyer à la carte pour exécuter l'instruction spécifiée dans l'entête
  - **Le**, l'octet correspondant au nombre d'octets attendu par l'hôte pour le champ de données de la réponse APDU de la carte.
- **Réponse APDU :** Envoyée par la carte vers l'hôte, sa structure comprend (voir Figure IV.6, partie (b)) :
  - Un corps optionnel de taille variable : Le **champ de données** de taille **Le** déterminé dans la commande APDU correspondante ;
  - Une zone de fin obligatoire comportant deux octets : **SW1** et **SW2** (Status Word). Ces deux champs d'un octet précisent l'état de la carte après l'exécution de la commande APDU. Par exemple, 0x9000 signifie que l'exécution s'est déroulée jusqu'à son terme et avec succès.

IV.1.3 Les cartes à puces multi-applicatives

Une carte multi-applicative est une carte capable d'embarquer plusieurs applications. Ces applications ne s'exécutent pas de façon concurrente car les systèmes d'exploitation des cartes à puce possèdent un unique thread. Dans cette section, nous présentons les cartes multi-applicatives les plus connues exceptée la Java Card qui sera présentée en détails dans la prochaine section. L'architecture d'une carte multi-applicative (Figure IV.7) consiste en [93] :

- un système d'exploitation embarqué qui supporte le chargement et l'exécution de plusieurs applications. Ce système d'exploitation est en fait constitué d'un environnement d'exécution et d'une machine virtuelle qui mettent en place des caractéristiques sécuritaires (par exemple,

un pare-feu entre les applications).
– des applications qui sont interprétées par la machine virtuelle.

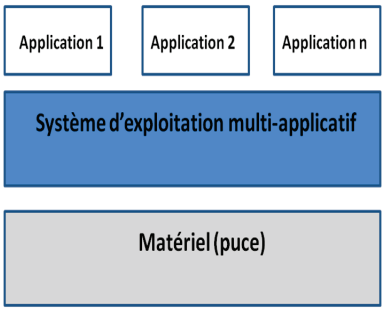

Figure IV.7 : Architecture générale des cartes multi-applicatives

IV.1.3.1 MULTOS

MULTOS [94] [93] est la première carte à puce ouverte, hautement sécurisée et possédant un système d'exploitation multi-applicatif. Elle est promue par le consortium industriel MAOSCO dont les fondateurs sont : American Express, Dai Nippon Printing, Mondex International Ltd, Siemens, Fujitsu, Hitachi, Motorola, MasterCard International et Keycorp. Les éléments clés de la plate-forme MULTOS sont :

– une architecture hautement sécurisée ;
– la possibilité d'avoir plusieurs applications sur la même carte ;
– l'indépendance des applications par rapport à la plateforme matérielle ;
– le chargement et l'effacement des applications durant la vie de la carte ;
– la compatibilité avec les standards industriels ISO 7816 et EMV (Europay, MasterCard et Visa).

Pour le développement des applications, Mondex International Ltd a mis en place des spécifications fixes pour les APIs MULTOS et un langage optimisé pour les cartes à puce : MEL (Multos Executable Language). Il est cependant possible de programmer des applications dans différents langages : C, Java, basic ou en assembleur, et de les traduire ensuite en MEL. A l'installation d'une nouvelle application, la carte à puce MULTOS vérifie la validité de l'application qui a été envoyée, alloue au programme un espace mémoire protégé grâce au pare-feu. Chaque nouveau service ou application reste aussi rigoureusement séparé des autres programmes déjà sur la carte afin qu'aucun d'eux, en cas de dysfonctionnement, ne puisse interférer avec un autre programme. Les développeurs de cartes MULTOS ne sont cependant pas libres d'ajouter leur propres extensions pour des raisons d'inter-opérabilité.

IV.1.3.2 Basic Card

Même si les cartes Basic Card [95] [93] existent depuis 1996 c'est seulement en avril 2004 que ZeitControl a sorti sa première carte multi-applicative, la MultiApplication BasicCard ZC6.5. Les cartes Basic Card de ZeitControl possèdent une machine virtuelle. Celle-ci se différencie des autres en étant la seule à supporter les nombres flottants. Cette carte se programme en ZeitControl Basic qui contient la plupart des constructions habituelles du langage Basic.

En intégrant diverses solutions cryptographiques elles offrent un niveau élevé de sécurité. L'outil de développement nécessaire et complet pour la réalisations des applications pour ces cartes est gratuit. Pour les personnes souhaitant s'initier au monde des cartes à puce, il a l'avantage de ne

nécessiter aucun matériel pour commencer à développer (ni lecteur, ni cartes). En outre, l'énorme avantage de ces cartes est leur très faible coût.

IV.1.4 Les applications des cartes à puces

La carte à puce a désormais pris une grande place dans notre vie quotidienne. Elle est en effet utilisée dans de nombreux domaines parmi lesquels nous citons :

- l'industrie des télécommunications avec, par exemple, les cartes téléphoniques prépayées, cartes USIM insérées dans les téléphones GSM, 3G ou 3G+ ;
- l'industrie bancaire et monétaire avec les cartes de crédit et les porte-monnaies électroniques ;
- les applications d'identification avec les cartes d'identité, les passeports, les permis de conduire ainsi que les contrôles d'accès au locaux et le contrôle horaire ;
- l'industrie audiovisuelle dans le cadre de la télévision à péage ;
- l'industrie du transport avec les cartes sans contact pour les transports en commun ou pour les transports routiers ;
- l'authentification (sur des sites internet) ;
- les applications de marketing comme les cartes de fidélité ;
- émergence d'autres applications de la carte à puce notamment grâce à internet et à la multiplication des e-services qui exigent de plus en plus une identification et une sécurisation des transactions. Ainsi, la carte à puce peut être utilisée comme terminal de stockage de clefs servant à l'authentification et à la sécurisation des communications dans le cadre du e-commerce, banque à distance, courrier électronique, télétravail, etc.

IV.2 La technologie Java Card

A ses débuts, la programmation des cartes à puce était réalisée en assembleur et en C, et longuement vérifiée. Le programme contenait à la fois le système opératoire et l'application. Le processus de développement était donc long, ceci est 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.

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 MULTOS, Basic Card ou ou bien Java Card font partie de cette nouvelle génération.

IV.2.1 Présentation et bref historique

La plateforme Java Card a pour but de faire fonctionner la technologie Java sur des équipements fortement contraints tels que les cartes à puce ou d'autres équipements avec peu de mémoire et peu de puissance de calcul. Une Java Card est donc une carte à puce sur laquelle on est capable de charger et d'exécuter des applications Java du type applets ou servlets (depuis la version 3.0).

C'est en 1996 qu' un groupe d'ingénieurs de Shlumberger à Austin au Texas cherchent à simplifier le modèle de programmation existant pour cartes à puce tout en préservant sa sécurité. Le langage de programmation Java apparaît comme la solution. Schlumberger devint alors la première entreprise à acquérir une licence en proposant une spécification de quatre pages (la spécification Java Card 1.0). En Février 1997, Bull et Gemplus se joignent à Schlumberger pour cofonder le Java Card Forum. Le but de ce consortium industriel était d'identifier et de résoudre les problèmes de la technologie Java

Card en proposant des spécifications à JavaSoft (la division de Sun microsystems à qui appartient Java Card). Il a également pour objectif de promouvoir des APIs Java Card afin de permettre son adoption par l'industrie de la carte à puce.

En Novembre 1997, Sun microsystems présente les spécifications Java Card 2.0 qui consistent en un sous-ensemble du langage et de la machine virtuelle Java. Elles définissent des concepts de programmation et des APIs très différentes de celles de la version 1.0. En Mars 1999 sort la version 2.1 des spécifications Java Card. Elle se compose de trois spécifications :

- la Java Card 2.1 Virtual Machine (JCVM) specification [96]. C'est une spécification qui définit la machine virtuelle Java Card ainsi que le langage de programmation pour les cartes à puce.
- la Java Card 2.1 Application Programming Interface (API) specification [97]. C'est une spécification de l'ensemble des paquetages et des classes Java nécessaires pour la programmation des cartes à puce.
- la Java Card 2.1 Runtime Environment (JCRE) specification [98]. C'est une spécification qui décrit le comportement de l'exécution du système comme les gestion des applets sur la plateforme ou les différentes notions relatives à l'instanciation des objets.

Durant presque une décennie, la Java Card 2 connut plusieurs évolutions des versions qui s'est déclinée à travers l'évolution des trois entités de spécification précitées. Ces évolutions furent relatives notamment à l'invocation de méthodes à distance (RMI : Remote Method Invocation) et les algorithmes cryptographiques.

C'est en 2008 que la spécification Java Card 3.0 fut publiée. Cette version apporta une révolution dans les spécifications, car elle introduit deux types de spécification Java Card :

- Java Card 3 *Classic Edition* qui est juste une évolution de la Java Card 2.2.2 faite pour fonctionner avec des cartes ayant de faibles ressources. Elle contient les fonctionnalités nécessaires aux supports des applets.
- Java Card 3 *Connected Edition* qui introduit de vraies nouveautés, car en plus des applets classiques, elle supporte un nouveau type d'application : les servlets. Les servlets sont des applications web qui nécessitent d'avoir un serveur web embarqué dans la carte. Elle nécessite donc des cartes avec des contraintes de ressources moins fortes (cartes modernes haut de gamme)

IV.2.2 Les avantages de Java Card

La plateforme Java Card offre aux développeurs d'applications pour cartes à puces les avantages suivants [99] [90] :

1. *Facilité de programmation.* L'utilisation de Java comme langage évolué au lieu des langages assembleurs rend le développement plus aisé et à la portée d'un plus grand nombre de développeurs. Cette facilité est due à :
   - la programmation orientée objet offerte par Java (contre l'assembleur auparavant) ;
   - une plateforme ouverte qui définit des interfaces de programmation (APIs) et un environnement d'exécution standardisé ;
   - l'utilisation d'une machine virtuelle permettant la portabilité des applications et une sécurité accrue lors de l'exécution du code ;
   - une plateforme qui encapsule la complexité sous-jacente (assembleur) et les détails du système des cartes à puce.
2. *Indépendance du matériel.* Tout comme Java, la plateforme Java Card assure l'indépendance du code par rapport au matériel sur lequel il s'exécute. Cette portabilité permet d'écrire du code qui fonctionnera sur n'importe quel microprocesseur de carte à puce.
3. *Sécurité.* C'est le facteur primordial pour les cartes à puces. Une partie de cette sécurité est offerte par le langage Java (un langage fortement typé, plusieurs niveaux de contrôle d'accès

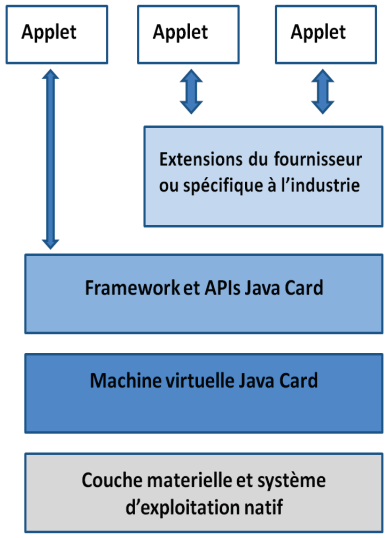

Figure IV.8 : Architecture de Java Card 3 *Classic Edition*

aux méthodes et aux variables, etc) alors que l'autre partie de cette sécurité est assurée par la plateforme Java Card elle même en implémentant plusieurs mécanismes tels que le pare-feu qui isole les applications hébergées dans la carte pour empêcher tout comportement hostile de leur part.

IV.2.3 Architecture de la plateforme Java Card

L'architecture de la plateforme Java Card se décline donc en deux éditions : l'édition classique (Java Card *Classic Edition*) et l'édition connectée (Java Card *Connected Edition*).

IV.2.3.1 La Java Card 3 *Classic Edition*

Elle est basée sur la version 2.2.2 [100][101][102] de la plateforme Java Card. Elle cible les cartes à faibles ressources qui supportent uniquement les applets comme type d'applications. L'édition classique adopte la même architecture que les versions précédentes. Les changements apportés par cette version portent essentiellement sur le support des derniers algorithmes cryptographiques et les standards régissant les communications sans contact. Son architecture (Figure IV.8 [91]) est articulée autour d'une machine virtuelle Java Card [103], d'un environnement d'exécution [104], et d'un ensemble de librairies accessibles par des Application Programmer Interface (API)s [105].

IV.2.3.2 La Java Card 3 *Connected Edition*

L'édition connectée de la plateforme introduit dans son architecture (Figure IV.9) [91] une nouvelle machine virtuelle ainsi qu'un nouvel environnement d'exécution qui supporte maintenant trois modèles d'application en opposition à l'unique modèle proposé par l'édition classique des précédentes versions de la plateforme. Elle est pensée pour les cartes de nouvelle génération avec plus de ressources. Ces trois modèles d'applications sont :

– Le modèle d'application utilisant les applets classiques basé sur celui hérité des précédentes versions de Java Card et à ce titre a les mêmes caractéristiques. Ces applications utilisent le modèle de communication avec la carte basé sur le protocole APDU.

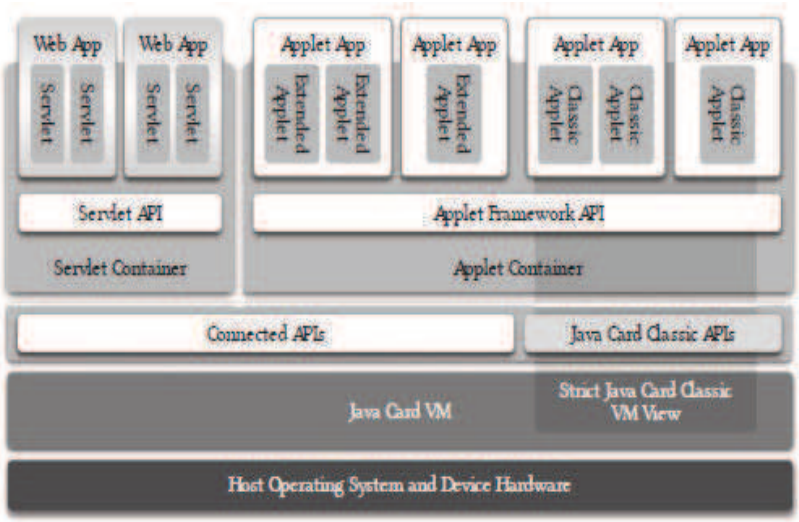

Figure IV.9 : Architecture de Java Card 3 *Connected Edition*

- le modèle d'application utilisant les applets étendues est aussi basé sur celui des applets, mais avec des améliorations parmi lesquelles on peut citer entre autre la gestion du multithreading, du format de fichier class, la gestion des chaînes de caractères, etc. Elle utilise aussi le protocole de communication APDU.
- le modèle basé sur les applications web utilise le modèle d'application basé sur la construction de servlets. Il hérite des nouvelles APIs introduites avec cette édition de la plateforme. Et elle utilise le protocole HTTPS pour effectuer les communications avec la carte.

Cette édition est spécifiée par les quatres entités suivantes :

- la Java Card Virtual Machine specification, Connected Edition [106].
- la Java Card Application Programming Interface specification, Connected Edition [107].
- la Java Card Runtime Environment specification, Connected Edition [108].
- Java Card Servlet specification, connected Edition [109]

IV.2.4 Le langage Java Card

La quantité de ressources (mémoire et processeur) disponibles sur la carte étant faible, la plateforme Java Card ne contient qu'un sous-ensemble de Java. Cette section explique les différences qui existent avec Java [90] [91].

a) Les éléments non supportés

Dans le cas de **l'édition classique**, les éléments de Java non supportés sont :

- le chargement dynamique de classes : les classes sont masquées dans la carte au moment de la fabrication ou téléchargées dans la carte après son émission. Ainsi, les programmes s'exécutant sur la carte doivent uniquement faire appel aux classes déjà présentes dans celle-ci car il n'existe aucun mécanisme permettant de télécharger des applications pendant l'exécution ;
- le gestionnaire de sécurité ou Security Manager : il est directement implémenté dans la machine virtuelle. La classe Java.lang.SecurityManager n'existe pas au contraire de Java où elle fait office de gestionnaire de sécurité ;
- la finalisation : il n'y a aucune invocation automatique de la méthode Finalize () ;

- les threads : la classe Threads n'est pas supportée ainsi qu'aucun des mots clefs liés à la gestion des threads ;
- le clonage d'objet : la classe Object n'implémente pas la méthode clone () et il n'y a pas d'interface cloneable ;
- le contrôle d'accès : le contrôle d'accès existe bien tout comme en Java par contre avec quelques restrictions ;
- les types énumérés : pas de types énumérés, ni de mot clef enum ;
- les arguments de taille variable : parce qu'elle nécessite que le compilateur génère un code qui crée un nouveau tableau d'objets à chaque fois qu'un tel argument est invoqué, ceci cause une allocation de mémoire implicite dans le tas de la Java Card ;
- les annotations visibles à l'exécution : car l'introspection de classe n'est pas supportée ;
- les types char, double, float et long ainsi que les tableaux de plus d'une dimension ;
- de façon générale aucune des classes des APIs principales de Java n'est supportée entièrement. À titre d'exemple dans le paquetage Java.lang les classes supportées sont Object et Throwable ;
- le paquetage Java.lang.System n'est pas supporté. La plateforme fournit une classe Java Card. framework.JCSystem qui apporte une interface vers les fonctionnalités du système ;
- les fichiers class ne sont pas supportés. En effet, le format des applications supporté est le fichier CAP (Converted APplets) qui est un fichier class dont une partie des éditions de liens est déjà faite afin d'accélérer l'exécution.

Dans le cas de **l'édition connectée** et donc des applications web ainsi que des applets étendues, les éléments non supportés sont :

- pas de prise en charge des nombres à virgules flottantes c.-à-d. les types float et doubles, par contre, tous les autres types de Java sont supportés ;
- pas de classloader définit par l'utilisateur car toute machine virtuelle Java Card 3.0 doit utiliser les classloaders de la plateforme qui ne peuvent pas être modifiés. Cette fonctionnalité a été supprimée pour des raisons de sécurité ;
- pas de groupes de threads ou des threads fonctionnant comme des veilleurs. Les opérations sur les threads telles que le démarrage d'un thread ne s'applique qu'à des objets threads individuels. Si le développeur a besoin de démarrer un groupe de threads, il doit utiliser une collection d'objets contenant plusieurs objects threads ;
- pas de finalisation des instances de classes ;
- pas d'erreur ou d'exception asynchrone : en effet, la gestion des erreurs sur la plateforme est très limitée. Il y a une classe d'erreurs qui est définie dans la spécification ; en dehors de ces erreurs la machine virtuelle doit soit s'arrêter, soit renvoyer l'erreur la plus proche possible de la super classe de l'erreur représentant la condition d'erreur à lever.

b) Caractéristiques Java supportées.

Les caractéristiques Java supportées dans le cas de **l'édition classique** sont :

- les paquetages ;
- la création dynamique d'objets ;
- les méthodes virtuelles ;
- les interfaces ;
- les exceptions ;
- la généricité ;
- l'importation statique ;
- les annotations invisibles à l'exécution ;
- les types simples court : boolean, short, byte ;
- les classes Object et Throwable ;

– ainsi qu'en option le type int et le mécanisme de suppression d'objets.

En revanche, **l'édition connectée** supporte l'intégralité du langage Java :

– ainsi tous les types de données existants sont supportés en Java sauf les double et les float ;
– le multi-threading ;
– toutes les APIs Java ;
– le support en natif des fichiers class avec l'édition des liens qui se fait dans la carte.
– la destruction automatique des objets non utilisés ou on demand Garbage Collector.

IV.2.5 La machine virtuelle Java Card

La machine virtuelle Java Card est basée sur les notions de piles pour la manipulation des valeurs et de tas pour le stockage des objets. Elle comporte un contexte d'exécution par méthode appelée. Tout comme la machine virtuelle Java, elle exécute du code sous forme de codes octet, c'est-à-dire une représentation par des mnémoniques d'instructions élémentaires, avec éventuellement des opérandes. Ces codes octet, appelé bytecode, seront ensuite traduits dans le langage du micro-processeur de la carte à puce par le système d'exploitation, qui associe à chaque code octet un comportement calculatoire correspondant à sa sémantique.

IV.2.5.1 Le langage bytecode

Comme le langage Java, le langage Java Card est interprété. La compilation d'un programme source Java Card génère du code à interpréter (le langage bytecode). C'est un format intermédiaire entre le source et le binaire qui rend les programmes indépendants. A chaque méthode d'une classe donnée, une suite d'instructions bytecode est générée sous la forme d'une liste de couples (numéro, instruction bytecode) tel que le numéro corréspond à l'indice du premier octet de cette instruction dans le tableau d'octets contenant la liste d'instructions et une instruction est composée d'un code opération (opcode) représenté sur un octet et des opérandes représentés sur zéro ou plusieurs octets.

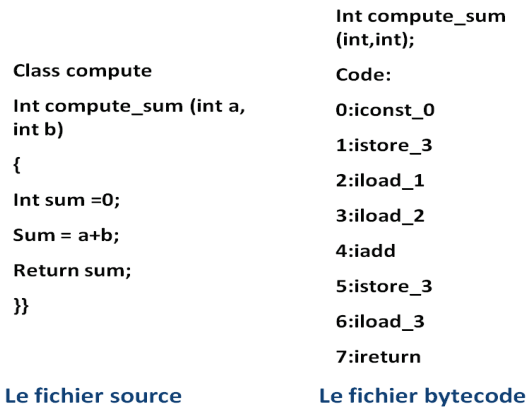

Figure IV.10 : Exemple d'un programme en bytecode

La Figure IV.10 montre un programme en bytecode. Les codes octet dans le langage Java Card sont au nombre de 186. Ce jeu d'instructions est classifié selon les types d'opérations dans les catégories suivantes :

– instructions de chargement et de sauvegarde, comme *iload*, *istore* et *iconst* ;
– opérations arithmétiques, par exemple : *iadd* et *ineg* ;
– opérations logiques comme *iand* et *ior* ;
– instructions de conversion de types, par exemple : *i2b*, *i2s* ;

– instructions de création et de manipulation d'objets, par exemple : *new*, *getfield*, *putfield* ;
– instructions de manipulation directe des opérandes dans la pile comme *pop*, *dup* ;
– instruction de transfert de contrôle comme *ifeq*, *goto* ;
– instructions d'appels et de retours de méthodes comme *invokevirtual*, *ireturn* ;
– instructions pour les exceptions comme *athrow* ;

IV.2.5.2 Les structures de données d'exécution

L'exécution des programmes se base sur des structures de données permettant de stocker différents types d'informations tels que les objets créés, les valeurs de retour des méthodes, les variables locales des 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 Card.

– **Le tas de la machine virtuelle.** C'est une zone partagée par tous les threads de la machine virtuelle. Elle permet de stocker toutes les instances d'objets ou de tableaux. 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.
– **Les cadres d'activation des méthodes (frames).** 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, et la pile d'opérandes. Cette pile sert lors des appels de méthodes pour passer les arguments et recevoir le résultat. Certaines instructions bytecode prennent des valeurs de cette pile, opèrent sur ces valeurs, et mettent le résultat de l'opération sur la pile. Elle sert aussi pour les calculs intermédiaires.
– **La pile Java.** A chaque thread exécuté est associé une pile Java (stack en anglais). La pile Java d'un thread est constituée par les appels de méthodes. 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. La pile d'exécution mémorise les variables locales et les paramètres mais aussi d'autres informations utiles dans le contexte d'une méthode : valeur de valeur de retour et point de retour dans la méthode appelante.
– **La zone de méthodes.** 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 la table des méthodes de la classe, la table des champs et la table des constantes.

IV.3 Java Card et les méthodes formelles

IV.3.1 Vérification formelle du typage du bytecode

La vérification du bytecode est une composante cruciale dans la sécurité des applications Java et Java Card : une erreur lors de la vérification causant l'acceptation d'un programme mal typé peut ouvrir des brèches de sécurité. D'un autre coté, la vérification est un processus complexe impliquant des analyses élaborées des programmes. Par conséquent, plusieurs recherches se sont intéressées à la spécification et la vérification du typage, la formalisation des algorithmes de vérification et la démonstration leurs corrections [110, 111].

IV.3.1.1 Les travaux de Freund et Mitchell

Dans leurs travaux [112, 113, 114], les auteurs s'intéressent à la vérification du bytecode Java. En effet la JVM dispose d'un vérifieur permettant de s'assurer de certaines propriétés comme : la validité des opcodes, des sauts, des signatures des méthodes etc. Les problèmes traités sont relatifs à la création et l'initialisation des objets ainsi que les routines.

La première étape consiste à définir un sous ensemble pour les instructions d'un sous langage bytecode qui prend en considération : les classes et les interfaces ; les constructeurs et les fonctions d'initialisation d'objets ; l'invocation de méthodes ; les tableaux ainsi que les exceptions et les sous-routines. La formalisation est donnée d'abord sous la forme d'une sémantique opérationnelle montrant comment l'exécution de chaque instruction fait passer la machine d'un état à un autre. Un état est représenté par le compteur ordinal du programme, la pile d'opérandes, les variables utilisées ainsi que le tas de la machine virtuelle. Une sémantique statique est ensuite donnée pour les instructions. Elle se présente sous la forme d'un système de typage qui spécifie les conditions pour que chaque instruction soit correctement typée. Une instruction est bien typée si elle a le bon nombre et le bon type d'arguments. Ces informations sont vérifiées par le typage des variables et du contenu de la pile d'opérandes. Un programme est considéré correctement typé si chaque instruction du programme est correctement typé.

La formalisation proposée est utilisée d'abord pour le problème de l'initialisation des objets. En effet, il est important de vérifier qu'aucun objet n'est utilisé avant d'être initialisé. Pour ce faire, l'accent est mis sur les règles sémantiques des instructions permettant la créations des objets (*new*) et l'initialisation des objets (*init*). Les informations issues des deux aspects (statique et opérationnel) sont combinées pour garantir la correction de l'initialisation des objets dans java selon le système proposé. La correspondance entre les deux aspects est exprimée par des prédicats qui permettent de déterminer si un état est bien formé. Ces prédicats expriment principalement la correspondances entre les structures de données dans la sémantique opérationnelle et leurs types dans la sémantique statique. Le cadre formel proposé est étendu pour étudier d'autres caractéristiques comme le problème des sous routines Java, l'utilisation des moniteurs et l'étude d'implémentations existantes pour la vérification de bytecode.

IV.3.1.2 Les travaux de Dufay et al.

Dans [115, 116], les auteurs proposent d'apporter une vérification formelle des principaux composants de la plate-forme Java Card ainsi qu'une méthodologie et des outils pour aide à la vérification. La formalisation est effectuée dans l'assistant de preuves Coq. Premièrement, la machine virtuelle Java Card est formalisée en suivant les spécifications de Sun sur le comportement calculatoire et les vérifications à effectuer. La machine obtenue ainsi est dite *défensive*. Sa formalisation présente d'abord les types utilisés, la formalisation des méthodes, des classes et des interfaces ainsi que la notion d'état en utilisant principalement les structures et types inductifs de Coq. L'environnement d'exécution est ensuite formalisé en définissant les types des valeurs, les objets, les contextes d'exécution des méthodes puis la notion d'état. La sémantique des instructions bytecode de Java Card est représentée par des fonctions sur les états de la machine virtuelle.

Un ensemble d'outils appelé *JCVM tools* est utilisé pour permettre d'abord de traduire les programmes en Java Card vers des expressions Coq dans le but de les vérifier vis-à-vis de la spécification formelle de la machine virtuelle. A partir de la formalisation de la machine virtuelle, une version abstraite de celle-ci est ensuite déduite à l'aide de *Jakarta* : un outil permettant l'abstraction et le raffinement des spécifications. La machine virtuelle abstraite permet d'effectuer les tests sans les calcul et constitue la base du vérificateur du typage de bytecode.

IV.3.1.3 Les travaux de Requet et al.

Dans [117], les auteurs proposent une formalisation d'un vérifieur de bytecode en utilisant la technique PCC (Proof-Carrying Code). La technique PCC [118] est un mécanisme qui permet au consommateur du code de définir une politique de sécurité et de vérifier que cette politique est respectée par les programmes fournis par le producteur du code. Dans le PCC, une preuve de la validité du programme accompagne celui-ci. Un théorème est d'abord construit à partir du programme à vérifier. Le producteur fournit la démonstration de ce théorème. Une fois celle-ci achevée, le code est expédié avec sa preuve au consommateur. Quand celui-ci reçoit le code, il applique la phase de la génération du théorème et s'assure que la preuve accompagnant le code correspond bien à la démonstration du théorème généré. Pour appliquer ce mécanisme, le processus de vérification est divisé en deux parties : une partie en dehors de la carte (off-card) où des informations de vérification sont construite pour former un certificat. Ce certificat est envoyé avec le bytecode sur la carte où la deuxième partie de la vérification est effectuée sur le code reçu et le certificat. Les auteurs présentent une modélisation du vérifieur du bytecode à l'aide de la méthode B. Le système est basé sur FACADE : un langage intermédiaire typé spécialement conçu pour les plateformes à ressources limitées comme les cartes à puce. Un algorithme d'inférence de type est exécuté en dehors de la carte pour construire le certificat sous forme d'une table de types pour les instructions du code. La méthode B permet d'établir la sûreté de l'algorithme de vérification, ce qui signifie qu'aucun programme mal formé ne sera accepté par le système.

IV.3.2 Vérification formelle de la correction du comportement

Plusieurs travaux de recherche se sont intéressés à la vérification des programmes Java Card vis-à-vis des spécifications des utilisateurs. Ces travaux sont généralement basés sur l'annotation des programmes et la logique de Hoare [77].

IV.3.2.1 Vérification au niveau du code source

La vérification des programmes Java Card au niveau du code source a fait l'objet de plusieurs travaux. Dans [119], les auteurs s'intéressent à la vérification statique de programmes Java Card annotés formellement avec leurs spécifications, en utilisant l'outil Krakatoa construit au-dessus de l'outil Why (devenu maintenant la plateforme Why3 [120]). Le système est basé sur la génération d'obligations de preuve en implémentant un calcul de plus faible précondition [81]. Les auteurs ont particulièrement défini une sémantique et des annotations pour l'arrachage des cartes ainsi que les mécanismes de transactions.

Dans [121, 122], les auteurs développent un système basé sur une interprétation du langage Java Card dans la logique dynamique [123]. Le système, baptisé *Key*, prend en considération des propriété de sûreté et de sécurité sous forme d' invariants forts : les invariants de classes doivent être vérifiés en tout point du programme. De plus, l'outil permet de faciliter l'intégration des méthodes formelles dans la conception et l'implémentation des applications. Dans ce cadre, l'outil permet l'expression et la génération automatique de contraintes qui seront vérifiées par un système de preuve pour assurer la cohérence des applications.

IV.3.2.2 Vérification au niveau du bytecode

L'idée d'annoter des programmes sources par des spécifications et de calculer des obligations de preuves dont la démonstration garantit la correction du programme a été étendue aux programmes au niveau du bytecode dans certains travaux. Ces travaux reposent sur un calcul de plus faible précondition adapté aux instructions du bytecode. Dans [124], les auteurs proposent un calcul en utilisant les spécifications d'instructions pour bytecode Java. Ce travail est effectué dans le but

d'établir un compilateur de preuves. Il s'agit d'établir la preuve que les propriétés au niveau source restent valides au niveau bytecode. Les spécifications d'instructions permettent d'avoir toutes les propriétés que le programme doit satisfaire à un point donné et par la suite permette de vérifier les méthodes dans leurs intégralités vis-à-vis de leurs spécifications.

Le calcul de plus faible précondition est également utilisé dans [125] pour constituer la base d'un outil permettant de générer des obligations de preuve pour des programmes écrits en bytecode . Les spécifications sont écrites dans un langage appelé BML (Bytecode Modelling Language) en terme de préconditions, postconditions, spécifications intermédiaires et invariants de classes. Les fichiers classes sont étendus de manière à inclure les spécifications pour les utiliser dans une démarche de PCC dans Java Card.

IV.3.3 Vérification formelle de caractéristiques de la JCVM

Plusieurs travaux se sont intéressés à établir formellement d'autres caractéristiques de la JCVM, comme par exemple, la propriété d'isolation. La plateforme Java Card est ouverte et multi applicative. Les interactions entre les applications d'une même carte doivent être strictement contrôlées. Cette isolation entre les application est assurée par le par-feu (firewall). Dans [126], les auteurs modélisent la propriété d'isolation sous la forme de propriétés d'intégrité et de confidentialité des données des applications. Le mécanisme du par-feu est ensuite modélisé dans un assistant de preuves et la démonstration des propriétés est réalisée. Cette propriété est également établie dans [127] par la définition d'un système de typage permettant de vérifier l'isolation des applications.

Dans [128], les auteurs établissent la correction formelle de propriétés fonctionnelles et de sécurité pour un protocole RMI (Remote Method Invoke). Le travail est basé sur deux modèles : un modèle fonctionnel pour spécifier les conditions des appels des méthodes et leurs résultats et un modèle pour la représentation algorithmique des ces méthodes grâce à la formalisation d'un sous ensemble de Java dans un assistant de preuve. Les auteurs démontrent que les modèles algorithmiques sont des raffinements corrects des modèles fonctionnels. La même idée est ensuite utilisée par [129] pour la vérification de l'interface native de Java Card.

IV.4 Conclusion

Nous avons présenté dans ce chapitre, dans un premier lieu, un aperçu sur les cartes à puces. En raison de son utilisation massive et de la diversité des domaines dans lesquels elle intervient, la carte à puce fut l'objet d'avancées technologiques qui ont mené au développement de plateformes visant à développer des applications pour cartes de façon simple et sécurisée. La plateforme Java Card présentée dans la deuxième partie de ce chapitre est actuellement la plus utilisée grâce à de nombreuses caractéristiques (comme la portabilité et la simplicité). Le vif succès de cette plateforme à fait d'elle l'objet de nombreuses thématiques d'études (sécurité, attaques, vérification), nous avons donc présenté un aperçu sur les travaux relatifs à la vérification formelle de la plateforme Java Card. Cet aperçu nous permet de nous situer vis-à-vis des travaux de formalisation et de vérification existants en vue d'appliquer les méthodes formelles à la vérification d'un système de mise à jour dynamique dédié à cette plateforme. Ce système, appelé EmbedDSU, est présenté en détail dans le chapitre suivant.

Chapitre V

Le système de mise à jour dynamique EmbedDSU

Sommaire

$\mathcal{L}$'application de la mise à jour dynamique aux cartes à puces a donné lieu au développement du système EmbedDSU, premier système de mise à jour dynamique pour les applications Java Card. Ce système a été développé à l'université de Limoges au sein de l'équipe SSD (Smart Secure Device) du laboratoire Xlim dans le cadre des travaux d'Agnès Cristelle Noubissi. Ce chapitre est fortement basé sur le document de thèse [5] et les articles relatifs au système [58, 130, 131]. Nous présentons d'abords les objectifs du système puis nous détaillerons son architecture à travers les différents modules qui le composent, ainsi que son fonctionnement pour mettre en évidence le traitement des problèmes scientifiques de la mise à jour dynamique. Nous présentons à la fin de ce chapitre nos motivations concernant l'étude de ce système qui représente le point de départ des contributions de cette thèse.

V.1 Introduction au système EmbedDSU

V.1.1 Présentation et objectifs du système

EmbedDSU est un système de mise à jour dynamique pour des applications Java pour carte à puce. Il est basé sur la modification de la machine virtuelle Java Card. Afin de réduire la complexité de l'opération de mise à jour, EmbedDSU tend à limiter le processus aux parties affectées par la modification. Il est basé sur un ensemble de mécanismes 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.

Le développement du système EmbedDSU répond principalement aux deux besoins suivants [132] :

- **Besoin de sécurité :** Le système EmbedDSU est développé comme mesure proactive de sécurité. En effet, un dispositif classique pour limiter l'impact de nouvelles menaces est de renouveler régulièrement le parc de cartes en circulation (dispositif suivi par exemple pour les cartes bancaires). Cependant ce dispositif s'avère inefficace dans certains type d'applications où la durée de vie de la carte est relativement longue (par exemple, le passeport électronique a une durée de dix ans). Il est difficile d'avoir a priori des garanties fortes sur la résistance des dispositifs embarqués dans la carte sur une aussi longue période. La DSU représente un mécanisme pour renforcer les applications sur cartes en proposant d'ajouter de nouvelles fonctionnalités, corriger des erreurs ou appliquer des correctifs de sécurité en proposant des mécanismes qui permettent de réduire les inconvénients liées aux mécanismes classiques de mise à jour (perte de l'état d'exécution et cohérence de l'état interne). Le contexte carte à puce pose également la contrainte du retour des cartes : dans le cas des e-Passeports par exemple, si une faiblesse algorithmique est découverte, l'unique solution est de retourner tous les passeports à l'organisme émetteur et d'en obtenir des nouveaux. Ceci représente un coût considérable. Un système de DSU pour carte à puce permet donc de renforcer la sécurité tout en tenant compte des contraintes liées au contexte des cartes à puce et des applications qui les utilisent.
- **Contraintes sur les ressources :** La mise à jour par post-issuance peut poser un problème relatif à taille de transfert de l'applet complet. L'interface de communication entre la carte et son lecteur peut avoir une bande passante limitée, particulièrement dans le cas d'une carte sans contact dont la durée de communication est limitée. De plus l'envoi d'une nouvelle version d'une application pose un problème au niveau de la mémoire occupée par l'ancienne et la nouvelle version. Un renouvellement complet d'une application est donc difficile par rapport aux ressources des cartes à puce, de plus, si la mise à jour a une taille limitée par rapport à la taille de l'application (correction de bug par exemple), alors il serait intéressant d'avoir une solution alternative pour ne mettre à jour qu'un sous-ensemble de l'application. Le système EmbedDSU est conçu de telle manière à répondre à ce besoin.

V.1.2 Caractéristiques du système

L'approche d'EmbedDSU permet de mettre à jour les composants Java Card d'une façon dynamique et possède les propriétés suivantes :

1. *Support de plusieurs granularités internes de mise à jour :* EmbedDSU supporte plusieurs niveaux de granularité. Cette notion définit la taille du plus petit élément, de la plus grande finesse du système. Quand le niveau de granularité d'un système est achevé, il n'est plus

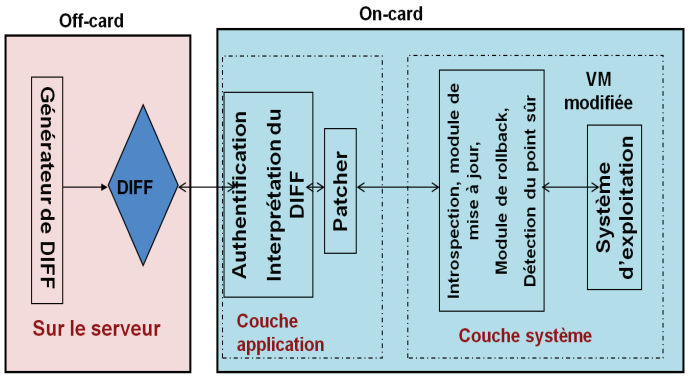

Figure V.1 : Architecture du système EmbedDSU

possible de découper l'information. La mise à jour dynamique peut être effectuée donc au niveau des champs de la classe, des signatures des méthodes, des blocs d'instructions et des classes relatives.

2. *Pas d'intervention humaine :* Le générateur de DIFF est capable de déterminer les différences syntaxiques entre deux versions d'une classe au niveau des interfaces, champs, signatures des méthodes (types des paramètres et types renvoyés) et instructions d'une classe d'une façon automatique, de plus, le processus de mise à jour à l'intérieur de la carte est totalement automatique et ne requiert pas une intervention humaine.
3. *Atomicité de la mise à jour :* La mise à jour dynamique se déroule d'une façon atomique afin d'assurer la cohérence du système. Dans le cas d'échec de la mise à jour, un mécanisme permettant une restauration de l'ancienne version est appliqué.

V.2 Architecture du système EmbedDSU

EmbedDSU est basé sur un ensemble de mécanismes à l'extérieur de la carte (off-card) et à l'intérieur de la carte (on-card) (voir Figure V.1).

La partie gauche de la figure comporte le générateur de DIFF qui prend en entrée deux versions d'une classe à mettre à jour : l'ancienne et la nouvelle. Son rôle est d'exprimer les différences syntaxiques entre ces versions dans un langage dédié. Le résultat est un fichier DIFF qui sera ensuite transmis à la partie on-card qui est la partie à droite de la figure. La partie on-card est représentée sur deux couches : la couche application et la couche système. La couche application comporte des modules permettant de vérifier la signature du fichier DIFF et l'initialisation des structures de données nécessaires à la mise à jour. La couche système consiste en la modification de la machine virtuelle en implémentant des modules permettant d'effectuer la mise à jour. L'ensemble des modules sera détaillé par les sections suivantes.

V.2.1 Architecture off-card

En off-card, l'objectif est de déterminer les différences entre l'ancienne version d'une classe et la nouvelle ainsi que d'analyser les fonctions de transfert fournies par le développeur dans le but d'extraire des informations qui seront stockées dans un fichier DIFF et ensuite transférées vers la carte.

L'architecture off-card d'EmbedDSU (Figure V.2) se décompose en deux sous-modules.

1. le module de génération des différences ;

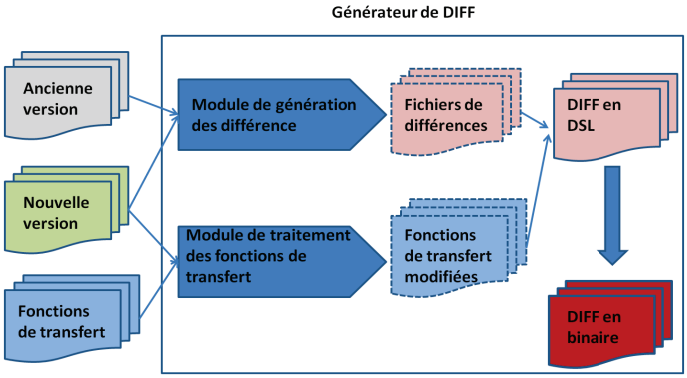

Figure V.2 : La partie off-card du système EmbedDSU

2. le module de traitement des fonctions de transfert.

Les différences entre les deux versions et les informations sur les fonctions de transfert obtenues sont exprimées dans un langage dédié (DSL : Domain Specific Language).

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 :

- *la 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.
- *la 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.

V.2.1.1 La génération du fichier DIFF

La génération du fichier DIFF est basée sur la notion de graphe de flot de contrôle (CFG pour Control Flow Graph) pour identifier les changements entre deux versions du fichier classe (*.class*) puis calculer les correctifs syntaxiques.

a) Présentation du concept de CFG

Un CFG 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 fonctions d'un programme. Il peut être modifié ou interrompu par divers types d'instructions pouvant entraîner la séparation des flots en zéro, un ou plusieurs chemins. Ces types d'instructions peuvent se subdiviser en plusieurs catégories : le branchement inconditionnel ou saut ; le branchement conditionnel ; le branchement conditionnel à choix multiples ; les boucles et l'arrêt inconditionnel. L'enchaînement d'instructions successives ne contenant aucune rupture du flot de contrôle et aucune cible d'instruction de branchement constitue un bloc élémentaire. Un CFG se compose d'un ensemble de sommets constitués par les blocs élémentaires et d'un ensemble d'arêtes 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 est présenté par la Figure V.3.

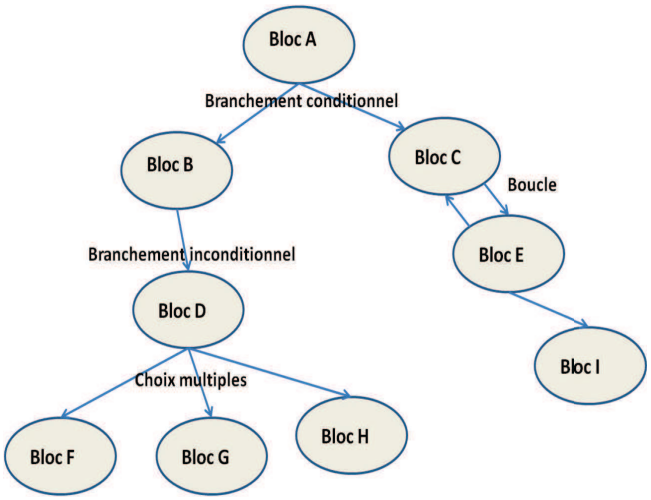

Figure V.3 : Illustration d'un CFG

b) Fonctionnement du générateur de DIFF

Le générateur de DIFF commence par créer une structure arborescente pour 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 associé. Une fois les CFG créés, le générateur de DIFF effectue les comparaisons à tous les niveaux :

1. Au niveau de la table des constantes : Il s'agit de déterminer les entrées de la table de constantes qui ont été affectées par la mise à jour en déterminant 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*. Les modifications détectées sont exprimées et rajoutées dans le fichier de DIFF.
3. Au niveau des champs de la 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 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. Ensuite, il compare les deux CFG associés pour déterminer les instructions ajoutées et supprimées et rajoute toutes ces informations dans le fichier de DIFF.

La règle appliquée au niveau des éléments de la classe (champs, méthodes, etc.) concernant l'ajout et la suppression est la suivante : les éléments apparaissant uniquement dans l'ancienne version sont considérés comme supprimés et ceux apparaissant uniquement dans les nouvelles versions sont considérés comme rajoutés.

V.2.1.2 L'expression du fichier DIFF

a) Le langage DSL

Les DSL (Domain Specific Languages) 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 de celui-ci. Les DSL contrairement aux langages généralistes sont conçus pour être utiles à une tache spécifique dans un domaine restreint. Dans la littérature, les DSL sont utilisés dans de nombreux domaines [133]. Pour le système EmbedDSU, un DSL capable de décrire les changements entre deux versions d'une classe en étant simple et concis a été conçu. Ce langage dédié à l'expression des différences est basé sur les types de modifications pouvant être effectuées dans un fichier Java Card notamment sur celles traitées par EmbedDSU.

b) Le fichier DIFF

Le fichier DIFF résultant est donc exprimé selon les règles syntaxique et de grammaire du DSL dédié [5]. Il contient les éléments suivants dans l'ordre :

- Une entête : 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 ;
- Les informations sur la table de constantes : Elles permettent d'exprimer les modifications effectuées au niveau des tables de constantes des deux versions d'une classe ;
- Les informations sur les méthodes : Cette partie décrit les modifications effectuées sur les entrées de la table des méthodes des classes ;
- Les informations sur les champs : Cette partie décrit les modifications détectées sur les tables de champs des deux versions d'une classe ;
- Les modifications des méthodes : Cette partie permet de décrire en détail les modifications détectées pour chaque méthode modifiée de la classe. Elle est représentée par une liste de méthodes. Chaque entrée de la liste est associée à une méthode et comporte une partie 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 bytecodes de la méthode ;
- Une entête de la fonction de transfert et les instructions de transfert d'état.

La Figure V.4 représente un petit exemple qui montre un code source où on modifie une instruction d'addition en soustraction. Ce changement se traduit en bytecode par le remplacement de l'instruction *iadd* (addition sur des entiers) en *isub* (soustraction sur des entiers). La modification est exprimée dans le fichier DIFF par l'action *Del % 4* qui renvoie à la suppression de l'instruction *iadd* qui se trouve initialement en ligne 4 (Del pour *Delete*). La deuxième action du fichier DIFF (*Add % isub 4*) exprime l'insertion de l'instruction *isub* à la ligne 4.

Avant l'envoi sur la carte le fichier DIFF est transformé en un DIFF binaire puis vérifié vis-à-vis du format spécifié : Il s'agit de la vérification du format du fichier ainsi que des instructions du fichier de DIFF concernant la validité des instructions (les opcodes et les arguments), la présence du nombre magique, les indexes des tables, etc.

V.2.2 Architecture de la partie on-card

Dans sa partie on-card (Figure V.5), le système EmbedDSU implémente les modules suivants :

- Module d'interprétation du fichier DIFF ;
- Module d'introspection ;

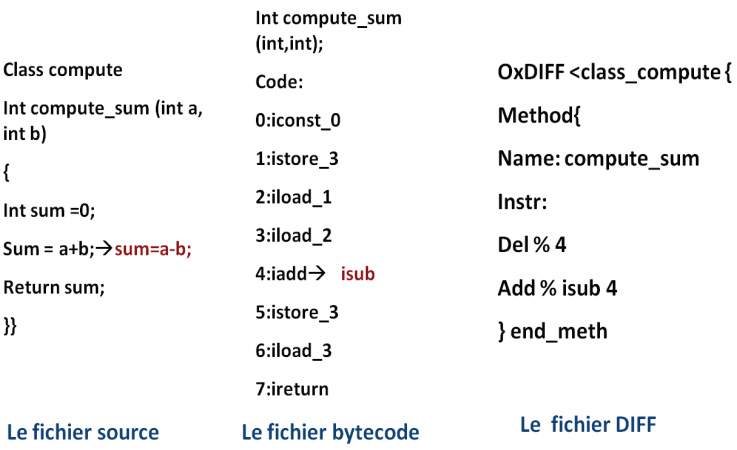

Figure V.4 : Exemple d'un fichier DIFF

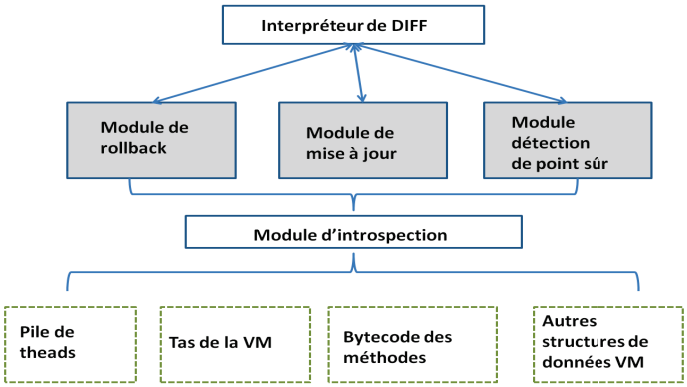

Figure V.5 : La partie on-card du système EmbedDSU

- Module de détection du point sûr ;
- Module du mise à jour : ce module prend en charge la mise à jour des composants suivants :
  1. la mise à jour du code ;
  2. la mise à jour des instances ;
  3. la mise à jour des frames ;
  4. la mise à jour des classes dépendantes ;
  5. la fonctionnalité de transfert d'état.
- Module de roll-Back.

V.2.2.1 Module d'interprétation du fichier DIFF

Ce module consiste en l'interprétation du fichier DIFF et à l'initialisation des structures de données afin d'extraire les informations nécessaires sur les champs et les méthodes qui seront ajoutés ou supprimés, ainsi que sur les fonctions de transfert. Cette interprétation permet aussi la restauration de l'ancienne version du système en cas d'échec du processus de la mise à jour dynamique.

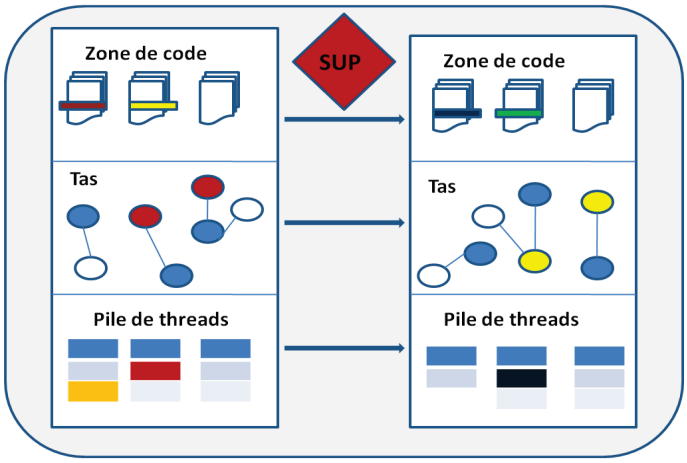

Figure V.6 : Les différentes parties mises à jour

V.2.2.2 Module d'introspection

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

- Déterminer les instances appartenant à la classe à mettre à jour et présentes dans le tas de la machine virtuelle ;
- Déterminer l'ensemble des frames associées aux méthodes restreintes ;
- Fournir la référence d'une frame appartenant à une méthode donnée si celle-ci est active dans la pile de threads.

V.2.2.3 Module de détection du point sûr

Ce module est chargé de rechercher un point sûr du système. Son objectif est de déterminer le moment opportun où la mise à jour peut être effectuée en se basant sur les informations contenues dans les structures de données de mise à jour. Ceci est fait en coopération avec le module d'introspection. Une fois un point sûr trouvé, le processus de mise à jour proprement dite peut commencer.

V.2.2.4 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 aux différents niveaux de 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 (voir Figure V.6), ainsi que la mise à jour des classes dépendantes, et la fonctionnalité de transfert d'état. Ce module contient plusieurs sous-modules relatifs à chacune de ces fonctionnalités.

1. **Mise à jour du code**
   A partir du bytecode contenu dans le fichier de DIFF, et des informations déterminées sur les méthodes qui ont été ajoutées, supprimées, ou modifiées, ce module procède à la mise à jour par un système de recopie en modification (expliqué dans la sous section suivante).
2. **Mise à jour des instances**
   Ce sous module a pour rôle de mettre à jour les instances et les références sur les structures de la machine virtuelle déterminées par le module d'introspection. La mise à jour des instances

s'appuie sur les informations des structures de données de mise à jour pour déterminer les champs à ajouter, supprimer, ou modifier. La mise à jour des instances est ensuite réalisée grâce au processus de recopie en modification.

3. **Mise à jour des frames**

   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.

   Ce sous-module permet de mettre à jour les informations dans les frames. Il s'agit notamment :

   (a) des références aux anciennes instances de la classe pour qu'elles pointent sur les nouvelles versions des instances ;

   (b) des adresses des méthodes non restreintes pour qu'elles pointent sur les nouvelles adresses de ces dernières.

4. **Mise à jour des classes dépendantes**

   Ce sous-module se base sur le module d'introspection pour déterminer les classes possédant des appels aux méthodes 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**

   L'initialisation des champs ajoutés ou modifiés est une étape nécessaire dans la mise à jour des instances afin d'éviter les erreurs. Le type d'initialisation est indiqué par le programmeur dans le fichier de DIFF. Après initialisation, le transfert d'état est effectué. Il existe deux types d'initialisations :

   (a) 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 ;

   (b) 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.

V.2.2.5 Module de roll-Back

Dans EmbedDSU, les références aux anciennes versions des instances et l'adresse de l'ancienne version de la classe en cours de mise à jour sont conservées dans les structures de données de mise à jour tant que le processus n'est pas terminé. Donc, en cas d'erreur, ou en cas d'opération illicite, ou d'échec de la mise à jour, ce module se base sur le module d'introspection et sur ces informations. Il permet de restaurer la version précédente du code, des instances, de la pile de thread et des structures de données de la machine virtuelle qui ont été modifiées. Ces informations sont récupérées pour remplacer les valeurs des nouvelles références et nouvelles adresses dans les structures de données de la machine virtuelle. Après le remplacement, les espaces mémoires préalablement alloués aux nouvelles versions sont libérés. Ceci permet, à la fin de la restauration du contexte, d'utiliser les anciens objets et l'ancien code de la classe.

V.3 Fonctionnement de EmbedDSU

V.3.1 La mise à jour du code

L'approche proposée pour la mise à jour du code se base sur les informations obtenues à partir du fichier de DIFF pour mettre à jour le bytecode 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. On peut donc définir les étapes de la mise à jour par :

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

La recopie en modification du code est basée sur les informations contenues dans les structures de données de mise à jour et sur les bytecodes des instructions ajoutées (s'il y en a) contenues dans le fichier de DIFF stocké en on-card. La recopie en modification du code d'une méthode donnée est illustré par une représentation algorithmique (Algorithme 1).

Algorithme 1 Recopie en modification du code

Entrée : Ancienne adresse de la méthode, taille de l'ancienne méthode, fichier de DIFF, structures de données de mise à jour
Pour un compteur de 1 à la taille de l'ancienne méthode **faire :**
  Vérifier les informations dans les informations de données de mise à jour
  Si le type de modification est l'ajout **alors :**
    recopier les instructions fournies dans le fichier de DIFF
  Sinon
    Si le type de modification est la suppression **alors :**
      aucune recopie n'est faite
    Sinon
      recopier l'instruction de l'ancienne version du code et mettre à jour
    Fin de si
  Fin de si
Fin de boucle pour

La mise à jour de la classe se fait grâce à la recopie en modification des bytecodes 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 des constantes est aussi mise à jour par recopie en modification grâce aux informations fournies par les structures de données de mise à jour. L'algorithme de mise à jour de la classe est basé donc sur les recopies en modification du code et des méta données. Après recopie en modification des métadonnées, des entêtes et bytecodes des méthodes, le processus, représenté par l'Algorithme 2, retourne l'adresse de la nouvelle version de la classe.

Algorithme 2 Mise à jour du code de la classe

Entrée : Ancienne adresse de la classe, fichier de DIFF, structure de données de mise à jour
Sortie : Nouvelle adresse de la classe
Mettre à jour les entrées de la table de constante et les autres métadonnées
Pour chaque méthode de la classe **faire :**
  recopie en modification de l'entête de la méthode,
  recopie en modification du bytecode de la méthode
Fin de boucle pour
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. Ceci est effectué en parcourant les classes de l'application. Si une référence à une méthode ou à un champ de la classe mise à jour est trouvée alors il faut mettre à jour la référence de la méthode ou du champ pour qu'elle corresponde à celle de la nouvelle version dans A.

V.3.2 La mise à jour des données

Il s'agit de la modification de toutes les données impactées par la mise à jour, relatives au contexte d'exécution de la classe à mettre à jour. 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 des threads ;
- des informations relatives au code de la classe et aux bytecodes de ses méthodes, et des autres structures de données de la machine virtuelle.

Toutefois, la mise à jour des données concerne principalement les instances dans le tas et les frames dans la pile de thread. Le processus de mise à jour des instances peut être subdivisé selon les étapes suivantes :

- 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.

a) La recherche des instances

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 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 des racines 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 machine virtuelle. 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.

b) Le transfert d'état

La mise à jour des instances est basée sur la technique de transfert d'état à partir des fonctions de transfert générées automatiquement et/ou celles 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. L'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.

La 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 DIFF. Ceci est effectué par une recopie en modification des instances (Algorithme 3).

Algorithme 3 Recopie en modification d'une instance

Entrée : Ancienne référence de l'instance, tas de la VM, structures de données de mise à jour
Sortie : Nouvelle référence de l'instance
Récupérer l'instance correspondant à la référence fournie
Pour chaque champ de l'instance **faire :**
  Si le champ est non modifié ou si uniquement la valeur du champ a changé **alors :**
    recopier en modification le champ concerné
  Sinon
    Si le champ est ajouté ou si le type du champ a changé **alors :**
      Ajouter un champ du type correspondant à la nouvelle instance en cours de création
    Sinon
      Si le champ est supprimé **alors :**
        aucune action de recopie n'est faite
      Fin de si
    Fin de si
  Fin de si
Fin de boucle pour

L'initialisation des nouvelles instances : 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.

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, il existe 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.

c) La 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. Ceci est réalisé en effectuant un parcours de la pile Java. Pour chaque pile de threads, une recherche des frames contenant des anciennes références des objets mis à jour est lancée. Les entrées des frames résultant de cette recherches et contenant les anciennes références sont ensuite modifiées en mettant les nouvelles références.

V.3.3 La recherche du point sûr

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 solution adoptée est le comptage de références, 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 méthodes pouvant être exécutées durant la recherche du point sûr. Ce seuil peut être défini par le programmeur pour chaque classe à mettre à jour.

Algorithme 4 Recherche du point sûr

Entrée : Références des méthodes modifiées, Pile de Java, CompteurReference, Seuil
Parcourir la pile Java
Pour chaque pile de thread **faire :**
  Parcourir la pile de thread à la recherche des frames associés aux méthodes restreintes
  Pour chaque frame trouvée **faire :**
    Si la frame est associée à une méthode restreinte **alors :**
      incrémenter CompteurReference
    Fin de si
  Fin de boucle pour
Fin de boucle pour
Pour chaque frame terminant son exécution **faire :**
  Si la frame est associée à une méthode restreinte **alors :**
    Décrémenter CompteurReference
    Décrémenter Seuil
    Si CompteurReference atteint zéro **alors :**
      un point sûr est trouvé
    Sinon
      Si Seuil atteint zéro **alors :**
        arrêter la recherche du point sûr et annuler la mise à jour
      Fin de si
    Fin de si
  Fin de si
Fin de boucle pour

V.3.4 Mises à jour supportées/non supportées par EmbedDSU

Les mises à jour supportées ou non par EmbedDSU sont représentées sur le Tableau V.1 :
Le système EmbedDSU ne supporte pas les mises à jour suivantes :

Modification de la 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 des signatures 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

Tableau V.1 : Les mises à jour supportées par EmbedDSU

- La modification des interfaces d'une classe ;
- La modification de la hiérarchie de la classe ;
- La modification des exceptions.

V.4 Evaluation du système EmbedDSU

L'évaluation du système EmbedDSU a été effectuée sur la plateforme d'évaluation AT91 EB40A et l'application est basée sur la machine virtuelle SimpleRTJ, qui est une machine virtuelle Java pour l'embarqué. L'évaluation s'est effectuée à la base des critères suivants :

- **Le coût du fichier de DIFF.** La mise en place du fichier de DIFF permet de réduire considérablement la taille du fichier à transférer dans la carte et ainsi de réduire le temps de transfert du correctif 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. Dans ces cas, le transfert du fichier de la nouvelle version de la classe et du fichier de fonctions de transfert représentent un coût considérable. 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 mise à jour (30 % à 57% de la taille du fichier de la nouvelle version cumulée à celui de la fonction de transfert)
- **L'occupation de la mémoire EEPROM.** Ce critère concerne le coût en occupation mémoire en EEPROM pour chaque module d'EmbedDSU. A la comparaison de la taille de la machine virtuelle initiale à celle obtenue après l'insertion des modules de mise à jour, une augmentation de 7 % est constatée. Ceci reste acceptable dans le monde de la carte à puce.
- **Le 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 des constantes de la classe. Elles permettent de spécifier les méthodes ajoutées, les méthodes modifiées (leurs adresses en mémoire, leurs nouvelles tailles, etc), les méthodes supprimées (les adresses), les champs ajoutés et supprimés, les entrées ajoutées et supprimés de la table de constantes ainsi que les nouvelles références des instances pour le processus de restauration. Le surcoût en mémoire RAM varie aussi en fonction du type de la modification.

– **La performance.** L'étude de la performance concerne la mise à jour des instances et la recherche d'un point sûr. Elle est basée sur la determination des coûts de chacune des opérations de mise à jour (l'ajout des champs , la suppression des champs et le réarrangement des champs). Le coût de la mise à jour des instances, pour chacune de ces types d'opérations est comparé au coût du GC (Garbage Collector) standard sur lequel est basée la mise à jour des instances. Le coût de la recherche d'un point sûr dépend du nombre de threads en cours d'exécution, du nombre de frames présentes dans la pile Java ainsi que le nombre de méthodes modifiées (à mettre à jour), présentes sur la pile d'exécution (méthodes restreintes).

V.5 Interêt de l'étude de EmbedDSU

L'étude de l'objectif et du fonctionnement du système EmbedDSU nous a permis de poser la problématique de la correction de la mise à jour dynamique effectuée à différents niveaux. L'intérêt porté à la correction du EmbedDSU est basée sur les motivations suivantes :

– **Adéquation aux faibles ressources.** L'évaluation du système EmbedDSU a montré que le processus est adaptés aux systèmes caractérisés par des ressources limitées comme les cartes à puce, grâce notamment à l'envoi d'un fichier de DIFF et le taux d'occupation mémoire des modules de mise à jour. Des applications utilisant de tels objets et reposant sur des technologies Java tendent à être de plus en plus utilisées dans la vie de tout les jours dans diverses domaines. L'étude d'un système dans ce contexte est intéressant pour de telles applications en constante évolution.

– **Nature critique des applications.** La technologie Java Card est la plus utilisée aujourd'hui. Elle touche des applications à caractère critique (téléphonie, protection de données personnelles, etc). L'intérêt de la mise à jour dynamique pour ce type d'applications étant établi, l'enjeu est de proposer des systèmes de mise à jour en veillant à ne pas introduire des brèches qui peuvent avoir des conséquences dramatiques dans de telles applications.

– **Représentativité et généralisation du processus.** Le système EmbedDSU est basé sur deux parties : la préparation de la mise à jours dynamique par le calcul du fichier DIFF et la mise à et l'application de la mise à jour dynamique. Plusieurs systèmes de mise à jour dynamiques (comme Jvolve) se basent sur un schéma analogue. De plus, le processus peut être généralisé à d'autre types d'applications (comme les OSGI [41]). Une étude de correction pour EmbedDSU sera exploitable par des systèmes suivant le même processus.

– **Traitement des problèmes scientifiques de la DSU.** Le système EmbedDSU se décline sur plusieurs modules et niveaux de mise à jour constituant une réponse technique aux différents problèmes scientifiques posés par la mise à jour dynamique. L'étude de ce système permet un apport scientifique à différents niveaux de mise à jour (Code, Données, recherche du point sûr, etc).

V.6 Conclusion

Nous avons présenté dans ce chapitre le système EmbedDSU : un système de mise à jour dynamique pour les applications Java Card. L'architecture de ce système se décline en deux parties : une fois les informations de mise à jour fournies en off-card, la mise à jour est réalisée par la suite en on-card par un ensemble de modules représentant principalement une extension de la JCVM. L'ensemble de ces modules constitue une solution technique à trois problématiques principales de la mise à jour dynamique : la mise à jour du code, des données et la recherche du point sur. L'intérêt que suscite un système tel que EmbedDSU, expliqué par la nature des applications auxquelles il est destiné, motive notre étude pour la quatrième problématique de la mise à jour dynamique à savoir la correction des mises à jours selon le processus proposé et qui fera l'objet de notre contribution.

Troisième partie

Contributions

Chapitre VI

Correction de la mise à jour du code

Sommaire

La mise à jour du code représente une partie basique pour chaque système de mise à jour. Elle permet de remplacer les anciennes versions des différentes parties d'un code (classes, fonctions ...) par de nouvelles versions. La correction de la mise à jour du code représente une étape primordiale dans la correction des systèmes de DSU. Nous présentons dans ce chapitre une sémantique formelle qui représente la première spécification qui capture comment le bytecode Java Card doit être vérifié pour la mise à jour. Sur la base de cette sémantique, notre travail permet d'étendre la vérification on-card pour garantir la sûreté de typage des programmes mis à jour. Après la vérification du bon typage, nous nous intéressons à la correction du comportement du programme. Nous proposons une approche permettant de garantir que la spécification du programme mis à jour correspond à la spécification souhaitée par le programmeur. Les contributions de ce chapitre sont publiées dans les références : [134, 135, 136].

VI.1 Motivations

Les cartes à puce Java sont utilisées pour stocker des données personnelles et exécuter des opérations sensibles. Elles sont également ouvertes : des applications peuvent être chargées après avoir émis la carte. De ce fait, avant son déploiement, une application Java Card passe par des

vérifications dans le but d'en assurer la validité et l'innocuité. Une application Java Card passe en effet par les phases suivantes :

- Développement et compilation : cette étape consiste à écrire le code source et le compiler pour obtenir un fichier *.class*. C'est le même processus pour le développement d'applications Java.
- Conversion et vérification : Cette étape consiste vérifier le bytecode obtenu de l'étape précédente et à le convertir *.class* en un fichier CAP (Converted APpelet). C'est un format adapté aux ressources restreintes de la carte à puce. Ce fichier est soumis à des vérifications avant son envoi sur la carte.
- Chargement et installation : Le fichier CAP vérifié est chargé sur la carte, l'environnement d'exécution enregistre et installe l'application.

La vérification du bytecode est l'un des mécanismes principaux de la sécurité Java Card. Cette vérification consiste en une vérification structurelle et une vérification de types. La vérification structurelle du programme vise à s'assurer de sa bonne formation, comme par exemple le nombre et la taille des composants du fichier CAP et la vérification des dépendances entre les composants. La vérification de types vise à respecter les règles de typage définies par Java Card et a pour rôle de protéger la machine virtuelle contre toute opération illégale ou débordement.

Le système de mise à jour dynamique EmbedDSU introduit une modification dans le processus de déploiement des applications Java Card. En effet, la machine virtuelle est étendue pour appliquer les opérations de mise à jour envoyées dans un fichier DIFF afin d'obtenir une nouvelle version. Ce schéma n'est pas exempt de problèmes. En effet, le processus mène à l'exécution d'un programme dont le bytecode n'a pas été soumis aux vérifications contenues dans le processus standard de déploiement des applications. De plus, le fichier DIFF envoyé peut contenir des erreurs pouvant compromettre le fonctionnement du programme sur la carte. L'objectif de ce chapitre est de proposer des approches formelles pour vérifier la correction du code obtenu par la mise à jour. Pour ce faire, nous nous intéressons aux propriétés suivantes :

1. **La sûreté de typage :** La sûreté de typage représente une pierre angulaire dans la sûreté et la sécurité de la plateforme Java Card. Notre but est de garantir que cette propriété est préservée par la mise à jour en garantissant que dans le programme obtenu :
   - les instructions s'appliquent sur les bon types d'arguments et l'utilisation des valeurs est en accord avec leurs types ;
   - les arguments passés en paramètres à une méthodes doivent être de types compatibles avec sa signature ;
   - il n'y' a pas de débordement de pile et on ne peut pas avoir des piles vides pour les instruction nécessitant de dépiler des valeurs ;
   - les sauts du programmes sont valides.
2. **La correction de la sémantique :** La sûreté du typage est une condition de correction nécessaire mais pas suffisante. En effet, un programme peut être mis à jour en respectant le typage des opérations mais effectuer après la mise à jour un traitement qui ne correspond pas à la spécification souhaitée par le programmeur. Cela peut arriver par exemple en remplaçant une instruction par une autre qui s'applique aux même types et nombre de valeurs et produisant le même type de résultat, mais avec un comportement différent. Nous proposons une approche permettant de s'assurer que le programme obtenu par mise à jour correspond à la specification exprimée par le programmeur.

VI.2 Langage et sémantique

VI.2.1 Présentation du langage

Nous présentons dans cette section le langage sur lequel nous construisons notre formalisation. Il fournit un sous ensemble représentatif des instructions de bytecode Java Card. Ces instructions sont relatives à la manipulation des piles, la création des objets dans le tas, l'arithmétique, l'accès aux champs, l'invocation des méthodes ainsi que les instructions de sauts. Le langage est étendu par des instructions dédiées à représenter les opérations de mise à jour. Sur la base de ce langage, nous formalisons ensuite la sémantique des opérations de mise à jour, ce qui nous permet de caractériser les mises à jour valides.

Nous introduisons d'abord les notations suivantes pour notre langage :
- **x** : représente une variable locale ;
- **a** : représente une constante ;
- **L** : représente une adresse d'instruction ;
- **A** : représente le nom d'une classe ;
- **fl** : représente le nom d'un champ ;
- **l** : représente le nom d'une méthode ;
- **t** : représente la signature d'une méthode ou le type d'un champs ;
- **pc** : représente le compteur de programme.

Nous définissons ensuite notre langage :

$$\begin{aligned} Instruction ::= &|pop\ |if\ L\ |store\ x\ |load\ x\ |new\ A \\ &|binop\ |neg\ |const\ a\ |invokevirtual\ A\ l\ t\ |goto\ L \\ &|getfield\ A\ fl\ t\ |putfield\ A\ fl\ t\ |return \end{aligned}$$

$$\begin{aligned} Upd\_Instr ::= &\ Add\_Inst\ Instruction\ pc \\ &|Dlt\_Inst\ Instruction\ pc \\ &|Mod\_Inst\ Instruction\ Instruction\ pc \end{aligned}$$

Dans ce langage, *Instruction* représente les instructions du langage bytecode et *Upd_Instr* représente les opérations de mise à jour concernant les instructions. Les instructions sont telles que :
- **pop :** cette instruction permet de dépiler le sommet de la pile.
- **const a :** cette instruction permet d'empiler la constante a au sommet de la pile.
- **if L :** cette instruction compare la valeur en sommet de pile avec 0. Si la valeur est différente de 0, l'exécution saute vers l'instruction L sinon, l'exécution continue avec l'instruction suivante.
- **store x :** cette instruction dépile une valeur à partir du sommet de la pile et la stocke dans la variable locale x.
- **load x :** cette instruction charge la valeur de la variable local x et le place au sommet de la pile.
- **new A :** cette instruction crée une nouvelle instance de la classe A et empile une référence vers cette instance sur la pile.
- **goto L :** cette instruction représente un branchement inconditionnel vers L.
- **neg :** inverse le signe de l'élément au sommet de la pile.
- **binop :** cette instruction représente les instructions arithmétiques : *add*, *sub*, *mul* et *div*. Elle dépile deux éléments en sommet de la pile puis les remplace respectivement par le résultat de leurs addition, soustraction, multiplication ou division.

- **getfield A fl t :** cette instruction permet l'accès aux attributs de la classe A pour récupérer la valeur de l'attribut *fl* ayant le type *t*. La référence de A en sommet de pile y est remplacée par la valeur de l'attribut.
- **putfield A fl t :** cette instruction permet de modifier la valeur du champs *fl* ayant le type t de la classe A. Les deux éléments au sommet de la pile d'opérandes avant l'exécution de l'instruction sont la référence vers l'instance modifiée et la nouvelle valeur du champs. Ces deux valeurs au sommet de la pile sont dépilées par cette instruction.
- **invokevirtual A l t :** cette instruction permet l'appel d'une méthode d'instance l d'une classe A et ayant la signature *t*. Pour cette instruction, la référence de l'instance ainsi que les valeurs des paramètres de la méthode se trouvent au sommet de la pile. Ils sont dépilés pour l'exécution de cette instruction. le résultat du retour est ensuite empilé.
- **return :** cette instruction représente la fin de l'exécution d'une méthode.

Les instructions de mise à jour sont respectivement :
- pour l'ajout d'instructions *Add_Inst* la suppression d'instructions *Dlt_Inst*. nous indiquons l'instruction à ajouter ou supprimer ainsi que l'emplacement de la mise à jour avec *pc*.
- La modification d'instruction est interprétée comme une suppression suivie d'un ajout.

VI.2.2 Sémantique opérationnelle

Dans cette section, nous allons expliquer les instructions du langage à travers une sémantique opérationnelle. La sémantique opérationnelle représente l'exécution d'un programme sous forme de changement d'états. Elle permet de montrer comment l'exécution de chaque instruction change l'état d'un programme. Un état est un quintuplet $< M, s, h, f, pc >$ où :
- M est la méthode en cours d'exécution ;
- s est la pile des opérandes ;
- h est le tas de la machine virtuelle ;
- f est une fonction qui donne la valeur stockée dans chaque variable locale ;
- *pc* est le compteur ordinal indiquant l'adresse de l'instruction qui sera exécutée.

La sémantique opérationnelle est définie par une relation de transition entre configurations (états). Ces transitions sont de la forme suivante : $< M, s, h, f, pc > \rightarrow < M, s2, h2, f2, pc2 >$ indiquant qu'un programme passe de l'état $< M, s, h, f, pc >$ à l'état $< M, s2, h2, f2, pc2 >$. La sémantique opérationnelle pour les instructions du langage que nous étudions est montrée par le Tableau VI.1. Dans cette représentation, nous considérons que :
- La notation $M[pc]$ représente l'instruction à l'indice *pc* dans la méthode M ;
- Les valeurs manipulées sont les entiers et les références sur les objets.
- La notation $f[x]$ représente le contenu de la variable locale x ;
- La pile d'opérandes s est considérée comme une chaîne de valeurs. L'insertion d'un élément v dans cette chaîne est noté par $v.s$;

Pour chaque instruction de bytecode une règle est définie pour exprimer sous quelles conditions une transition donnée est valide :
- la règle *Rpop* représente la sémantique de l'instruction ***pop***. Elle permet de passer à l'instruction suivante $(pc + 1)$ en retirant l'élément au sommet de la pile qui passe de $v.s$ à s.

- La règle *Rnew* représente la sémantique de l'instruction ***new A***. Elle permet crée un nouvel objet de la classe A représentée par la fonction *create* qui modifie ainsi le tas h. La référence ref de l'instance créée est empilée. L'exécution se poursuit ensuite à $pc + 1$.

- La règle $Rldx$ représente la sémantique de l'instruction ***load x***. Elle permet de placer la valeur de la variable locale x au sommet de la pile ($f[x].s$) et incrémente pc pour passer à l'instruction suivante.

- La règle $Rstx$ représente la sémantique de l'instruction ***store x***. Elle permet de placer la valeur extraite du sommet de la pile dans la variable x ($f[x \leftarrow v]$) et incrémente pc pour passer à l'instruction suivante.

- La règle $Rif1$ représente la sémantique du premier cas du ***If L***. Elle permet de passer à l'instruction L si le contenu du sommet de la pile est différent de 0 ($v \neq 0$). La deuxième règle du ***If L*** est représenté par la règle $Rif2$, dans ce cas, la valeur au sommet de la pile est un zéro, l'instruction suivante est exécuté, la valeur de pc est incrémentée.

- La règle $Rcst$ représente la sémantique de l'instruction ***const a***. Elle permet de passer à l'instruction suivante en poussant la constante a au sommet de la pile.

- La règle $Rneg$ représente la sémantique de ***neg***, elle indique que si l'instruction actuelle est neg alors l'état change en inversant le signe de la valeur qui se trouve au sommet de la pile ($-v$) , avec une incrémentation de pc pour passer à l'adresse de l'instruction suivante.

- La règle $Rget$ représente la sémantique de ***getfield A fl t***, la référence (o) dont on veut avoir la valeur du champ est au sommet de la pile. Elle est dépilé et remplacé par la valeur du champs v obtenue par la notation $v = h[o.fl]$ qui signifie, la valeur du champs fl de l'objet référencé par o dans le tas h. Ceci permet de passer à l'instruction suivante.

- La règle $Rput$ représente la sémantique de ***putfield A fl t***, la référence (o) de l'objet à modifier et la nouvelle valeur (v) du champ sont au sommet de la pile. Ils sont dépilés. La nouvelle valeur est rangée dans le champ fl. On obtient donc un nouveau tas h' en rangeant la valeur v dans le champs fl de l'objet référencé par o ($h' \leftarrow h[o.fl \leftarrow v]$). Ceci permet de passer à l'instruction suivante.

- La règle $Rgto$ représente la sémantique de ***goto L***. La configuration suivante est à l'instruction à l'adresse L.

- La règle sémantique Rop représente la sémantique des opérations arithmétiques ***Binop*** , pour passer à la configuration suivante, les deux arguments $v1$ et $v2$ de l'opérations sont dépilés du sommet de la pile. Le résultat de l'opération ($v1$ op $v2$) est ensuite empilé.

  La règle $Rinv$ représente la sémantique de l'instruction ***invokvirtual A l t***. Les n éléments au sommet de la pile ($a_1 \dots a_n$) représentent les arguments de la méthode plus la référence de l'objet. Pour exécuter cette instruction, une frame lui est créé avec sa propre pile d'opérandes initialement vide (ε), ses propres variables représentées par f'_l et son propre compteur pc_l. La méthode appelé partage le même tas h que la méthode appelante. A la fin de son exécution, le résultat v est empilé sur la pile de la méthode appelante qui poursuit son exécution à $pc+1$.

La sémantique opérationnelle permet une compréhension détaillée des instructions et permet d'introduire par la suite la sémantique formelle des opérations de mise à jour.

Tableau VI.1 : Règles de la sémantique opérationnelle

$$\frac{M[pc]{=}pop}{<M,v.s,h,f,pc>\rightarrow\ <M,s,h,f,pc{+}1>}\ (Rpop) \qquad \frac{M[pc]{=}new\ \ A,h'{=}h[create(A,ref)]}{<M,s,h,f,pc>\rightarrow\ <M,ref.s,h',f,pc{+}1>}\ (Rnew)$$

$$\frac{M[pc]{=}load\ \ x}{<M,s,h,f,pc>\rightarrow\ <M,f[x].s,h,f,pc{+}1>}\ (Rldx) \qquad \frac{M[pc]{=}store\ \ x}{<M,v.s,h,f,pc>\rightarrow\ <M,s,h,f[x\leftarrow v],pc{+}1>}\ (Rstx)$$

$$\frac{M[pc]{=}if\ \ L,v{\neq}0}{<M,v.s,h,f,pc>\rightarrow\ <M,s,h,f,L>}\ (Rif1) \qquad \frac{M[pc]{=}if\ \ l}{<M,0.s,h,f,pc>\rightarrow\ <M,s,h,f,pc{+}1>}\ (Rif2)$$

$$\frac{M[pc]{=}const\ \ a}{<M,s,h,f,pc>\rightarrow\ <M,a.s,h,f,pc{+}1>}\ (Rcst) \qquad \frac{M[pc]{=}neg}{<M,v.s,h,f,pc>\rightarrow\ <M,(-v).s,h,f,pc{+}1>}\ (Rneg)$$

$$\frac{M[pc]{=}getfield\ \ A\ \ fl\ \ t,v{=}h[o.fl]}{<M,o.s,h,f,pc>\rightarrow\ <M,v.s,h,f,pc{+}1>}\ (Rget) \qquad \frac{M[pc]{=}putfield\ \ A\ \ fl\ \ t,h'{=}h[o.fl\leftarrow v]}{<M,o.v.s,h,f,pc>\rightarrow\ <M,s,h',f,pc{+}1>}\ (Rput)$$

$$\frac{M[pc]{=}goto\ \ L}{<M,s,h,f,pc>\rightarrow\ <M,s,h,f,L>}\ (Rgto) \qquad \frac{M[pc]{=}binop,op{\in}\{+,-,*,/\}}{<M,v1.v2.s,h,f,pc>\rightarrow\ <M,(v1\ \ op\ \ v2).s,h,f,pc{+}1>}\ (Rop)$$

$$\frac{M[pc]{=}invokevirtual\ \ A\ \ l\ \ t\ ,<l,\varepsilon,h,f_l,0>\rightarrow *<l,v,h1,f'_l,pc_l>}{<M,a_1...a_n.s,h,f,pc>\rightarrow\ <M,v.s,h1,f,pc{+}1>}\ (Rinv)$$

VI.3 Sémantique des opérations de mise à jour

Nous proposons dans cette section une sémantique statique permettant de formaliser les opérations de mise à jour. Nous représentons cette sémantique par un système qui spécifie les conditions pour que chaque instruction de mise à jour soit correctement typée. Pour ce faire, nous introduisons d'abord les différents concepts et notations que nous utilisons.

VI.3.1 Notations et concepts

d) Informations de typage

Nous introduisons principalement deux éléments, F et S, pour le typage des variables locales et de la pile d'opérandes respectivement. Ces informations de typage F et S sont utiles pour garder trace de ces éléments sur toutes les addresses i de la méthode considérée.

– **Typage des variables locales :** Le symbole F représente une applications d'un point dans un programme vers une fonction associant des types aux variables locales du programme. Elle est définie de l'ensemble des adresses d'instructions $ADDR$ et l'ensemble des variables locales $\mathcal{LOC}$ de la méthode vers l'ensemble des types $\mathcal{T}$ (entiers et références). L'application F_i donne le type des variables locales à un point (adresse) i du programme :

$$\begin{array}{c} F : \ \mathcal{ADDR} * \mathcal{LOC} \rightarrow \mathcal{T} \\ (i, x) \mapsto F_i[x] \end{array}$$

Le symbole $\top$ est utilisé pour représenter les valeurs par défaut des types pour l'initialisation.

– **Typage de la pile d'opérandes :** Le symbole S représente une application d'un point du

programme vers une séquence ordonnée de type, i représente un point d'un programme ou bien l'adresse d'une instruction dans le code. La chaîne S_i permet d'obtenir les types des entrées dans la pile d'opérandes à une adresse i sous forme d'une séquence de types. La séquence vide est représentée par (ε).

$$\begin{array}{c} S : \ \mathcal{ADDR} \rightarrow \mathcal{T}^* \\ i \mapsto S_i \end{array}$$

En plus des informations sur les types des variables locales et la pile d'opérandes, nous utilisons les notations suivantes :

- $DOM(BC)$: est un ensemble des adresses dans le programme bytecode BC ;
- SD : représente la profondeur de la pile ;
- M : est une fonction qui associe un numéro à chaque ligne. Elle permet de définir formellement par une fonction des adresses vers des instructions ;
- PC_MAX est utilisé pour exprimer l'offset maximal dans le bytecode d'une méthode.

On définit une configuration à une ligne i comme un tuple $< (F, S, SD, M), i >$ où : F et S représentent les informations de typage, SD, la profondeur de la pile, M un mapping associant un numéro à chaque ligne dans le bytecode BC.

VI.3.2 Règles sémantiques

Nous définissons maintenant une règle qui nous permet de garantir qu'un bytecode BC est bien typé par F et S :

$$\frac{\begin{array}{l} F_1 = F_\top, SD1 = 0 \\ S_1 = \varepsilon, M1 = Map(BC) \\ \forall i \in DOM(BC), F, S, i \vdash BC \end{array}}{F, S \vdash BC}$$

En effet, ces informations de typage permettent de garantir que toutes les instructions du programme mis à jour utilisent les bons types d'arguments dans la pile d'opérandes et les variables locales. Le système permet aussi de s'assurer qu'on n'aura pas de débordement de piles ainsi que des instructions nécessitant de dépiler, ne s'appliquent à une pile vide. Les deux premières lignes de la règle représentent la configuration initiale : toutes les variables sont associées à la valeur par défaut $\top$, la profondeur de la pile est à zéro, la séquence des types pour la pile d'opérandes est vide et $M1$ est le mapping initial du bytecode BC ($Map(BC)$). Dans la dernière ligne, l'expression $F, S, i \vdash BC$ signifie que BC est bien typé jusqu'à l'étape i dans l'évaluation. La ligne entière signifie que le programme BC est bien typé à chaque étape i de l'évaluation. L'inférence du jugement exprime que BC est bien typé. La prémisse $F, S, i \vdash BC$ est dérivée de la sémantique des opérations de mise à jour que nous présentons maintenant en détail.

VI.3.2.1 Sémantique des opérations d'ajout d'instructions

Pour une bonne lisibilité du document, nous adoptons une classifications pour présenter ces règles selon ces catégories :

- instructions manipulant la pile d'opérandes ;
- instructions manipulant les variables locales ;
- instructions manipulant des instances d'objets ;
- instructions de sauts et d'appel de méthodes.

Les règles sémantiques expriment quelles sont les vérifications à effectuer pour valider l'insertion d'une instruction à un point donné du programme. Pour toutes les règles que nous présentons, la première ligne correspond toujours à l'opération de mise à jour. Cette ligne est suivie des vérifications à effectuer (les premisses) pour avoir un changement de configuration valide (la conclusion). Les vérifications suivantes sont communes à toutes les instructions :

- Vérification que l'adresse d'insertion est une adresse valide dans la méthode, exprimé par $i+1 \in DOM(BC)$;
- Vérification que le PC_MAX de la méthode est incrémenté pour chaque insertion ;
- Dans ces règles, $M2$ est le résultat de l'insertion dans $M1$. Les opérations permettant la manipulation des bytecodes à cet effet sont dites fonctions auxiliaires.

a) Les fonctions auxiliaires utilisées

Les fonctions auxiliaires sont définies pour réajuster les adresses des instructions et les sauts dans le code suite à l'insertion ou à la suppression d'instructions :

- Les fonctions ***range n m*** et ***shift n m p***. La fonction *range* permet extraire d'un mapping $M1$ une partie $M2$ incluse entre les lignes n et m. La fonction $shift$ permet de décaler un mapping M contenu entre n et m de p positions.
- Les fonctions ***look_for_jumps m*** et ***update_jumps m l***. Les décalage effectués sur un programme suite l'insertion ou la suppression d'une instruction peut aboutir à des erreurs dans les sauts. Il convient donc d'effectuer les corrections nécessaires pour préserver la cohérence des instructions de saut (*If L* et *goto L*). La fonction *look_for_jumps m* retourne à partir d'un mapping la liste des adresses d'instructions de sauts. La fonction *Update_jumps* permet de modifier les instructions de sauts contenues dans un mapping. Les adresses des sauts sont représentées par une liste d'entiers. Cette fonction prend également un entier représentant la valeur avec laquelle on souhaite modifier les sauts. Le résultat est un nouveau mapping.

b) Les instructions manipulant la pile d'opérandes

Le Tableau VI.2 montre les règles sémantiques pour l'ajout des instructions suivantes : *pop*, *neg*, *Binop* et *const a*. Ces instructions n'agissent pas sur les variables locales. De ce fait, une vérification est effectuée pour s'assurer que le passage de la configuration i à $i+1$ n'affecte pas les variables locales ($F_{i+1} = F_i$). Les autres vérifications sont comme suit :

- pour ajouter correctement une instruction ***pop*** à l'adresse $i+1$, la vérification est effectuée pour garantir qu'un type t est au sommet de la pile correspondant à l'adresse i et que ce type est dépilé. La profondeur de la pile est décrémentée.
- pour ajouter correctement une instruction ***const a*** à l'adresse $i+1$, la vérification est effectuée pour garantir qu'on empile au sommet de la pile correspondant à l'adresse i le type *int* pour avoir la pile correspondant à l'adresse $i+1$. La profondeur de la pile est incrémentée.
- pour ajouter correctement une instruction ***neg*** à l'adresse $i+1$, la vérification est effectuée pour garantir qu'un type *int* est au sommet de la pile correspondant à l'adresse i et que cette pile reste inchangée à $i+1$. La profondeur de la pile reste inchangée aussi.
- pour ajouter correctement une instruction ***Binop*** à l'adresse $i+1$, la vérification est effectuée pour garantir que deux entiers sont au sommet de la pile correspondant à l'adresse i. Ces deux élément sont dépilés et remplacés par le type du résultat (*int*). La profondeur de la pile est décrémentée.

Tableau VI.2 : Règles pour l'insertion des instructions (1)

$$\frac{\begin{array}{l} Add\_inst\ \ pop\ (i+1) \\ SD_{i+1} = SD_i - 1 \quad F_{i+1} = F_i \\ S_i = t.S_0 \Rightarrow S_{i+1} = S_0 \\ M2 = Add\_inst(M1, pop, i+1) \\ PC\_MAX++ \\ i+1 \in DOM(BC) \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Add\_inst\ \ const\ a(i+1) \\ SD_{i+1} = SD_i + 1 \\ PC\_MAX++ \\ S_{i+1} = int.S_i \quad F_{i+1} = F_i \\ M2 = Add\_inst(M1, const\ a, i+1) \\ i+1 \in DOM(BC) \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Add\_inst\ \ neg\ (i+1) \\ SD_{i+1} = SD_i \ F_{i+1} = F_i \\ S_i = int.S_0 = S_{i+1} \\ M2 = Add\_inst(M1, neg, i+1) \\ PC\_MAX++ \\ i+1 \in DOM(BC) \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Add\_inst\ Binop(i+1) \\ SD_{i+1} = SD_i - 1 \quad PC\_MAX++ \\ S_i = int.int.S_0 \Rightarrow \\ S_{i+1} = int.S_0 \\ M2 = Add\_inst(M1, Binop, i+1) \\ i+1 \in DOM(BC) \quad F_{i+1} = F_i \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

Tableau VI.3 : Règles pour l'insertion des instructions (2)

$$\frac{\begin{array}{l} Add\_inst\ \ load\ x(i+1) \\ SD_{i+1} = SD_i + 1 \\ PC\_MAX++ \\ S_{i+1} = F_i[x].S_i \quad F_{i+1} = F_i \\ M2 = Add\_inst(M1, load\ x, i+1) \\ i+1 \in DOM(BC)\ x \in LOC(BC) \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Add\_inst\ \ store\ x(i+1) \\ SD_{i+1} = SD_i - 1 \quad PC\_MAX++ \\ S_i = t.S_0 \Rightarrow S_{i+1} = S_0 \\ F_{i+1} = F_i[x \leftarrow t] \\ M2 = Add\_inst(M1, store\ x, i+1) \\ i+1 \in DOM(BC)\ x \in LOC(BC) \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

c) Les instructions manipulant les variables locales

Le Tableau VI.3 montre les règles sémantiques pour l'ajout des instructions *load x* et *store x* :

- pour ajouter correctement une instruction ***load x*** à l'adresse $i+1$, une vérification sur l'appartenance de x aux variables locales de la méthode est effectuée ($x \in LOC(BC)$) et la profondeur de la pile est incrémentée. Le typage des variables est inchangé et le type de la variable x est inséré à la pile des type qui résulte de l'instruction à l'adresse i.
- pour ajouter correctement une instruction ***store x*** à l'adresse $i+1$, la vérification est effectuée pour garantir la compatibilité du type de la variable x avec le sommet de la pile de l'instruction à l'adresse i. Une vérification sur l'appartenance de x aux variables locales de la méthode est également effectuée . La profondeur de la pile est décrémentée.

Tableau VI.4 : Règles pour l'insertion des instructions (3)

$$\frac{\begin{array}{l} Add\_inst\ getfield(A,f,t)(i+1) \\ SD_{i+1} = SD_i \\ S_i = A.S_0 \Rightarrow S_{i+1} = t.S_0 \\ M2 = \\ Add\_inst(M1, getfield(A,f,t), i+1) \\ PC\_MAX + 3 \quad F_{i+1} = F_i \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Add\_inst\ putfield(A,f,t)(i+1) \\ SD_{i+1} = SD_i - 2\ F_{i+1} = F_i \\ S_i = A.t.S_0 \Rightarrow S_{i+1} = S_0 \\ M2 = \\ Add\_inst(M1, putfield(A,f,t), i+1) \\ PC\_MAX + 3 \quad i+1 \in DOM(BC) \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

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

d) Les instructions manipulant des instances d'objets

Le Tableau VI.4 montre les règles sémantiques pour l'ajout des instructions *getfield*, *putfield* et *new*. Ces instructions n'agissent pas sur les variables locales. De ce fait, une vérification est effectuée pour s'assurer que le passage de la configuration i à $i+1$ n'affecte pas les variables locales ($F_{i+1} = F_i$). Les autres vérifications sont comme suit :

- pour ajouter correctement une instruction ***getfield A f t*** à la ligne $i+1$, la vérification est effectuée pour garantir que le type d'une référence sur un objet de type A est au sommet de la pile correspondant à l'adresse i et que ce type est dépilé et remplacé par le type du champs consulté. La profondeur de la pile reste donc inchangée.
- pour ajouter correctement une instruction ***putfield A f t*** à la ligne $i+1$, la vérification est effectuée pour garantir que le type d'une référence sur un objet de type A et le type t du champs à modifier (correspondant au type de la nouvelle valeur du champs) sont au sommet de la pile correspondant à l'adresse i. Ces deux élément sont dépilés à $i+1$. La profondeur de la pile est de ce fait décrémentée de deux.
- pour ajouter correctement une instruction ***new A*** à la ligne $i+1$, la vérification est effectuée pour garantir qu'on empile au sommet de la pile correspondant à l'adresse i le type référence pour un objet A pour avoir la pile correspondant à l'adresse $i+1$. La profondeur de la pile est incrémentée.

e) Les instructions de sauts et d'appel de méthodes

Le Tableau VI.5 montre les règles sémantiques pour l'ajout des instructions *If L*, *goto L* et *invokevirtual A l t*. Ces instructions n'agissent pas sur les variables locales. De ce fait, une vérification est effectuée pour s'assurer que le passage de la configuration i à $i+1$ n'affecte pas les variables locales ($F_{i+1} = F_i$). Les autres vérifications sont comme suit :

Tableau VI.5 : Règles pour l'insertion des instructions (4)

$$\frac{\begin{array}{l} Add\_inst\ \ goto\ L(i+1) \\ SD_{i+1} = SD_i\ \ PC\_MAX++ \\ S_{i+1} = S_i\ \ \ F_{i+1} = F_i \\ M2 = \\ Add\_inst(M1, goto\ L, i+1) \\ i+1, L \in DOM(BC) \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Add\_inst\ \ if\ L(i+1) \\ SD_{i+1} = SD_i - 1 \\ PC\_MAX++ \quad F_{i+1} = F_i \\ S_i = int.S_0 \Rightarrow S_{i+1} = S_0 \\ M2 = \\ Add\_inst(M1, if\ L, i+1) \\ i+1, L \in DOM(BC) \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Add\_inst\ \ invokevirtuel(A,l,t)(i+1) \\ SD_{i+1} = SD_i - (card(dom(t)) + 1) \\ S_{i+1} = A.tn_1.tn_2 \ldots tn_n.S_0 \Rightarrow S_{i+1} = codom(t).S_0 \\ comp(dom(t), n_1.tn_2 \ldots tn_n) \\ M2 = Add\_inst(M1, invokevirtuel(A,l,t), i+1) \\ i+1 \in DOM(BC)\ \ F_{i+1} = F_i \\ PC\_MAX + 3 \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

- pour ajouter correctement une instruction ***If L*** à l'adresse $i+1$, la vérification est effectuée pour garantir qu'un type *int* est au sommet de la pile correspondant à l'adresse i et que ce type est dépilé. La profondeur de la pile est décrémentée. Une vérification est également effectué pour s'assurer que L est une adresse valide dans le programme.
- pour ajouter correctement une instruction ***goto L*** à l'adresse $i+1$, la vérification est effectuée pour garantir la pile reste inchangée à $i+1$. La profondeur de la pile reste inchangée aussi. Une vérification est également effectué pour s'assurer que L est une adresse valide dans le programme.
- pour ajouter correctement une instruction ***invokevirtual A l t*** à l'adresse $i+1$, la vérification est effectuée pour garantir que les types des paramètres de la méthode ainsi que la référence de l'objet sur lequel on l'invoque se trouvent au sommet de la pile correspondant à l'adresse i. Ces éléments sont dépilés et remplacé après l'évaluation de l'instruction par le type du résultat retourné. Dans cette règles, l'expression $card(dom(t))$ représente le nombre d'arguments de la méthode. L'expression $comp(dom(t), n_1.tn_2 \ldots tn_n)$ signifie qu'on vérifie que les type des arguments correspondent aux type présents au sommet de la pile. Le nombre des arguments est soustrait de la profondeur de la pile.

VI.3.2.2 Sémantique des opérations de suppression d'instructions

La formalisation de la sémantique pour les opérations de suppression d'instruction est différente de celle des ajouts. La vérification de la validité d'une suppression fait intervenir l'instruction qui vient après l'instruction supprimée. Contrairement à la vérification pour les ajouts où nous vérifions la validité de l'opération vis-a-vis de typage des données relatives à l'adresse i pour une insertion à $i+1$, la vérification de la validité d'une suppression à $i+1$ met en relation les deux instructions qui se trouvent avant et après $i+1$. (Resp. i et $i+2$). Le Tableau VI.6 montre des règles pour la suppression des instructions suivantes : *new A*, *Binop*, *load x* et *If L*. Les autres règles pour la

Tableau VI.6 : Règles pour la suppression d'instructions

$$\frac{\begin{array}{l} Dlt\_inst\ \ new\ A\ \ (i+1) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, new\ A, i+1) \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX-- \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Dlt\_inst\ \ (Binop\ \ (i+1)) \\ M2 = Dlt\_inst(M1, Binop, i+1) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ S_i = int.int.S_0 \rightarrow \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\quad PC\_MAX-- \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Dlt\_inst\ \ (store\ x\ \ (i+1)) \\ SD_i = a \rightarrow \\ \quad SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, store\ x, i+1) \\ S_i = t.S_0 \wedge F_i[x] = t \rightarrow \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], t.S_0) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX-- \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Dlt\_inst\ \ if\ L\ \ (i+1) \\ SD_i = a \rightarrow \\ \quad SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, if\ L, i+1) \\ S_i = int.S_0 \rightarrow \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX-- \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

suppression sont présentées en annexe. Dans les règles de suppression, nous adoptons les notations suivantes :

- Les notations $Effect\_STK$, $Effect\_F$ et $Effects\_SD$ sont utilisés pour exprimer l'effet d'une instruction sur la pile des types, les variables locales et la profondeur de la pile. Elles sont utilisées pour garantir que, dans le nouveau mapping obtenu après la suppression, l'instruction qui remplace l'instruction supprimé (donc la nouvelle instruction à l'adresse $i+1$) corresponde bien au contenu de la pile des type et des variables locales en terme de type et du nombre d'arguments.
- La notation $(M2)F$ (resp., $(M2)S$) est utilisée pour faire référence aux informations de typage F (resp., S) dans le mapping $M2$ (c'est à dire après la suppression). Le mapping $M2$ est calculée avec les fonctions auxiliaires expliquées dans la sémantiques des ajouts.

Ainsi, pour l'ensemble des règles, la vérification est effectuée en utilisant $Effects\_SD$ pour garantir que la profondeur de la pile, avant l'instruction supprimée, est cohérente avec l'instruction qui la remplace dans le mapping. La vérification effectuée permet également de garantir la sûreté vis-à-vis des informations de typage. Pour la suppression de l'opération *Binop* à $i+1$ par exemple, les deux éléments au sommet de la pile correspondant à l'adresse i représentent deux entiers. Cette configuration de la pile doit correspondre à l'instruction qui s'insère à $i+1$ après la suppression de *Binop*. L'écriture $(M2)S_{i+1} = Effects\_STK(M2[i+1], S_i)$ signifie que l'information de typage S à l'adresse $i+1$ résulte de la bonne application de l'instruction à l'adresse $i+1$, après suppression, sur la configuration de la pile des types à l'adresse i. Ce type de vérification est relatif, dans la suppression de *If L* et *store x* à un sommet de pile correspondant respectivement à *int* et le type de la variable x.

Tableau VI.7 : Exemple pour les informations de typage

i	instruction	F_i	S_i
0	const 0	$(1, int), (2, int), (3, int)$	$int.\varepsilon$
1	store 3	$(1, int), (2, int), (3, int)$	ε
2	load 1	$(1, int), (2, int), (3, int)$	$int.\varepsilon$
3	load 2	$(1, int), (2, int), (3, int)$	$int.int.\varepsilon$
upd	Del_inst add (4)	$(1, int), (2, int), (3, int)$	$int.int.\varepsilon$
upd	Add_inst sub (4)	$(1, int), (2, int), (3, int)$	$int.int.\varepsilon$
4	sub	$(1, int), (2, int), (3, int)$	$int.\varepsilon$
5	store 3	$(1, int), (2, int), (3, int)$	ε
6	load 3	$(1, int), (2, int), (3, int)$	$int.\varepsilon$
7	return	$(1, int), (2, int), (3, int)$	ε

Exemple VI.1

Pour illustrer comment les informations de typage évoluent selon la sémantique des instructions, nous reconsidérons l'exemple du programme mis à jour présenté en Section V.2.1.2. La fonction calculant la somme de deux entiers est mise à jour pour calculer la différence au lieu de la somme. La mise à jour consiste à supprimer un *add* et insérer un *sub*. Ces deux instructions représentent une opération *Binop*. L'évolution est montrée sur le Tableau VI.7. La première colonne représente les numéros des instructions. Les instructions de mise à jour sont indiquées par *upd*. La troisième colonne représente l'évolution des informations de typage relatives aux variables locales. Celles-ci sont représentées par des entiers (1, 2 et 3). Un type est associé à chaque variable sous forme d'un couple $(var, type)$. L'évolution des type dans la pile d'opérandes est montrée dans la quatrième colonne. Au niveau des instructions de mise à jour, les informations de typage garantissent que les variables locales et la pile d'opérandes sont conformes aux instructions selon la sémantique. Cet exemple montre que la vérification des opérations de mise à jour réussit car l'évaluation fournit les arguments adéquats au sommet de la pile, particulièrement pour l'instruction ajoutée.

VI.3.3 Cas de mise à jour sur les méthodes et les champs

Nous proposons maintenant une extension de la sémantique formelle pour les mises à jour au niveau des méthodes et des champs dans une classe. Dans cette formalisation, une classe est déterminée par :

- C : son nom ;
- Mt : une table de méthodes contenant une entrée pour chaque méthode m dans la classe ;
- Ft une table des champs contenant une entrée pour chaque champ f dans la classe ;
- $Meta$ une ensemble de méta données contenant une table de constantes, le statut de la classe et la table des offsets.

VI.3.3.1 Mise à jour des méthodes

Cette section présente la sémantique pour les mises à jour au niveau des méthodes. Nous considérons qu'une méthode est définie par :

- son nom m ;
- son code BC ;
- sa signature sig ;

– une structure lm représentant des informations incluant le maximum des variables locales, et la taille maximale pour sa pile d'opérandes.

Trois cas sont considérés pour les mises à jour sur les méthodes :

1. ***Add_method :*** Cette opération permet d'introduire une nouvelle méthode dans la classe C.
2. ***Dlt_method :*** Cette opération permet permet de supprimer une méthode d'une classe C.
3. ***Mod_method :*** Cette opération permet la modification d'une méthode existante dans le but d'implementer une nouvelle version en ajoutant ou en supprimant des instructions ou des variables locales.

La formalisation requiert la définition d'extension aux fonctions pour exprimer les informations de typage. Nous introduisons d'abord les ensembles M , Fs et Ss représentant respectivement :

– L'ensemble des noms des méthodes ;
– L'ensemble des information de typage F pour représenter le type des variables locales sur l'ensemble des méthodes ;
– L'ensemble des information de typage S pour représenter les types des entrées dans la pile d'opérandes sur l'ensemble des méthodes.

Les notation F et S gardent les mêmes définitions introduites dans la sémantiques des mises à jour des instructions. Nous introduisons deux fonctions F^c et S^c :

– la fonction $F^c : M \rightarrow Fs$ est utilisée pour associer à chaque méthode m dans la classe C à une application F représentant les information de typage sur ses variables locales ;
– la fonction $S^c : M \rightarrow Ss$ est utilisée pour associer à chaque méthode m dans la classe C une application S représentant les informations de typage de sa pile d'opérande.

Le Tableau VI.8 illustre les règles sémantiques pour les opération d'ajout, de suppression et de modification de méthodes garantissant le bon typage :

– La règle ($Rm1$) exprime l'ajout d'une nouvelle méthode. Dans cette règle, la première ligne représente l'opération. La deuxième ligne représente la vérification que la méthode ajoutée n'existe pas déjà dans la table des méthodes. Cette vérification permet d'éviter que l'opération remplace par inadvertance une méthode avec le même nom. La fonction *look_for_entry* retourne la valeur booléenne vraie si la méthode avec les même nom et signature existe, faux sinon. Dans ce cas, une entrée est créée dans la table des méthodes avec *create_entry*. La création de l'entrée permet de créer et initialiser les informations de typage relatives à la méthode. L'expression $init\_S(S^c(m), S_s, \varepsilon)$ représente l'initialisation de la pile des type, pour la méthode ajoutée, à ε. L'expression $init\_F(F^c(m), Im.loc, F_s)$ représente l'initialisation des type des variables locales $Im.loc$ au type par défaut. La notation *upd_meta* est utilisée pour exprimer la mise à jour des méta données de la classe avec les informations relatives à la méthode pour mettre à jour ses offsets. La conclusion de la règle sémantique exprime que la classe C est bien typée par rapport aux informations de typage présentes dans F^c et S^c.

– La règle ($Rm2$) formalise l'opération de suppression de méthodes. La vérification en seconde ligne permet de s'assurer que cette méthode existe dans la table des méthodes. Cette opération permet de supprimer l'entrée de la méthode de la table des méthodes de la classe ainsi que les informations de typage la concernant. La dernière prémisse concerne le réajustement des offsets de la classe.

– La règle ($Rm3$) exprime la modification d'une méthode existante par un fichier Δ contenant les modifications. La vérification en ligne 2 permet de s'assurer que la méthode existe dans la

Tableau VI.8 : Règles sémantiques pour les méthodes

$$\frac{\begin{array}{l} Add\_method(m, lm, BC, sig) \\ look\_for\_entry(Mt, m, sig) = false \\ \quad \Rightarrow create\_entry(Mt, m, sig) \\ init\_S(S^c(m), S_s, \varepsilon) \\ init\_F(F^c(m), Im.loc, F_s) \\ upd\_meta(Meta, m) \end{array}}{F^c, S^c \vdash C} \; (Rm1)$$

$$\frac{\begin{array}{l} Mod\_method(m, lm, BC, sig, \Delta) \\ look\_for\_entry(Mt, m, sig) = true \\ var\_upd(m, Im.loc) \\ init(F^c(m), Im.loc) \\ upd\_meta(Meta, m) \\ S^c(m), F^c(m) \vdash BC \end{array}}{F^c, S^c \vdash C} \; (Rm3)$$

$$\frac{\begin{array}{l} Del\_method(m, lm, BC, sig) \\ look\_for\_entry(Mt, m, sig) = true \\ \quad \Rightarrow Del\_entry(Mt, m, sig) \\ S_s \leftarrow S_s \setminus S^c(m) \\ F_s \leftarrow F_s \setminus F^c(m) \\ upd\_meta(Meta, m) \end{array}}{F^c, S^c \vdash C} \; (Rm2)$$

table des méthodes. L'information $F^c(m)$ est réinitialisée après la réalisation des mises à jour potentielles dans les variables locales de la méthode ($var\_upd(m, Im.loc)$). Les informations de typage doivent valider la correction de toutes les instructions de la nouvelle méthodes (la dernière prémisse). Cette condition est basée sur les vérifications effectuées au niveau des instructions et définies par la sémantiques sur les instructions. La fonction ($var\_upd(m, Im.loc)$) est utilisée pour la mise à jour des variables locales. La cas d'ajout, de suppression ou de modification d'une variable locale (définie par son nom et son type) est un cas particulier pour la modification des méthodes.
Notons que la mise à jour de la signature d'une méthode est considérée comme une suppression de l'ancienne version, suivi d'un ajout de la nouvelle méthode.

VI.3.3.2 Mise à jour des champs

Cette section présente la sémantique pour les opérations relatives aux champs. Dans cette formalisation, un champs est représenté pas son nom f et son type tf. Une entrée dans la table des champs est créée pour chaque champs d'une classe. L'information sur le typage est fournie par une application Fl qui permet de garder trace des informations de typage des champs en associant un type à chaque champs à chaque point du programme. La figure suivante représente les règles pour l'ajout, la suppression et la modification d'un champs dans une classe.

– La règle ($Rf1$) exprime l'introduction d'un nouveau champ. Dans cette règle, par analogie aux règles sur les méthodes, la première ligne représente l'opération. La deuxième ligne représente la vérification si le champ ajouté existe déjà. La fonction *look_for_entry* retourne un booléen true si un champ avec les mêmes nom et type existe, faux sinon. Dans ce cas, une entrée pour le nouveau champ est créée dans la table des champs avec la fonction *create_entry*. L'information concernant le type est enregistrée dans Fl.
La conclusion de la règle sémantique exprime que la classe C est bien typée par rapport aux informations de typage présentes dans F^c S^c, et Fl.

Tableau VI.9 : Règles sémantiques pour les champs

$$\frac{\begin{array}{l}Add\_field(f, tf, val)\\ look\_for\_entry(Ft, f, tf) = false\\ \quad \Rightarrow create\_entry(Ft, f, tf)\\ Fl \leftarrow Fl \cup \{(f, tf)\}\end{array}}{F^c, S^c, Fl \vdash C}\ (Rf1) \qquad \frac{\begin{array}{l}Del\_field(f, tf)\\ look\_for\_entry(Ft, f, tf) = true\\ \quad \Rightarrow Del\_entry(Ft, f, tf)\\ Fl \leftarrow Fl \setminus \{(f, tf)\}\end{array}}{F^c, S^c, Fl \vdash C}\ (Rf2)$$

$$\frac{\begin{array}{l}Mod\_field(f, tf, val)\\ look\_for\_entry(Ft, f, tf) = true\\ \quad \Rightarrow Set\_val(Ft, f, tf, val)\end{array}}{F^c, S^c, Fl \vdash C}\ (Rf3)$$

– La règle $(Rf2)$ formalise l'opération de suppression de champs. La vérification à la deuxième ligne permet de s'assurer que le champ existe dans la table des champs. Cette opération résulte ne la suppression de l'entrée dans la table des champs ainsi que son information de typage.

– La règle $(Rf3)$ exprime la modification d'un champ existant. La vérification à la deuxième ligne permet de s'assurer que le champ existe dans la table des champs. La modification concerne la valeur du champs en utilisant la fonction *Set_val*. Une modification dans le type est considérée comme une suppression du champs suivi d'un ajout d'un champ avec un nouveau type.

VI.4 La sûreté du typage

Pour garantir la validité de la mise à jour d'un programme, nous devons nous assurer que tout programme obtenu après modification est bien typé. A cet effet, nous allons procéder pour démontrer ce résultat en utilisant la preuve pas induction sur le nombre d'opérations de mise à jour. Nous allons d'abord définir et démontrer quelques lemmes intermédiaires.

Définition1 *(**Correction initiale**). Considérons un code BC, son mapping initial M et ses information de typage F et S. A la configuration initiale $< F_\top, \varepsilon, 0, M, 0 >$ où les types des variables sont initialisés à une valeur par défaut, l'information de typage sur la pile d'opérande vide, la profondeur de la pile nulle, et le numéro de l'étape d'évaluation égale à zéro, le programme est bien typé, nous notons : $F, S, 0 \vdash BC$.*

Cette définition est utilisée pour exprimer les lemmes et théorèmes suivants.

Lemme 1 ***(Correction d'une étape de mise à jour)****. Etant donnée un programme BC, des informations de typage F et S, un mapping M et un correctif Δ contenant les opérations de mise à jour, nous avons :*

$\forall\ SD, SD' \in N\ , i, i' \in DOM(BC),\ j \in N.$
$F, S, i \vdash BC$
$\wedge \quad < F_i, S_i, SD, M, i > \overset{\Delta(j)}{\rightarrow} < F_{i'}, S_{i'}, SD', M', i' >$
$\wedge \quad j \leq length(\Delta)$
$\Rightarrow\ F, S, i' \vdash BC$

Ce lemme exprime que l'évaluation d'une opération de mise à jour à une étape i tel que le programme est bien typé jusqu'à i produit une programme bien typé jusqu'à la prochaine étape évaluée. La transition $\overset{\Delta(j)}{\rightarrow}$ représente l'application d'une instruction de mise à jour à la ligne j. Cette instruction doit être une instruction valide du correctif Δ. L'expression $length(\Delta)$ représente le nombre d'instructions de mise à jour dans le correctif qui est sous forme de liste d'instructions de mise à jour. M' représente le mapping résultant.

Preuve. La preuve de ce lemme est basée sur le fait de prouver que l'application de chacune des instructions de mise à jour, représentées par leurs sémantiques, préserve le bon typage. Nous présentons la preuve pour l'insertion des instructions suivantes : *new A*, *Binop*, *store x* et *If L* :

- ***Cas 1 :*** $\Delta(j)$ représente l'instruction de mise à jour *add_inst new A (i+1)*. Nous supposons que les hypothèses sont satisfaites. Cette instruction de mise à jour produit, par la sémantique, une configuration où : $F_{i'} = F_i$, $S_{i'} = A.S_i$, $SD' = SD + 1$, $i' = i + 1$, et le mapping M' est obtenu en insérant la nouvelle instruction en appliquant les opérations *shift*, *range*, et *Update_jumps*.

- ***Cas 2 :*** $\Delta(j)$ représente l'instruction de mise à jour *add_inst Binop i+1*. Nous supposons que les hypothèses sont satisfaites. Cette instruction de mise à jour produit, par la sémantique, une configuration où : $F_{i'} = F_i$, $S_{i'} = int.S_0$ tel que $S_i = int.int.S_0$, $SD' = SD - 1$, $i' = i + 1$, et le mapping M' est obtenu en insérant la nouvelle instruction en appliquant les opérations *shift*, *range*, et *Update_jumps*.

- ***Cas 3 :*** $\Delta(j)$ représente l'instruction de mise à jour *add_inst store x i+1*. Nous supposons que les hypothèses sont satisfaites. Cette instruction de mise à jour produit, par la sémantique, une configuration où : $F_{i+1} = F_i[x \leftarrow t]$, $S_{i'} = S_0$ tel que $S_i = t.S_0$, $SD' = SD - 1$, $i' = i + 1$, $x \in LOC(BC)$ et le mapping M' est obtenu en insérant la nouvelle instruction en appliquant les opérations *shift*, *range*, et *Update_jumps*.

- ***Cas 4 :*** $\Delta(j)$ représente l'instruction de mise à jour *add_inst If L i+1*. Nous supposons que les hypothèses sont satisfaites. Cette instruction de mise à jour produit, par la sémantique, une configuration où : $F_{i'} = F_i$, $S_{i'} = S_0$ tel que $S_i = int.S_0$, $SD' = SD - 1$, $i' = i + 1$, $L \in DOM(BC)$ et le mapping M' est obtenu en insérant la nouvelle instruction en appliquant les opérations *shift*, *range*, et *Update_jumps*. □

La correction au niveau d'une étape de mise à jour permet d'exprimer la correction de multiples étapes et la correction au niveau méthode.

Lemme 2 *(**Correction d'étapes multiples**). Étant donné un programme BC, des informations de typage F et S, un mapping M et un correctif Δ contenant les opérations de mise à jour, nous avons :*

$\forall\ SD, SD' \in N\ , i, i' \in DOM(BC),\ j \in N.$
$F, S, i \vdash BC$
$\wedge \quad < F_i, S_i, SD, M, i > \overset{\Delta(j*)}{\rightarrow} < F_{i'}, S_{i'}, SD', M', i' >$
$\wedge \quad j \leq length(\Delta)$
$\Rightarrow\ F, S, i' \vdash BC$

Ce lemme exprime qu'une séquence de correctes étapes de mise à jour sur un programme bien typé produit un programme bien typé. Le symbole $\overset{\Delta(j*)}{\rightarrow}$ représente l'application d'instructions de mise à jour contenues dans Δ allant de la première à la $j^{ème}$ instruction de mise à jour. Ceci est représenté par une séquence de transitions commençant par la configuration correspondant à i et se terminant à la configuration correspondant à i' : $C_i \overset{\Delta(1)}{\rightarrow} C_{i+1} \overset{\Delta(2)}{\rightarrow} C_{i+2} \ldots \overset{\Delta(j)}{\rightarrow} C_{i'}$. M' représente le mapping résultant.

Preuve. La preuve de ce lemme est effectuée par induction sur la longueur du correctif $\Delta(j*)$ (le nombre d'instruction de mise à jour qu'il contient). Nous supposons que les hypothèses sont satisfaites

- ***Cas de base :*** $length(\Delta(j*)) = 0$. La configuration reste inchangée et de ce fait, satisfaite par les hypothèses.
- ***Cas d'induction :*** Nous supposons que la propriété est satisfaite pour $length(\Delta(j*)) = n$. De ce fait, pour $length(\Delta(j*)) = n + 1$, la démonstration est ramené à l'étude des différents cas pour l'instruction de mise à jour (n+1) et est analogue à la démonstration du lemme 1.□

Ce lemme introduit la mise à jour au niveau du code entier d'une méthode.

Théorème 1 *(**Correction de la mise à jour d'une méthode**). Étant donnée une méthode m, son code BC, un mapping initial M, des informations de typage F et S, et un correctif Δ_m contenant les opérations de mise à jour relatives à la méthode. La mise à jour d'une méthode m selon Δ_m en partant d'une configuration initiale C_0 telle que $C_0 =< F_\top, \varepsilon, 0, M, 0 >$ produit un programme bien typé.*
$\forall\ SD' \in N\ i' \in DOM(BC).$
$C_0 \overset{\Delta_m}{\rightarrow} < F_{i'}, S_{i'}, SD', M', i' >$
$\wedge \quad \exists j, j \geq i' \wedge j \in DOM(BC) \wedge BC(j) = return$
$\Rightarrow\ F, S \vdash BC$

Ce théorème exprime que la correction de la mise à jour d'une méthode selon des instructions de mise à jour contenues dans un correctif Δ_m en commençant par la configuration initiale produit une méthode ayant un code bien typé. L'étape i' représente la configuration obtenu après l'application de tout le correctif Δ_m. La transition $\overset{\Delta_m}{\rightarrow}$ représente l'application des instructions de mise à jour contenues dans Δ_m qui peuvent être représentées par une séquence de transitions $C_0 \overset{\Delta_m(0)}{\rightarrow} C_1 \overset{\Delta_m(1)}{\rightarrow} C_2 \ldots \overset{\Delta_m(j)}{\rightarrow} C_{i'}$ où $\Delta_m(j)$ représente la dernière instruction du correctif. La suite de l'évolution jusqu'à la fin de la méthode (l'instruction *return*) est évalué selon les instructions standard du langage. La démonstration de ce théorème est effectuée par induction sur la longueur du correctif relatif à la méthode et suit donc le même schéma que la preuve du lemme précédant.

Nous introduisons maintenant le théorème de correction générale. Ce théorème permet de ga-

rantir la correction de tout type de mise jour contenues dans le correctif (instructions, méthodes et champs).

Théorème 2 ***(General soundness)*** *Étant donnée une classe C, l'ensemble de ses méthodes $Meth\_c$, les informations de typage de l'ensemble des méthodes F_s et S_s, un correctif Δ, ainsi que les informations de typage de la classe F^c, S^c et Fl, la mise à jour de la classe C selon le correctif Δ produit un programme bien typé si et seulement si :*

$$
\begin{array}{l}
\forall m \in Meth\_c, \forall i \in 1...length(\Delta), \\
\Delta(i) = Add\_method(m, Im, BC, sig) \Rightarrow (S_m = \varepsilon) \wedge S_m, init(F_m), 0 \vdash BC \Rightarrow F^c, S^c, Fl \vdash C \quad \wedge \\
\Delta(i) = Del\_method(m, Im, BC, sig) \Rightarrow (F_s \leftarrow F_s \setminus \{F_m\} \wedge S_s \leftarrow S_s \setminus \{S_m\}) \Rightarrow F^c, S^c, Fl \vdash C \quad \wedge \\
\Delta(i) = Mod\_method(m, Im, BC, sig) \Rightarrow F_m, S_m \vdash BC \Rightarrow F^c, S^c, Fl \vdash C \quad \wedge \\
\Delta(i) = Add\_field(f, tf, val) \Rightarrow F^c, S^c, Fl \cup \{(f, tf)\} \vdash C \quad \wedge \\
\Delta(i) = Del\_field(f, tf) \Rightarrow F^c, S^c, Fl \setminus \{(f, tf)\} \vdash C \quad \wedge \\
\Delta(i) = Mod\_field(f, tf, val) \Rightarrow F^c, S^c, Fl \vdash C
\end{array}
$$

Dans ce théorème, tous les types des mises à jour pour une classe sont considérés. Les mises à jour sont contenues dans un correctif. La variable i est utilisée pour indiquer l'instruction de mise à jour en cours. Le nombre d'instructions de mise à jour dans le correctif est représenté par $length(\Delta)$. Quand il s'agit de la modification d'une méthode, nous considérons que les mises à jour relatives à la méthode sont spécifiées par Δ_m qui est une partie du correctif général Δ. Dans ce théorème, F_m et S_m expriment les informations de typage d'une méthode m. Elles sont initialisées aux valeurs par défaut et vide quand il s'agit d'insérer une nouvelle méthode.

Lors de la modification d'une méthode, son code doit être bien typé selon F_m et S_m, ceci résulte du théorème 1. Les opérations de mise à jour relatives aux champs sont obtenues directement des règles sémantiques. Ce théorème permet donc de garantir qu'un programme mis à jour ne va pas exécuter des operations interdites et que toutes les opérations sont correctes vis-à-vis du système de typage. Il implique que notre modèle est correct en montrant qu'aucun programme erroné n'est accepté.

VI.5 Vérification de comportement des programmes

Notre objectif, après avoir établi la sûreté de typage, est de garantir que l'application de la mise à jour aboutit à une spécification du programme conforme à la spécification attendue par l'utilisateur. Cette spécification est fournie initialement par le programmeur. Étant donné un programme initial $P1$, sa nouvelle version $P2$, et un correctif Δ contenant la spécification de la mise à jour obtenue à partir de $P1$ et $P2$, l'application du correctif sur la carte sur $P1$, notée App_PATCH permet d'obtenir un programme $P2'$. Les deux programmes $P2$ et $P2'$ doivent être sémantiquement équivalents. Cette équivalence, que nous formulerons dans le modèle de Hoare, permet de garantir que le système implémente la modification souhaitée par le programmeur. Ce problème peut être exprimé à l'aide de l'équation suivante :

$$\forall P1, P2, P2', \Delta = DIFF(P1, P2), P2' = App\_PATCH(P1, \Delta) \Rightarrow P2 \sim P2'$$

La démonstration de cette équivalence pose deux problèmes :

1. La modélisation de l'application du correctif sur un programme existant ;
2. L'expression de l'équivalence sémantique (notée $\sim$ dans l'équation) qui permet de garantir la correction de la mise à jour.

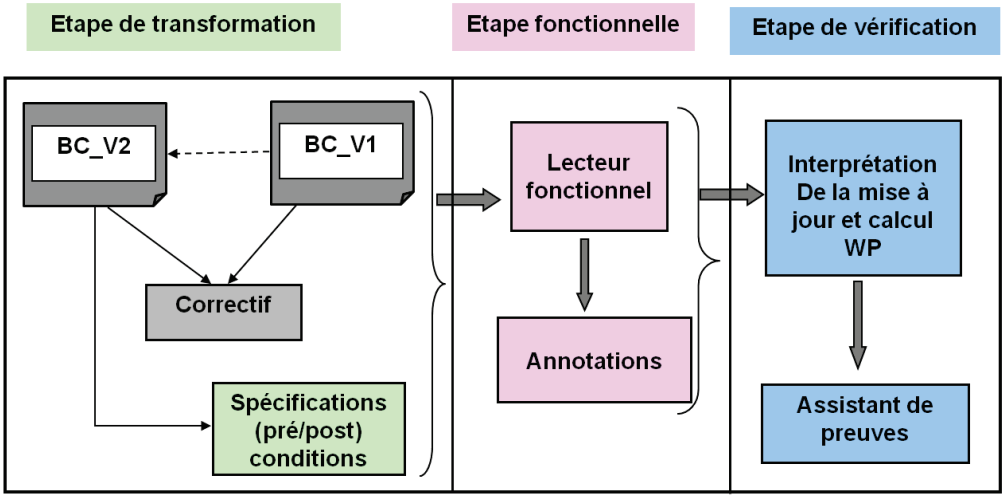

Figure VI.1 : Approche de vérification de correction de comportement

VI.5.1 Méthodologie

La Figure VI.1 illustre l'approche que nous proposons pour la vérification formelle des comportements des programmes mis à jour. L'approche est divisée en trois étapes :

i ***Étape de transformation :*** Dans cette étape, on obtient à partir d'une première version du code d'une application BC_V1 et d'une deuxième version BC_V2 (la première version modifiée), un correctif. Les versions BC_V1 et BC_V2 sont supposées correctes. Le correctif sera appliqué sur la première version qui s'exécute sur la carte. On obtient alors, sur la carte, une nouvelle version. Le but de notre approche est d'établir que la nouvelle version obtenue sur la carte et BC_V2 sont sémantiquement équivalentes. A cette étape, les spécifications des programmes BC_V1 et BC_V2 sont exprimées par le programmeur en utilisant des langages de spécifications existants.

ii ***Étape fonctionnelle :*** Un modèle fonctionnel est défini pour représenter et manipuler le programme bytecode Java Card. Nous implémentons un traducteur appelé *functional reader*. Ce lecteur fonctionnel prend un programme en langage bytecode et produit un modèle fonctionnel de celui-ci. L'application du correctif est représentée à cette étape par des annotations sur le modèle fonctionnel. Les annotations indiquent les opérations de la mise à jour et leurs emplacements dans le code.

iii ***Étape de vérification :*** L'objectif est de vérifier que le programme obtenu par la mise à jour est équivalent au programme écrit par le programmeur. La spécification du code obtenu dans sa représentation fonctionnelle avec les annotations est déduite par un calcul de plus faible précondition (WP : Weakest precondition) spécialement défini pour les opérations de mise à jour. Après ce calcul, un générateur de conditions de vérification permet d'obtenir les formules à prouver pour établir que la spécification obtenue est conforme à la spécification écrite par le programmeur lors de l'étape de transformation. Un assistant de preuve est alors utilisé pour la démonstration.

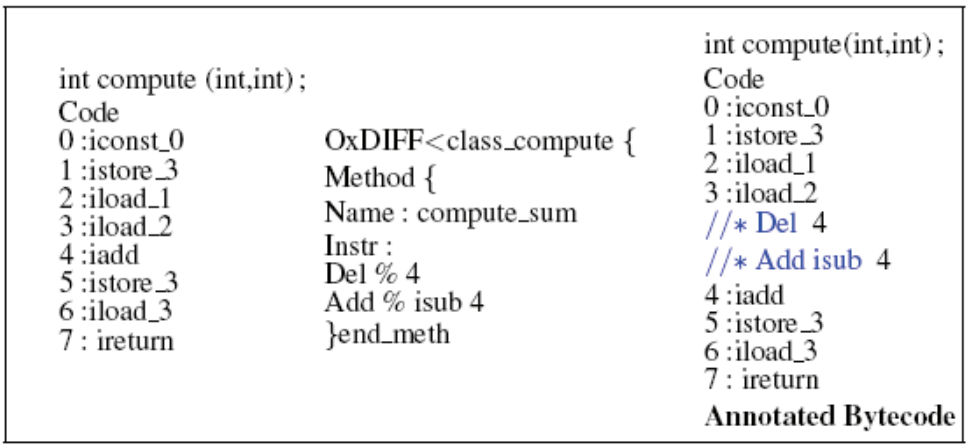

Figure VI.2 : Bytecode annoté par des instructions de mise à jour

VI.5.2 Annotation et représentation fonctionnelle du bytecode

Le correctif contenant les instructions de mise à jour est calculé sur les bytecodes des anciennes et nouvelles versions puis envoyé pour effectuer les mises à jour sur la carte. Dans le but de garantir la correction du correctif envoyé, nous modélisons l'application de celui-ci sur la version initiale sous la forme d'annotations. Le mécanisme d'annotation du bytecode avec des expressions indiquant où s'effectue la mise à jour et de quelle instruction de mise à jour il s'agit, est définie récursivement par une fonction *Annot* d'annotation qui transforme un programme en un programme annoté :

$$Annot([], P) = P$$
$$Annot(Upd_i :: \Delta, P) = let\ P' = Add\_Annot\_Line(Upd_i, P)\ in\ Annot(\Delta, P')$$

L'annotation d'un programme avec un correctif vide ([]) produit le programme lui même. La fonction effectue des itérations sur les instructions de mise à jour en commençant par l'opération en tête du correctif (Upd_i) et ajoute la ligne d'annotation correspondante ($Add\_Annot\_Line(Upd_i, P)$) au programme. Le reste des instructions de mise à jour (Δ) est ensuite considéré. Le symbole :: représente le constructeur de listes qui relie la tête de la liste au reste de celle-ci. La Figure VI.2 montre un programme annoté obtenue par l'application d'un correctif sur un code initial. Les annotations sont représentées par des commentaires spéciaux ; par exemple, *Del 4* : exprime la suppression de l'instruction au point (*pc*) 4 (program counter) du programme et *add isub 4*, insère l'instruction *isub* à l'emplacement *pc* 4.

Dans cette approche, nous définissons la représentation fonctionnelle avec le langage fonctionnel Ocaml pour représenter d'abord les données manipulées, instructions du langage, instructions de mise à jour, programme et programmes annotés. La représentation de ces concepts est utilisée pour calculer la spécification du programme mis à jour avec un calcul dédié. Ce calcul est présenté à la section suivante.

VI.5.3 Vérification par calcul WP

L'approche pour la vérification est basée sur l'idée suivante : la mise à jour d'un programme et donc de sa sémantique implique une modification de sa spécification. Dans la logique de Hoare ([77]), la spécification d'un programme $P1$ est représentée par le triplet $\{pre1\}P1\{post1\}$ où $pre1$ et ($post1$) représentent la precondition et postcondition du programme $P1$. Un triplet de Hoare est

dit valide, noté $\models \{pre1\}P1\{post1\}$, si à partir d'un état satisfaisant la précondition, le programme termine et aboutit à la postcondition. Soit $P2$ la nouvelle version de $P1$. La spécification de la nouvelle version est écrite en off-card et est représenté par le triplet $\{pre2\}P2\{post2\}$. Ce triplet est dit triplet cible : la spécification du programme obtenu sur la carte, sur la base du correctif calculé à partir $P1$ et $P2$, doit correspondre à ce triplet écrit en off-card. Pour ce faire, nous introduisons d'abord un ensemble de définitions relatives à la notion de correction et nous proposons un calcul de transformation de prédicats pour l'établir.

VI.5.3.1 Interprétation de la mise à jour

Dans le but de définir formellement l'interpréteur de la mise à jour, nous présentons quelques notions. Dans cette interprétation, un état est représenté par un triplet $< Heap, Frame, Stk\_Frame >$ où : $Heap$ représente le contenu du tas, $Frame$, c'est le cadre d'activation qui représente l'état d'exécution de la méthode en cours et Stk_Frame la liste des cadres d'activation correspondant à la pile des appels. Un cadre d'activation contient les éléments suivants : une pile d'opérandes $OperandStack$ et les valeurs des variables locales $LocalVar$ à un point PC d'une méthode $Method$. Le quintuplet : $< Method, OperandStack, LocalVar, Heap, PC >$ représente un état pour l'exécution d'une méthode. La définition de l'interprétation de la mise à jour est basée sur la notion d'étape.

Définition2 *(**Étape**) La sémantique d'une instruction (ou d'une instruction de mise à jour) est spécifiée comme une fonction :*

$$étape : Bytecode\_Prog * State * Specification -> State * StepName * Specification$$

qui, étant donné une programme BC, une état S, et une spécification SP , calcule le prochain état S', le nom de la prochaine étape et la nouvelle spécification.

Définition3 *(**Interpréteur de la mise à jour**) Nous définissons un interpréteur de la mise à jour (Upd_int) qui itère sur les étapes. Il prend en paramètre un programme annoté dans sa représentation fonctionnelle, un état initial et une spécification initiale. Il repose sur un calcul de prédicats et une fonction d'interprétation et produit un nouvel état et une nouvelle spécification. L'interpréteur est défini comme $Upd\_int(BC, S) = (S', Sp')$, tel que $S = initial(BC, Sp)$ représente une fonction définissant l'état initial pour l'exécution du programme BC avec une spécification initiale Sp. Le programme BC est donné avec ses paramètres et un tas initial. Le résultat de l'interpréteur est un état S' et une nouvelle spécification Sp'.*

Définition4 *(**Programme mis à jour vérifié**)*

- *soit $P1$ et $P2$ la première et la nouvelle version d'un programme et P un correctif ;*
- *soit $P2' = annot(P1, P)$ le programme obtenu par annotation de $P1$ avec P ;*
- *soit $f(P2')$ la représentation fonctionnelle de $P2'$;*
- *soit $spec(P1) = (pre1, post1)$ la spécification de $P1$ et $spec(P2) = (pre2, post2)$ la spécification de $P2$,*

Le programme $P2'$ est une mise à jour vérifiée de $P1$ si et seulement si : $verif(spec(P2), spec(P2'))$ est satisfaite tel que $spec(P2')$ est obtenue par une transformation de prédicats sur $f(P2')$ en partant de $post2$.

Tableau VI.10 : Règles de plus faible précondition pour les opérations de mise à jour

wp(Add_instr(pop,i)) $= (shift\_exp^2(@E_i))$
wp(Add_instr(store x,i)) $= shift\_exp^2(@E_i)(S(0)/x)$
wp(Add_instr(if L,i)) $= ((\neg S(0) = 0) \Rightarrow shift\_exp^2(E_L)) \wedge ((S(0) \neq 0) \Rightarrow shift\_exp^2(@E_i))$
wp(Add_instr(load x,i)) $= unshift\_exp(shift\_exp(@E_i))(x/S(0))$
wp(Add_instr(const a,i)) $= unshift\_exp(shift\_exp(@E_i))(a/S(0))$
wp(Add_instr (new A,i)) $=$ $unshift\_exp(shift\_exp(@E_i[create(H, A)/S(0), A :: H/H])$
wp(Add_instr(add,i)) $= (shift\_exp^2(@E_i))[(s(1) + S(0))/S(1)]$
wp(Add_instr(neg,i)) $= (shift\_exp(@E_i))[-S(0)/S(0)]$
wp(Add_instr (getfield a f t,i)) $=$ $shift\_exp(@E_i[(val(S(0), f))/S(0)]) \wedge S(0) \neq null$
wp(Add_instr(putfield a f t,i)) $=$ $(shift\_exp^3(@E_i))[H(val(S(0), f)) := S(1))/H] \wedge S(0) \neq null$
wp(Add_instr(goto L,i)) $= shift\_exp(E_L)$

Le prédicat $verif(spec(P2), spec(P2'))$ signifie que les deux spécifications sont sémantiquement équivalentes. La notion d'équivalence repose sur le concept de plus faible précondition.

VI.5.3.2 Le calcul de plus faible précondition

a) Principe du calcul

Le calcul de plus faible précondition est développé par Dijkstra [81]. C'est une approche de calcul ascendant sur les assertions. Il propose en partant d'une postcondition, de trouver la plus faible précondition (Weakest Precondition) qui vérifie le triplet de Hoare. Ainsi, on raisonne toujours sur des triplets, mais les règles sont établies dans l'autre sens dans un calcul ascendant.

Soient A et B deux formules logiques. On dit que A est plus fort que B et inversement B est plus faible que A, Si $A \Rightarrow B$. Soit l'ensemble des formules suivant : $\{A_1, A_2, A_3 \ldots\}$. La formule A_i est la plus faible formule de l'ensemble si $A_j \Rightarrow A_i$ pour toute formule A_j de l'ensemble.

Pour un programme S et une formule Q, $WP(S, Q)$, la plus faible précondition pour S et Q est la plus faible formule P tel que :$\models \{P\}S\{Q\}$. Un programme S est correct par rapport à une précondition P et une postcondition Q si P implique la plus faible précondition pour S et Q :

$$\models \{P\}S\{Q\} \Leftrightarrow \models P \Rightarrow WP(S, Q)$$

Nous proposons dans cette section un calcul de plus faible précondition pour les opérations de mise à jour des programmes en bytecode. Dans le but d'établir l'équivalence sémantique du code écrit par le programmeur et celui obtenu par l'application du correctif, nous vérifions que la plus faible précondition du programme annoté obtenue par le calcul WP est impliqué par la précondition donnée par le programmeur.

b) Fonctions et notations utilisées

Le calcul WP que nous proposons est inspiré de [124]. Nous considérons que chacune des instructions et des instructions de mise à jour possède une précondition. Une instruction avec sa précondition est appelée instruction spécifiée et est notée $E_i : I_i$ où : I_i représente l'instruction et E_i sa spécification. La notation exprime que la précondition E_i est vérifiée quand le compteur du programme est à l'adresse i. Les éléments de la pile d'opérandes S sont indexés par des entiers positifs, le sommet de la pile correspond à l'indice 0. Nous utilisons deux fonctions : $shift\_exp$ et $unshift\_exp$ pour représenter par des décalages dus à l'effet d'empiler et dépiler des éléments sur (de) la pile S, et l'effet de décaler une expression par rapport aux éléments de la pile due à l'insertion d'instructions. Elles sont définies par :

$$shift\_exp(Exp) = Exp[S(i+1)/S(i)\ forall\ i \in N]$$
$$unshift\_exp = shift\_exp^{-1}$$

La notation $s(i+1)/s(i)$ signifie que l'élément $S(i+1)$ remplace l'élément $S(i)$ dans l'expression Exp. Nous utilisons également le symbole @ pour exprimer l'ancienne spécification associée à une position i : quand une instruction est ajoutée à une position i, le programme ainsi que les spécifications sont décalés à partir de la position i puis une nouvelle instruction est insérée. Sa précondition est calculé avec la spécification de l'instruction qui était à la position i avant la mise à jour.

c) Règles du calcul

Le Tableau VI.10 montre les règles de calcul WP pour les opérations de mise à jour, plus précisément, l'insertion de nouvelles instructions. Dans les règles, pour les instructions *store* x, *load* x, *pop*, *const* a et *add* (qui est un cas particulier de *binop*), la précondition est obtenue, comme dans une affectation dans la logique de Hoare en substituant la partie droite par la partie gauche, qui est soit une variable soit le sommet de la pile, dans la postcondition. La precondition d'une instruction *store* x sous une postcondition E_{i+1} (la precondition de l'instruction suivante) est donnée par : $shift\_exp(E_{i+1})(S(0)/x)$. Ce qui signifie que si l'expression E est vérifiée après l'exécution de *store* x, alors elle est aussi vérifiée pour le sommet de la pile avant de le sauvegarder dans x. La fonction $shift\_exp$ est utilisée pour exprimer que avant l'exécution de l'instruction, le sommet de la pile correspondant à l'instruction à $i+1$ était à l'indice i.

L'insertion d'une instruction, par exemple *store* x à la ligne i signifie que la précondition de l'ancienne instruction à i devient la postcondition de l'instruction insérée et donc, la précondition calculée commence par l'ancienne postcondition ($@E_i$). La fonction $shift\_exp$ est utilisée deux fois ($shift\_exp^2$) pour exprimer aussi l'impact du à l'insertion de l'instruction sur la spécification des instructions suivantes.

Les instructions *new*, $putfield$ et $getfield$ sont des instructions manipulant le tas. La fonction *create* utilisée dans l'instruction *new* A retourne un nouvel objet de type A dans le tas H. Ce tas obtenu ($A :: H$) remplace l'ancien tas. La fonction *val* utilisée dans la définition de $getfield$ pour obtenir la valeur du champs f de la classe a à partir de la référence(sommet de la pile). Cette valeur est alors empilée sur la pile. Cette instruction dépile un élément et empile une valeur, de ce fait, seul le décalage de l'insertion est appliqué (la même explication s'applique à l'instruction *neg*). Dans $putfield$, la valeur du champ désigné par le sommet de la pile est mise à jour avec la valeur se trouvant après le sommet de la pile. L'insertion de cette instruction qui dépile deux valeurs implique trois applications de la fonction $shift\_exp$.

Le calcul est dirigé vers la specification se trouvant à l'adresse L dans l' instructions *goto* L et le premier cas de $if\ L$. Dans le deuxième cas du $if\ L$, le calcul s'effectue avec la specification

de l'instruction suivante. Pour l'instruction $If\ L$, la fonction shift est appliquée deux fois car un élément est dépilé de la pile.

VI.5.4 Implémentation du calcul

L'implémentation de l'approche proposée repose sur la définition d'un certain nombre de types de données permettant de représenter les différents concepts manipulés par le calcul :

- La définition des données manipulées, instructions et instructions de mise à jour est basée principalement sur la structure de liste pour les programmes et la fonction d'annotation. La Figure VI.3 illustre un extrait en Ocaml représentant les structures de données principales. L'extrait commence par les types des instructions en bytecode (*instr*) et les instructions de mise à jour (*upd_instr*). La définition d'une instruction est donnée par le nom du constructeur pour les instructions de bytecode et le constructeur suivi de deux arguments pour les instructions de mise à jour. Les deux arguments représentent le nom de l'instruction à ajouter ou supprimer ainsi qu'un entier représentant l'adresse de l'ajout ou de la suppression. Les arguments des différentes instructions en bytecode sont représentés par le type *arg* qui est soit un argument numérique soit un argument chaîne de caractères pour les noms des classes, des méthodes et des champs. Une ligne d'un programme bytecode (*lineBC*) est un triplet composé d'un numéro, d'une instruction et d'une liste d'arguments. Un programme en bytecode est une liste de lignes (*lineBC*). Une ligne annotée est basée sur les instructions de mise à jour. Une ligne dans un code annoté est soit une ligne standard (*Std*), soit une ligne d'annotation (*Annoted*). Un code annoté est une liste de lignes annotées.
- L'implémentation du calcul WP est basée principalement sur la notion de piles. Deux catégories de piles sont utilisées : une pile d'opérandes et une pile logique. La première pile est utilisée pour contenir les arguments manipulées par les instructions du programme. Le deuxième type représente une pile contenant des expressions logiques. A cet effet, nous avons défini un type pile polymorphe (*'a stack*) pour représenter ces deux types en spécifiant à chaque fois ce que représente le type polymorphe $'a$: un entier ou bien une expression logique. Le type des expressions logique *logic_Exp* est utilisé pour représenter les spécifications (les expressions E_i). Les expressions de la pile logique sont reliées au contenu de la pile d'opérande et aux instructions du programme. Une modification à travers une instruction de mise à jour possède un impact sur la pile logique. Cet impact est définie en utilisant les fonctions $shift$ et $unshift$ qui calculent récursivement sur la pile logique en se basant sur les règles WP. La correspondance entre la pile d'opérandes, la liste des instructions et la pile des spécifications est établie par une notion d'identifiant d'instruction.

Exemple VI.2

Dans le but d'illustrer le fonctionnement de la logique définie, considérons l'exemple de la fonction *abs* qui retourne la valeur absolue d'une entier pris en argument. Cette fonction est mise à jour dans le but de calculer le double de la valeur absolue : pour chaque entier pris en argument, la nouvelle version retourne la valeur absolue multipliée par deux (modified abs). Les spécifications des deux fonctions sont respectivement :

$\{p = P\}\ abs\ \{(P \geq 0 \rightarrow result = P) \wedge (P < 0 \rightarrow result = -P)\}$
$\{p = P\}\ modified\ abs\ \{(P \geq 0 \rightarrow result = 2 * P) \wedge (P < 0 \rightarrow result = -2 * P)\}$

Dans la spécification, P dénotes la valeur logique de l'entrée et $result$ le résultat de la fonction. La Figure VI.4 montre le code en bytecode de la première version (a) et de la seconde version (b) de la fonction. La partie (c) de la figure montre le fichier DIFF généré à partir des deux versions.

```
type instr =
| Pop
| Store
| Load
| Const
| Binop
| Neg
| IfL
| Goto
| New
| Getfield
| Putfield
| Invoke
| Return;;

type upd_instr =
| Add_instr of int * instr
| Del_instr of int * instr;;

type arg = ArgN of int | ArgS of string;;
type lineBC = Line of (int*instr*arg list);;
type bc = lineBC list;;
type annot_line = AnnotL of (int * upd_instr);;
type annot_code_line =
| Std of lineBC
| Annoted of annot_line;;
type annot_code = annot_code_line list;;
type diff = upd_instr list;;

type 'a stack =
| Vide
| Emp of 'a * 'a stack;;
type logic_Exp=
| True
| False
| Atom1 of (int → bool) * int
| Atom2 of (int stack * int → bool) * int stack * int
| Ou of logic_Exp * logic_Exp
| Et of logic_Exp * logic_Exp
| Non of logic_Exp;;
```

Figure VI.3 : Les type de données utilisés dans le calcul WP

La partie (d) représente le bytecode de la fonction *abs* annoté avec des instructions de mise à jour. Notons que dans ce code, les variables locales sont représentée par des entiers : dans *load* 1 par exemple, le nombre 1 signifie la variable locale 1. La même signification s'applique aux autres variables locales.

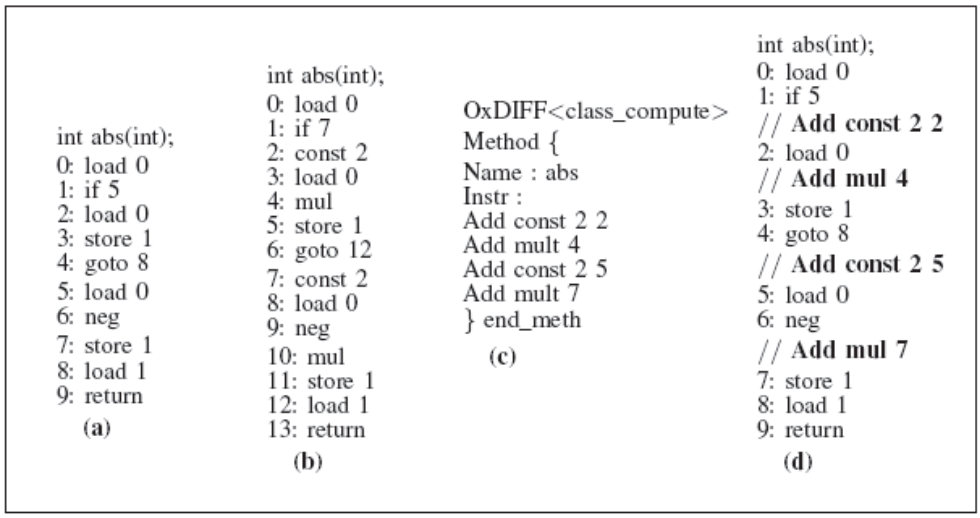

Figure VI.4 : Exemple d'un bytecode annoté

Dans la Figure VI.5, la calcul WP est effectué sur le programme sans annotation en commençant par la post condition de la nouvelle version. Le calcul WP est appliqué ensuite au programme annoté (Figure VI.6). Les spécifications pour les instructions de mise à jour sont en gras. Cette exemple montre que nous obtenons la même precondition $\{P = v0\}$ ce qui signifie que au début du calcul, la valeur logique P est dans la première variable locale de la fonction. Ce résultat exprime l'équivalence des deux programme vis-à-vis de la définition des programmes mis à jour vérifiés. Ceci garantit la correction du comportement selon le correctif.

int abs(int);
0: load 0 $\{(P = v0\}$
1: if 7 $\{(P \geq 0 \rightarrow 2 * v0 = 2 * P) \wedge (P < 0 \rightarrow 2 * v0 = -2 * P)\ \}$
2: const 2 $\{(P \geq 0 \rightarrow 2 * v0 = 2 * P) \wedge (P < 0 \rightarrow 2 * v0 = -2 * P)\ \}$
3: load 0 $\{(P \geq 0 \rightarrow s(0) * v0 = 2 * P) \wedge (P < 0 \rightarrow s(0) * v0 = -2 * P)\ \}$
4: mul $\{(P \geq 0 \rightarrow s(1) * s(0) = 2 * P) \wedge (P < 0 \rightarrow s(1) * s(0) = -2 * P)\ \}$
5: store 1 $\{(P \geq 0 \rightarrow s(0) = 2 * P) \wedge (P < 0 \rightarrow s(0) = -2 * P)\ \}$
6: goto 12 $\{(P \geq 0 \rightarrow v1 = 2 * P) \wedge (P < 0 \rightarrow v1 = -2 * P)\ \}$
7: const 2 $\{(P \geq 0 \rightarrow 2 * (-v0) = 2 * P) \wedge (P < 0 \rightarrow 2 * (-v0) = -2 * P)\ \}$
8: load 0 $\{(P \geq 0 \rightarrow s(0) * (-v0) = 2 * P) \wedge (P < 0 \rightarrow s(0) * (-v0) = -2 * P)\ \}$
9: neg $\{(P \geq 0 \rightarrow s(1) * (-(s0)) = 2 * P) \wedge (P < 0 \rightarrow s(1) * (-(s0)) = -2 * P)\ \}$
10: mul $\{(P \geq 0 \rightarrow s(1) * s(0) = 2 * P) \wedge (P < 0 \rightarrow s(1) * s(0) = -2 * P)\ \}$
11: store 1 $\{(P \geq 0 \rightarrow s(0) = 2 * P) \wedge (P < 0 \rightarrow s(0) = -2 * P)\ \}$
12: load 1 $\{(P \geq 0 \rightarrow v1 = 2 * P) \wedge (P < 0 \rightarrow v1 = -2 * P)\ \}$
13: return $\{(P \geq 0 \rightarrow result = 2 * P) \wedge (P < 0 \rightarrow result = -2 * P)\ \}$

Figure VI.5 : Calcul WP sur la fonction modifiée

int abs(int);
0: load 0 $\{(P = v0\}$
1: if 5 $\{(P \geq 0 \rightarrow 2 * v0 = 2 * P) \wedge (P < 0 \rightarrow 2 * v0 = -2 * P)\ \}$
// **Add const 2 2** $\{(\mathbf{P \geq 0 \rightarrow 2*v0 = 2*P}) \wedge (\mathbf{P < 0 \rightarrow 2*v0 = -2*P})\ \}$
2: load 0 $\{(P \geq 0 \rightarrow s(1) * v0 = 2 * P) \wedge (P < 0 \rightarrow s(1) * v0 = -2 * P)\ \}$
// **Add mul 4** $\{(\mathbf{P \geq 0 \rightarrow s(2)*s(1) = 2*P}) \wedge\ (\mathbf{P < 0 \rightarrow s(2)*s(1) = -2*P})\ \}$
3: store 1 $\{(P \geq 0 \rightarrow s(0) = 2 * P) \wedge (P < 0 \rightarrow s(0) = -2 * P)\ \}$
4: goto 8 $\{(P \geq 0 \rightarrow v1 = 2 * P) \wedge (P < 0 \rightarrow v1 = -2 * P)\ \}$
// **Add const 2 5** $\{(\mathbf{P \geq 0 \rightarrow 2*(-v0) = 2*P}) \wedge (\mathbf{P < 0 \rightarrow 2*(-v0) = -2*P})\ \}$
5: load 0 $\{(P \geq 0 \rightarrow s(1) * (-v0) = 2 * P) \wedge (P < 0 \rightarrow s(1) * (-v0) = -2 * P)\ \}$
6: neg $\{(P \geq 0 \rightarrow s(1) * (-s(0)) = 2 * P) \wedge (P < 0 \rightarrow s(1) * (-s(0)) = -2 * P)\ \}$
// **Add mul 7** $\{(\mathbf{P \geq 0 \rightarrow s(2)*s(1) = 2*P}) \wedge\ (\mathbf{P < 0 \rightarrow s(2)*s(1) = -2*P})\ \}$
7: store 1 $\{(P \geq 0 \rightarrow s(0) = 2 * P) \wedge (P < 0 \rightarrow s(0) = -2 * P)\ \}$
8: load 1 $\{(P \geq 0 \rightarrow v1 = 2 * P) \wedge (P < 0 \rightarrow v1 = -2 * P)\ \}$
9: return $\{(P \geq 0 \rightarrow result = 2 * P) \wedge (P < 0 \rightarrow result = -2 * P)\ \}$

Figure VI.6 : Calcul WP sur un bytecode annoté

VI.6 Conclusion

Nous avons présenté dans ce chapitre nos contributions concernant la correction de la mise à jour du code. Nous nous sommes d'abord intéressés à établir la propriété de la sûreté du typage des programmes mis à jour. Pour ce faire, nous avons d'abord présenté un sous langage du bytecode Java Card. Nous avons proposé une sémantique formelle des opérations de mise à jour. Cette formalisation permet d'énoncer pour chaque opération de mise à jour, les conditions permettant d'en assurer la validité vis-à-vis du bon typage des programmes Java Card. La formalisation nous a permis d'énoncer et de démontrer la propriété de sûreté de typage pour les programmes mis à jour.

Nous nous somme ensuite intéressés à la correction du comportement des programmes mis-à-jour vis-à-vis des spécifications de l'utilisateur. Pour cela, nous avons proposé une démarche permettant de calculer, à partir d'un programme initial et d'un fichier DIFF, une spécification pour la nouvelle version du programme. Cette spécification est ensuite comparée avec la spécification fournie par le programmeur dans le but d'établir la propriété de correction via une équivalence sémantique. Cette partie est basée principalement sur la représentation des mises à jour par des annotations et la définition d'un nouveau calcul de plus faible précondition destiné à prendre en compte les opérations de mise-à-jour.

Concernant les travaux relatifs à l'application des méthodes formelles à la plateforme Java Card, notre première contribution s'inscrit dans le cadre des approches de vérification du bon typage du bytecode. Elle est de ce fait proche des travaux comme [112, 137] dans l'utilisation des informations de typage pour l'analyse des programmes en bytecode. Elle présente l'apport de formalisation d'un nouvel aspect des opérations de la machine virtuelle. La sémantique proposée constitue une base pour des approches de vérification adaptées à la nouvelle fonctionnalité implémentée par la machine virtuelle à travers le système de DSU.

La deuxième partie relative au calcul de plus faible précondition s'inscrit dans la vérification du comportement des programmes au niveau du bytecode. La contribution présente des points communs avec les travaux de ce domaine [124, 125] pour l'approche d'une vérification basée sur un calcul de plus faible précondition sur des programmes en bytecode. Le calcul que nous proposons est adapté aux spécificités du processus de mise à jour qui s'applique à partir d'une version initiale d'un programme et d'un fichier DIFF contenant les informations détaillées sur les opérations de

mise à jour.

Nos propositions répondent aux spécificités du fonctionnement du système EmbedDSU mais sont exploitables par d'autre travaux. Les approches proposées peuvent servir de base pour la vérifications des mises à jour au niveau code pour d'autres systèmes utilisant le langage bytecode et fonctionnant avec le principe de l'envoi des fichiers DIFF. Par ailleurs, le calcul de plus faible précondition peut être utilisé pour analyser l'impact d'injection de code dans des application à des fins de sécurité.

Chapitre VII

Correction de la recherche des points sûrs

Sommaire

Dans le chapitre précédent, nous avons établi des propriétés de correction de mise-à-jour du code. Nous nous intéressons dans ce chapitre à la correction d'un deuxième aspect de la DSU : il s'agit de la correction de la recherche de points sûrs de mise à jour (Safe Update Points : SUP). Nous commençons par présenter les critères de correction concernant les points sûrs de mise à jour. Ensuite, nous introduisons les concepts et outils utilisés, à savoir, le model checking et la logique temporelle linéaire (LTL : Linear Temporal Logic). Nous détaillons par la suite la description du module de recherche de SUP du système EmbedDSU à travers les différents modes de la machine virtuelle, les principales fonctions et la modélisation du langage cible. Nous présentons ensuite la spécification et la vérification des propriétés de correction. La contribution présentée dans ce chapitre est publiée dans le papier en référence [138].

VII.1 Motivations

La recherche des points sûrs auxquels les mises à jour dynamiques peuvent être effectuées représente une tâche cruciale dans les solutions DSU. En effet, une mise à jour effectuée à des points hasardeux conduit le système à des comportements erronés ou à un arrêt de service. Le plus grand défi des différentes techniques proposées est de trouver un compromis entre une application rapide et la garantie de la consistance de la mise à jour. Le système de recherche de points sûrs dans EmbedDSU repose sur le principe suivant : la mise à jour est appliquée à des états dits quiescents. Un état est quiescent si aucune méthode proposée à la mise à jour n'est active. Un tel état définit un point sûr pour la mise à jour. La machine virtuelle Java Card est modifiée pour amener le système, quand une requête de mise à jour est disponible, à atteindre un état quiescent. Ceci est réalisé principalement en bloquant les appels des threads qui n'ont plus de méthodes à mettre

à jour dans la pile des threads Java Card. Ceci dans le but d'éviter d'avoir plus de méthodes à mettre à jour en exécution. Un thread ayant des méthodes à mettre à jour dans sa pile continue son exécution jusqu'à ce que ces méthodes terminent. Le thread est alors bloqué. Ce mécanisme, introduit au chapitre EmbedDSU, sera détaillé dans ce chapitre. En effet le fonctionnement soulève les questions suivantes :

- Comment s'assurer que l'application du mécanisme de recherche SUP n'introduit pas des inter-blocages dans le système ?
- Comment garantir que la mise à jour s'effectue toujours à des états quiescents ?
- Comment garantir que le système atteindra les points sûrs de mise à jour ?

Dans ce chapitre, nous contribuons à répondre à ces problèmes en proposant d'utiliser la vérification formelle pour établir les critères de correction. Nous établirons la correction de la recherche de SUP dans EmbedDSU à travers les critères suivants :

- **L'absence d'inter-blocage :** Cette propriété permet de s'assurer que l'application du principe de fonctionnement du module de recherche de SUP du système EmbedDSU n'introduit pas d'inter blocages dans le système.
- **Sûreté d'activation :** Cette propriété nous permet d'établir que lors de l'application de la mise à jour, aucune méthode concernée par la mise à jour n'est en cours d'exécution.
- **Garantie de la mise à jour :** Cette propriété nous permet d'établir que le mécanisme de recherche de SUP de EmbedDSU aboutit à un état quiescent permettant d'effectuer la mise à jour à chaque fois que celle-ci est requise.

Le "model checking" nous semble approprié pour établir la correction de ces critères. En effet, notre système est représenté par un système d'états-transitions permet de représenter toutes les exécutions possibles tout en étant adéquat à la vérification de propriétés temporelles. La section suivante présente les outils et concepts utilisés pour cette contribution.

VII.2 Notions et outils

VII.2.1 La logique temporelle linéaire

Les logiques temporelles [139] permettent d'exprimer des propriétés sur les états/transitions passés, présents ou futurs qu'un système peut/doit atteindre. A cet effet, elles offrent des opérateurs temporels spécifiques, des modalités, tels que finalement ou jamais, qui permettent de décrire des ordres entre les évènements/états sans pour autant introduire le temps explicitement. Pour ces logiques, le temps est une donnée abstraite qui n'est pas quantifiée par opposition aux logiques temporelles temporisées qui permettent de manipuler explicitement le temps. D'un point de vue plus théorique, on peut distinguer deux grandes familles de logiques temporelles. Ces deux familles, arborescentes et linéaires, sont distinguées par la nature de la relation d'ordre associée au temps que l'on considère. Cet ordre capture l'antériorité temporelle entre les états/événements et :

- Si l'ordre est total, le comportement du système est vu comme un ensemble de séquences d'exécution et la spécification se fait via les logiques temporelles linéaires. La logique LTL (Linear Time Logic) constitue un exemple de ces logiques.
- Si l'ordre est partiel, le comportement du système est vu comme un graphe et la spécification se fait via les logiques temporelles arborescentes. La logique CTL (Conditional/Computer Tree Logic) constitue un exemple de ces logiques.

La Figure VII.1 représente les principaux opérateurs de la LTL :

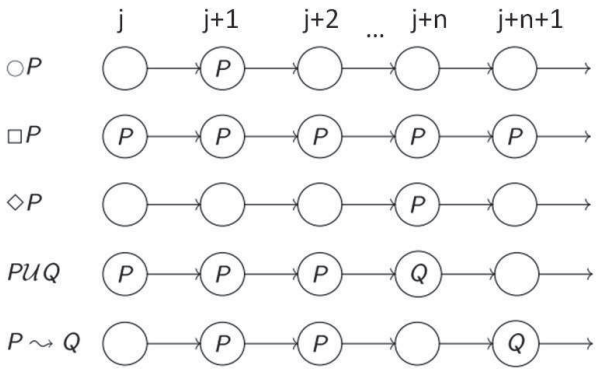

Figure VII.1 : Quelques opérateurs de la logique temporelle

- L'opérateur toujours (always) : l'opérateur toujours (représenté par $\Box$) exprime qu'une propriété P est satisfaite par tout les suffixes d'une exécution, c'est-à-dire à chaque instant. La formule $\Box P$ est vraie à la position j si et seulement si P est vraie pour toute position supérieure ou égale à j.
- L'opérateur prochain (next) : cet opérateur (représenté par $\circ$) exprime qu'une propriété P va être satisfaite à l'instant suivant. La formule $\circ P$ est vraie à la position j si et seulement si P est vraie à la position $j+1$.
- L'opérateur éventuellement (eventually) : l'opérateur éventuellement (représenté par $\diamond$) exprime que la propriété P sera satisfaite au bout d'un temps fini. La formule $\diamond P$ est vraie à la position j si et seulement si P est vraie en une certaine position k supérieure ou égale à j.
- L'opérateur jusqu'à (until) : l'opérateur jusqu'à (représenté par U) exprime que la propriété P sera satisfaite jusqu'à ce que la propriété Q deviennent vraie. La formule PUQ est vraie à la position j si et seulement si P est vraie jusqu'à un instant $j+n > j$ où la formule Q devient vraie.
- L'opérateur mène à (leads to) : l'opérateur jusqu'a (représenté par $\rightsquigarrow$) exprime que la propriété P est vraie alors la propriété Q sera vraie plus tard. La formule $P \rightsquigarrow Q$ est vraie à la position j si et seulement si P est vraie à j et il existe un instant dans le futur ou Q sera vraie. Entre temps, P peut devenir fausse.

VII.2.2 Outils et méthodologie

VII.2.2.1 Le langage PROMELA

PROMELA ([140]) est un langage de spécification de systèmes parallèles asynchrones. Il est basé sur les concepts de processus, de canaux de communication servant à transmettre des messages et de variables. Il permet de décrire des systèmes concurrents, en particulier des protocoles de communication. Il autorise la création dynamique de processus. Les communications peuvent s'effectuer de différentes manières : partage de variables globales, envoi et réception de messages via des canaux de communication. Ces derniers peuvent être définis pour réaliser des communications tant synchrones qu'asynchrones. Les interactions entre processus concurrents peuvent créer de nombreux problèmes tels que des inter-blocages et des modifications inattendues de valeurs de variables. La vérification de la correction d'un système décrit en PROMELA peut être réalisée grâce à l'outil SPIN.

Pour la génération des modèles PROMELA, nous utiliserons l'outil Modex. Modex (Model Extractor) ([141]) est un outil qui permet l'extraction automatique d'un modèle PROMELA à partir d'un code C. PROMELA est un langage de spécification, inspiré du C, dans lequel les propriétés à

vérifier sont exprimées à travers la logique temporelle LTL. Dans Modex, l'utilisateur peut générer un modèle PROMELA en se basant sur une table de règles fournie par défaut. Cette table peut être enrichie par l'utilisateur afin de générer un modèle plus précis.

VII.2.2.2 L'outil SPIN

Un "model checker", l'outil logiciel qui effectue la vérification de modèles, examine tous les scénarios possibles du système de manière systématique. De cette façon, il peut être démontré qu'un modèle d'un système donné répond véritablement à certaines propriétés. Il existe beaucoup d'outils puissants de model checking comme NuSMV, BLAST, SMV et SPIN. Dans ce travail nous adoptons le model checker SPIN.

SPIN ([140]) est un model checker basé sur l'approche automate développé par Gerard J. Holzmann pour vérifier les protocoles de communication. Il a été largement répandu dans le domaine académique ainsi que dans les industries qui construisent des systèmes critiques. Les modèles devant être vérifiés par SPIN sont écrits en PROMELA, un langage qui décrit le comportement du système. Les propriétés à vérifier doivent être formulées en Logique Temporelle Linéaire (LTL).

Étant données la spécification formelle du comportement du système et la spécification d'une propriété en logique temporelle, SPIN vérifie si cette propriété est satisfaite. En effet, le mode de vérification de SPIN détermine si le modèle proposé en PROMELA satisfait ou non une propriété LTL. Ceci est effectué en menant, si nécessaire, une vérification exhaustive *i.e.* en parcourant l'ensemble des exécutions possibles du système.

Le principe de vérification de SPIN est qu'il transforme, à partir des propriétés spécifiées, leurs négations en automates de Büchi. SPIN calcule ensuite le produit synchronisé de cet automate avec celui du système. Le résultat est encore un automate de Büchi dont on vérifie si le langage est vide ou non. S'il est vide, alors il n'existe aucune exécution violant la formule de départ, sinon il accepte au moins une exécution (ou mot du langage de l'automate, chaque lettre du mot étant représentée par une action du système ou de la propriété).

Un automate de Büchi accepte un mot (et donc son langage est non-vide) si et seulement si l'exécution du système passe infiniment souvent par un ou plusieurs des états acceptants de l'automate. *i.e.* qu'il existe un cycle accessible depuis l'état initial qui comporte un état acceptant. Pour prouver que toutes les exécutions du système sont valides vis-à-vis de la propriété à vérifier, il suffit de prouver qu'il n'existe pas de cycle acceptant accessible depuis l'état initial. Si ce mot existe, il va permettre de trouver un ensemble de transitions des automates du système qui viole la propriété. Un mot qui viole la propriété s'appelle un contre-exemple.

L'ensemble des transitions pour lesquelles la propriété n'est pas vérifiée forment les contre-exemples qui peuvent être générés par SPIN. Ces contre-exemples sont censés permettre à l'utilisateur de modifier par la suite son modèle (ou la propriété si elle a été mal exprimée), afin de corriger les erreurs éventuelles. La méthode utilisée, pour trouver un contre-exemple, est une recherche en profondeur imbriquée (Nested Depth-First Search) : un premier DFS est lancé de façon à trouver tous les états acceptants accessibles à partir de l'état initial. Ensuite, l'algorithme cherche, en relançant un DFS (dit nested, ou imbriqué) à partir de ces états.

VII.2.2.3 Méthodologie

Les outils décrits s'articulent autour de la démarche illustrée par la Figure VII.2. La figure illustre qu'initialement, nous disposons d'un code en C représentant le module de recherche de SUP du système EmbedDSU. A partir de ce code, nous utiliserons l'outil Modex pour extraire le modèle PROMELA correspondant au programme. Ce modèle sera ensuite soumis à l'outil SPIN avec les propriétés qu'on souhaite vérifier. Dans le cas ou la propriété est satisfaite, l'outil retourne un

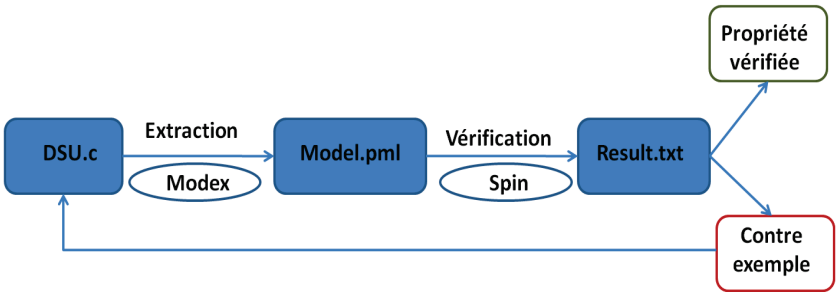

Figure VII.2 : Méthodologie et outils

message de succès de vérification. Dans le cas où la propriété n'est pas vérifiée, un contre example est montré.

La section suivante présente la modélisation de notre système. Elle sera suivie de la vérification formelle des propriétés de correction du module de recherche de SUP du système EmbedDSU.

VII.3 Modélisation

Dans ce travail, nous présentons une méthode, basée sur le vérifieur de modèles SPIN, pour la vérification des propriétés de correction du module de recherche de SUP dans EmbedDSU. Avant d'interroger SPIN, une préparation du modèle à vérifier est nécessaire. Nous allons dans un premier temps définir le mécanisme à travers ses fonctions principales et les différentes données utilisées. Nous présentons particulièrement un langage cible qui nous permet de modéliser les principales instructions vis-à-vis du fonctionnement de la recherche de SUP. A l'issue de cette étape nous soumettons à l'outil Modex le code du mécanisme écrit en C pour obtenir un modèle écrit dans le langage Promela pour vérifier les propriétés de corrections avec l'outil SPIN.

VII.3.1 Principe de fonctionnement

VII.3.2 Les différents mode de la machine virtuelle

La recherche du point approprié pour appliquer la mise à jour est effectuée selon un algorithme de détection de point sûrs. Cet algorithme permet de garantir l'absence de méthodes restreintes durant l'exécution de la mise à jour. Pour ce faire, trois modes sont d'abord définis pour la machine virtuelle Java Card (voir Figure VII.3) :

- **Le mode *standard* :** Durant ce mode, la machine virtuelle s'exécute normalement, sans le processus de mise à jour. A la detection d'une mise à jour, la machine virtuelle passe directement au mode recherche appelé aussi mode *pending*.
- **Le mode *pending* :** A la détection d'une mise à jour, la machine virtuelle passe au mode *pending* pour la recherche d'un SUP. Un SUP est détecté à l'atteinte d'un état quiescent. Un état quiescent est caractérisé par l'absence de méthodes restreintes dans l'environnement d'exécution.
- **Le mode *process* :** A la détection d'un SUP, la mise à jour est effectuée selon le fonctionnement expliqué en chapitre 5. Lors de la mise à jour, la machine virtuelle est en mode *process*. La fin de l'application de la mise à jour fait passer la machine virtuelle au mode standard.

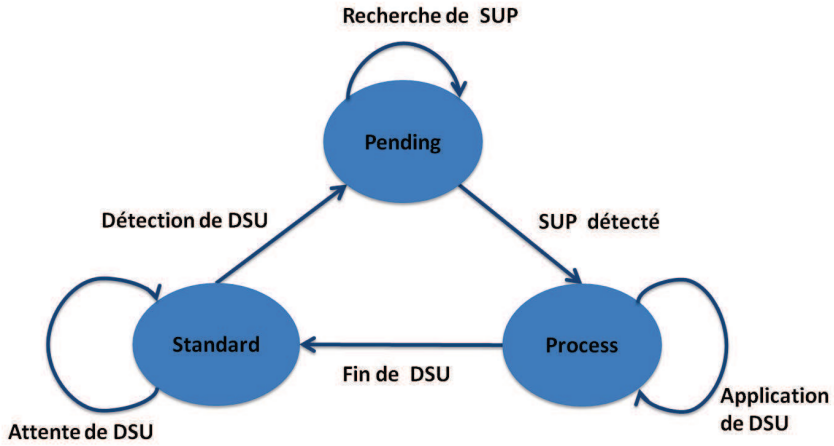

Figure VII.3 : Les différents modes de la machine virtuelle

VII.3.2.1 Description du langage cible

Nous considérons un langage cible pour étudier la correction de la détection des SUP. Dans ce langage cible, les instructions sont classifiées de manière à représenter les comportements pertinents pour l'étude de la recherche des SUP. Nous retenons les instructions suivantes :

- L'instruction **INST** : Cette instruction est utilisée pour modéliser les instructions bytecode habituelles. Le terme *habituelles* veut dire tous les types des instructions excepté les appels des méthodes et les retours des méthodes. On y retrouve donc par exemple, les opérations arithmétiques, les opérations sur la pile d'opérandes et les instructions de branchement.
- L'instruction **INVOKE** : Cette instruction est utilisée pour représenter les instructions d'appel des méthodes. L'exécution d'une instruction INVOKE permet de créer et d'initialiser un cadre d'activation (frame) dans la pile du thread. Cette catégorisation des instructions d'appel des méthodes permet une écriture adéquate des fonctions de recherche de SUP.
- Les instruction **SETLOCK** et **UNLOCK** : L'instruction SETLOCK permet de verrouiller le prochain thread dans la table des threads. L'instruction UNLOCK est utilisée pour lever le verrou sur un thread déjà verrouillé. Ces instructions sont utilisées pour représenter les instructions de synchronisation dans les langages multi thread.
- L'instruction **RETURN** : Cette instruction est utilisée pour représenter les retours des méthodes.

La détection des SUP est basée sur la notion d'état quiescent caractérisé par l'absence de méthodes restreintes dans l'environnement d'exécution. La classification suivante permet d'éclairer cette notion :

- **Méthode active/inactive :** Une méthode est active si elle est en cours d'exécution. Ceci signifie que la méthode possède un cadre d'activation dans la pile d'exécution et qu'elle a été invoquée. La méthode est dite inactive sinon.
- **Méthode restreinte/non restreinte :** Une méthode est restreinte si elle est active et concernée par la mise à jour. Une méthode est dite non restreinte si elle n'est pas concernée par la mise à jour, active ou inactive.

A l'invocation d'une méthode, un cadre d'activation est créé dans la pile du thread. L'initialisation du cadre d'activation contient, en plus des informations standard comme l'identifiant du cadre et l'identifiant de la méthode, une donnée pour indiquer si la frame est relative une méthode restreinte. Cette information est utilisée par les fonctions d'introspection de la recherche de SUP pour ramener le système à un état quiescent. Ces fonctions parcourent les structures de données

dans la machine virtuelle et vérifient l'existence et le nombre de méthodes restreintes pour chaque thread.

VII.3.2.2 États des threads

Un thread est caractérisé par : un identifiant, une liste de cadres d'activation, son statut et sa classe. Un thread passe par plusieurs états :

- En mode standard, un thread est schédulé puis exécuté, son état est alors dit actif (noté *active*). A l'exécution d'une instruction sleep, son état passe à bloqué (noté *Locked*).
- Un thread devient bloqué si un autre thread a posé un verrou sur le thread par une fonction *lockThread()*. Il est actif si le verrou est levé par le thread qui l'avait posé par une fonction *unlockThread()*. Un thread devient inactif (noté *Idle*) si le programme qui lui est associé termine.
- Pour prendre en compte la solution proposée par EmbedDSU pour la recherche des SUP, un autre état est introduit : il s'agit de thread bloqué dans le but d'atteindre un état quiescent et effectuer la mise à jour. Un thread devient bloqué pour mise à jour, noté *DSU_locked*, lorsque la machine virtuelle entre dans le mode *pending* et que le thread ne contient pas de cadres d'activation relatifs à des méthodes restreintes dans sa liste de cadres. L'idée principale est de ne pas avoir de nouvelles méthodes restreintes dans la pile des threads et d'exécuter les méthodes restreintes pour atteindre l'état quiescent.

VII.3.2.3 Les fonctions principales du module

Dans le but de détecter un SUP, la fonction d'introspection , appelée *searchSUP* permet de chercher les cadres d'activation de chaque thread de l'application. Quand une mise à jour est possible, La machine virtuelle passe alors en mode *process*. S'il existe des méthode restreintes dans la pile, le nombre de celles-ci est d'abord obtenu par introspection. Cette valeur est sauvegardée dans une variable appelée *nb_meth_stack*. La recherche lors du mode *pending* de la machine virtuelle utilise les fonctions suivantes :

- *createFrame :* Cette fonction est utilisée à chaque invocation de méthode. Son objectif est de créer un cadre d'activation associé à la méthode activée. Le cadre est placé à la tête de la liste des cadres d'activation du thread. Cette fonction est adaptée au mécanisme EmbedDSU comme ceci : à chaque fois qu'un cadre d'activation est créé, une vérification est effectuée : si le thread ne possède plus de cadre d'activation correspondant à des méthodes restreintes dans sa pile d'exécution alors son état passe à *DSU_locked* dans le but d'éviter que des méthodes restreintes ne soient évoquées par le thread. Dans le cas contraire, son exécution continue dans le but de permettre aux méthodes restreintes de finir leurs exécutions et atteindre l'état quiescent.
- *releaseFrame :* Cette fonction est utilisée à la fin de l'exécution de chaque méthode pour supprimer le cadre d'activation qui lui était créé. Lors de la recherche du SUP, cette méthode est adaptée dans le but de vérifier s'il existe des méthodes restreintes dans la pile du thread. Si le thread ne possède plus de cadre d'activation correspondant à des méthodes restreintes dans sa pile d'exécution alors sont état passe à *DSU_locked*, sinon, le thread continue son exécution. Dans le cas où la méthode qui termine son exécution est une méthode restreinte, le compteur *nb_meth_stack* est décrémenté. L'état quiescent est atteint quand ce nombre devient nul.
- *switchThread :* Quand la machine virtuelle est au mode *Standard*, cette fonction s'occupe de sélectionner le prochain thread à exécuter. Dans le mode *Pending*, cette fonction permet de passer au mode *Process* quand les threads sont bloqués.

Initialement, tous les cadres d'activations relatifs à des méthodes restreintes sont comptés. Si la valeur est supérieure à zéro, le mode de la machine virtuelle passe à *Pending*. La machine virtuelle continue à exécuter d'autres applications. La valeur est décrémentée à chaque fois qu'une méthode restreinte finit son exécution. Quand le compteur atteint zéro, un SUP est détecté et la machine virtuelle passe au mode *Process*.

Algorithme 5 SUP_mechanism

```
Entrées: UpdateSM, Instruction, nb_meth_stack ;
1  UpdateSM ← NoUpdate ;
2  Répéter
3      suivant Instruction faire
4          cas où INVOKE faire
5              CreateFrame() ;
6          fin
7          cas où INSTR faire
8              incr_pc() ;
9          fin
           /* Exécution d'une instruction standard, le compteur du programme est incrémenté  */
10         cas où SETLOCK faire
11             lockThread() ;
12         fin
           /* Recherche du thread suivant dans la table des threads                          */
13         cas où UNLOCK faire
14             unlockThread() ;
15         fin
           /* Effectuer une instruction unlock pour déverrouiller un thread par le thread
              responsable de l'instruction lock précédente                                   */
16         cas où RETURN faire
17             ReleaseFrame() ;
18         fin
           /* Suppression d'un frame associé à une méthode qui termine, adapté pour la DSU  */
19     fin
       /* Vérification de requêtes de DSU                                                    */
20     Si Notify_DSU() alors
21         SearchSUP() ;
22     finsi
       /* Commencer une recherche SUP en comptant les méthodes restreintes                   */
23     Si UpdateSM = Process alors
24         Break ;
25     finsi
       /* Quitter si le SUP est détecté dans la fonction SearchSup                           */
26 Jusqu'à ¬switchThread();
27 Si UpdateSM = Pending ∧ nb_meth_stack = 0 alors
28     UpdateSM ← Process ; /* Mettre UpdateSM à process dans le cas de la terminaison du
          thread                                                                             */
29 finsi
```

L'utilisation de ces fonctions principales dans le fonctionnement du mécanisme de recherche du SUP est représenté par l'algorithme *SUP_mechanism* (Algorithme 5). L'algorithme utilise les données suivantes : le mode de la machine virtuelle *UpdateSM*, l'instruction à exécuter *Instruction* et le nombre de méthodes restreintes *nb_meth_stack*. Initialement, la machine virtuelle est en mode *standard*, représenté par la valeur *NoUpdate*. Les méthodes sont exécutées à travers les instructions

du langage cible (ligne de 3 à 19). Le système vérifie la présence d'une requête de mise à jour avec le test *if Notify_DSU()* (ligne 20). Dans ce cas, la fonction *SearchSUP* est appelée. Cette fonction vérifie la présence de méthodes restreintes dans les piles des threads. La variable *UpdateSM* initialisée au mode standard *NoUpdate* est changée à *Process* dans le cas où il n'y' a pas de méthodes restreintes. La variables est changée à *Pending* dans le cas contraire. Lors du mode *Pending*, les fonctions *CreateFrames* et*ReleaseFrame* (lignes 5 et 17) sont exécutées de telle sorte à atteindre un état quiescent. Ceci est effectué principalement en mettant les threads à l'état *DSU_locked* quand les conditions sont réunies. Ce traitement est répété jusqu'à ce qu'il n'y ait plus de thread à scheduler ($\neg switchThread()$). Dans ce cas, soit la machine virtuelle atteint un état quiescent et passe au mode *Process*. Dans le cas où les threads terminent sans entrer dans l'état *Process*, une vérification est effectuée pour passer à cet état.

VII.3.3 Vérification des propriétés de correction

VII.3.3.1 L'absence d'inter-blocage

Nous avons d'abord utilisé SPIN pour vérifier l'absence d'inter-blocage. Nous décrivons d'abord un scénario représentant un exemple d'inter-blocage et proposons une solution pour le résoudre.

a) Un scenario d'inter-blocage

Le vérifieur de modèles SPIN offre un mécanisme automatique pour la détection des inter-blocages. Donc pour cette propriété l'utilisateur n'a pas besoin de l'exprimer en LTL. L'application de SPIN sur l'implémentation de l'algorithme *SUP_mechanism* a fait ressortir la présence d'inter-blocage. Cette situation est expliquée par l'exemple suivant : Considérons deux threads $T1$ et $T2$ avec respectivement les listes des cadres d'activation $FT1$ et $FT2$. Les éléments de la liste des cadres d'activation sont de la forme (F_{ij}, R) si le cadre d'activation F_{ij} correspond à une méthode restreinte, et de la forme (F_{ij}, NR) si le cadre F_{ij} correspond à une méthode non restreinte.

Comme illustré sur la Figure VII.4, les threads commencent leurs exécutions avec le statut *Active* correspondant à l'exécution normale. La machine virtuelle est initialement au statut *NoUpdate* correspondant à l'exécution standard. Les lignes verticales correspondent à des événements. Le premier événement $e1$ correspond à l'exécution d'une instruction *lock* par le thread $T1$, par conséquence, le thread $T2$ passe à l'état *locked*. Le deuxième événements $e2$ correspond à la détection d'une requête de mise à jour. La machine virtuelle passe au mode *pending* pour la recherche du SUP. Le thread $T1$ continue son exécution : l'événement $e3$ corresponds à la suppression du cadre d'activation F_{12} correspondant à une méthode restreinte. Ce thread passe à l'état *DSU_locked*, en vue d'atteindre l'état quiescent, parce que le cadre F_{12} était le dernier dans la pile d'exécution à correspondre à une méthode restreinte.

Comme le montre l'illustration, le statut de la machine virtuelle demeure indéfiniment à *Pending*. Ceci est dû au fait que dans la liste des cadres d'activation du thread $T2$, qui est à l'état *locked*, il existe encore des méthodes restreintes. Le mécanisme est dans l'impossibilité d'atteindre l'état quiescent et passer à l'état *process* de la machine virtuelle à cause de la situation d'inter-blocage telle que : le thread $T1$ est à l'état *DSU_locked* dans l'attente de la fin du mécanisme de la mise à jour et le thread $T2$ est en attente de passer de l'état *locked* à l'état *active* pour continuer son exécution et exécuter les méthodes restreintes, la machine virtuelle est dans l'attente de la condition de l'état quiescent pour passer au mode *Process*.

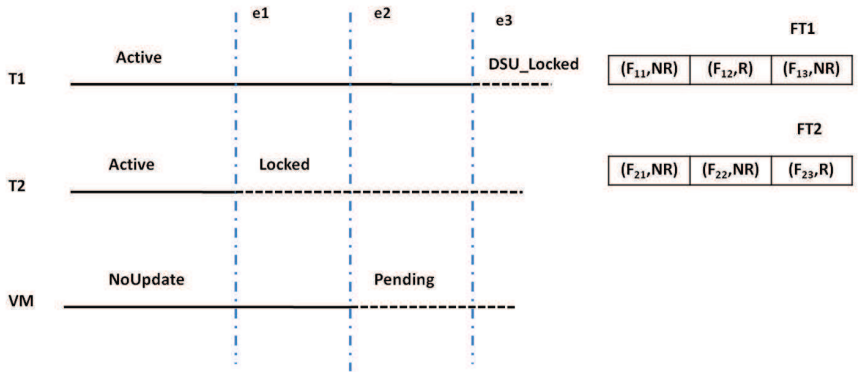

Figure VII.4 : Illustration d'un scénario d'inter-blocage

VII.3.3.2 Résoudre les situations d'inter-blocage

Dans le but d'éviter les situations d'inter-blocage, nous devons nous assurer qu'un thread passe à l'état *DSU_locked* seulement après vérification qu'il ne détient pas un verrou sur un autre thread qui possède encore des cadres d'activation relatifs à des méthodes restreintes. Ceci signifie qu'une introspection supplémentaire est effectuée pour détecter la présence de thread bloqués possédants des méthodes restreintes. Le mécanisme est amélioré donc par un fragment de code qui permet de garantir qu'avant chaque instruction pour faire passer un thread à l'état *DSU_locked*, un parcours est réalisé sur la liste des thread pour vérifier que le thread concerné n'a pas effectué le blocage d'un autre thread. Cette solution permet de reporter les opérations changeant les états des threads vers *DSU_Locked* jusqu'à ce que tous les threads possédant des méthodes restreintes soient débloqués.

Comme l'illustre la Figure VII.5, cette solution implique le report de l'application du statut *DSU_locked* sur les threads. Dans le scénario considéré, le thread $T1$ continue son exécution jusqu'à ce que le mécanisme reçoive une notification (évènement $e3'$) qu'il n'existe pas de threads bloqués possédant des cadres d'activation relatifs à des méthodes restreintes. Sur la figure, l'évènement $e3$ correspond à la fin d'une méthode de $T1$ et $e4$ correspond à l'exécution par $T1$ d'une instruction UNLOCK qui permet au thread $T2$ de retrouver le statut *Active*. L'évènement $e5$ représente la fin de la méthode dont le cadre d'activation est F_{23} qui correspond à une méthode restreinte. Ceci permet de mettre le thread $T2$ au statut *DSU_locked* et de ramener la machine virtuelle au statut *process*.

VII.3.3.3 La sûreté d'activation

La sûreté d'activation permet de s'assurer qu'une mise à jour est effectuée si et seulement si les méthodes concernées par la mise à jour ne sont pas en exécution (actives). Le nombre des méthodes restreintes *nb_meth_stack* est utilisé lors du processus de la recherche SUP : un SUP est atteint lorsque le nombre de méthodes restreintes est nul. Ce critère permet d'éviter l'inconsistence : une mise à jour d'une méthode active conduit le système à utiliser différentes versions d'une même méthode. La propriété de sûreté d'activation est exprimée en LTL comme suit :

$$\Box((UpdateSM == Process) \rightarrow (nb\_meth\_stack == 0))$$

Ceci exprime que, toujours, quand la machine virtuelle entre dans le mode *Process*, alors le nombre de méthodes restreintes est nul. Cette propriété est vérifiée avec succès par l'outil SPIN.

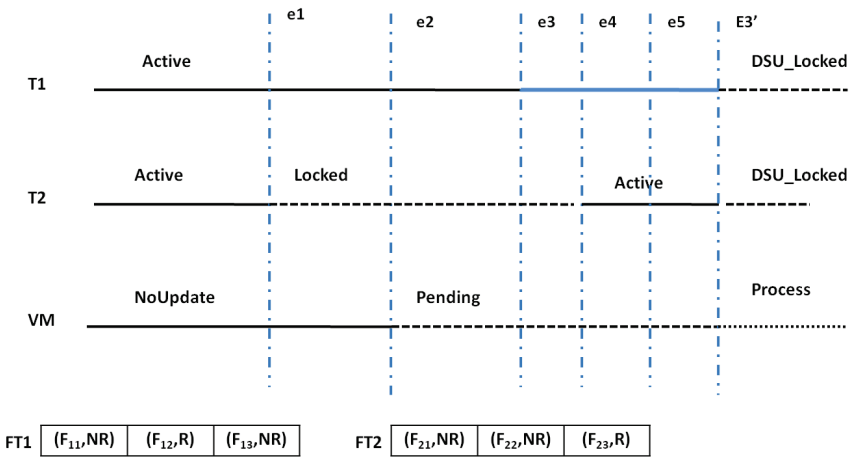

Figure VII.5 : Illustration de la solution évitant les inter-blocages

VII.3.3.4 La garantie de mise à jour

La propriété de garantie de mise à jour permet de s'assurer qu'une fois une requête de mise à jour effectuée, le système de DSU permet de ramener l'application vers un SUP. C'est une propriété de *liveness*. Dans la vérification de modèles, une propriété de *liveness* permet d'énoncer que quelques chose de bien va éventuellement se produire. Dans le cas de la recherche de SUP, cette propriété que nous appelons garantie de mise à jour assure que le système va éventuellement atteindre un état quiescent. Ceci est formalisé comme suit :

$$\Box((UpdateSM == Pending) \rightarrow \diamond(UpdateSM == Process))$$

Dans cette formule, la variable $UpdateSM$ est utilisée pour représenter le mode de la machine virtuelle. La propriété exprime que, toujours, quand la machine virtuelle est dans le mode *Pending*, ce qui veut dire qu'une requête de mise à jour a été détectée et que la machine virtuelle est en recherche de SUP, le processus va éventuellement atteindre un état quiescent qui permettra d'effectuer la mise à jour , ce qui veut dire, la machine virtuelle ca atteindre le mode *Process*. Cette propriété est vérifiée avec succès par la version amélioré du programme.

VII.4 Conclusion

Nous avons présenté dans ce chapitre notre contribution concernant la correction de la recherche de points sûrs dans le système EmbedDSU. Nous avons proposé d'utiliser le model checking pour vérifier trois propriétés de correction : l'absence d'inter-blocage, la sûreté d'activation et la garantie de mise à jour. Nous avons présenté dans un premier temps le fonctionnement détaillé du mécanisme à travers ses principales fonctions et données utilisées. Nous avons ensuite obtenu à partir d'un code en C un modèle écrit en PROMELA. Ce modèle ainsi que les propriétés souhaitées sont alors soumis au vérifieur de modèles SPIN. L'utilisation du model checking nous a permis d'abord de détecter un problème d'inter-blocage dans le système. Après avoir corrigé la situation d'inter-blocage, nous avons spécifié puis vérifié nos propriétés temporelles. Nous avons amélioré le mécanisme de recherche SUP de EmbedDSU et démontré qu'il satisfait les trois propriétés de correction énoncées en objectif.

Chapitre VIII

La mise à jour des données

Sommaire

La mise à jour dynamique des programmes consiste à modifier ceux-ci sans arrêter leurs exécutions. Lors de la mise à jour dynamique des applications Java Card, trois parties principales sont considérées : le bytecode, la pile des blocs d'activation des méthodes et le tas des instances utilisées par les applications. La mise à jour des instances dans le tas passe par l'application de fonctions de transfert d'états (State Transfer Functions : STFs). Les différentes relations existantes entre les objets ainsi qu'entre les objets et les méthodes en exécution rendent la mise à jour dynamique du tas délicate : une mise à jour hasardeuse peut introduire des erreurs dans l'application par des inconsistances entre les différentes données et leurs utilisations par les différentes versions du code. Dans ce chapitre, nous proposons une approche sûre pour la mise à jour dynamique du tas. Notre approche est basée sur une modélisation formelle du tas et sur un algorithme d'ordonnancement des classes à mettre à jour et des STFs. Elle permet de prendre en compte les différentes relations entre les objets dans le tas ainsi que les codes écrits par l'utilisateur dans les STFs. Le travail présenté dans ce chapitre est à l'état préliminaire. Il est publié comme papier court dans [142].

VIII.1 Motivations et objectifs

La mise à jour des données représente une partie majeure dans la conception des systèmes de DSU. Les données d'une application en exécution sont structurées de manière à permettre aux programmes de les manipuler. Les mises-à-jour dynamiques peuvent apporter des modifications au niveau du code et aussi au niveau des structures ou des valeurs des données. Des erreurs arrivent inévitablement si les différentes versions des données et des codes ne sont pas compatibles. Ces problèmes peuvent survenir par exemple, si le nouveau code utilise les anciennes versions des données. Les systèmes de DSU fournissent des mécanismes pour permettre la transformation ou la migration des données vers les nouveaux formats et/ou valeurs.

A l'instar de plusieurs systèmes de DSU, le système EmbedDSU effectue ces transformations en appliquant des STFs. Ces STFs sont contenues dans le fichier DIFF envoyé sur la carte où elles sont utilisées pour transformer les instances dans le tas de la machine virtuelle pour obtenir les nouvelles versions d'instances correspondant à la nouvelle version de la classe. Les STFs dans EmbedDSU sont fournies par le programmeur.

Ces transformations sont souvent complexes et soulèvent plusieurs questions :

- **Les dépendances entre les objets , les méthodes et les STFs :** L'ajustement apporté à l'état de l'application durant l'application de la mise à jour dynamique, est spécifié dans les STFs. Deux problèmes peuvent survenir avec ces dépendances. Le premier provient des dépendances entre les classes. En effet, si une classe fait référence à une autre classe et si les deux sont concernées par la mise à jour alors le résultat peut dépendre de l'ordre de l'exécution de la mise à jour et de l'ordre de l'exécution des STFs relatives aux deux classes. Le deuxième problème est posé par les STFs qui peuvent utiliser dans leurs traitements des instances de classes concernées par la mise à jour.

- **Les méthodes en exécution :** Une méthode en cours d'exécution peut utiliser un objet plusieurs fois. Si par exemple, une méthode non concernée par la mise à jour utilise une instance à mettre à jour deux fois, dans le cas où la mise à jour se produit entre la première et la deuxième utilisation, la méthode utilisera deux versions différentes d'une même instance, ce qui produira des erreurs dans le programme.

- **Les contextes Java Card :** Un des avantages de la technologie Java Card consiste en la sécurité qu'elle assure en terme de partage des données entre les applets embarquées. En particulier, la machine virtuelle assure le principe d'isolation des applets en terme d'accès aux données et invocation de méthodes virtuelles grâce à un mécanisme appelé le pare-feu(firewall). Ce mécanisme permet un contrôle d'accès aux données en répartissant les applications en espaces protégés appelés contextes et associés à leurs packages : le contexte d'une applet est son package. Il existe de surcroît un package privilégié appelée le *contexte JCRE* associés aux opérations du systèmes. La politique de sécurité stipule que l'accès à des objets à l'intérieur d'un contexte est autorisé alors que l'accès à des objets n'appartenant pas au même contexte est interdit excepté dans des cas bien précis où l'application partage ses données via une interface d'objets partageables (*Shareable Objects Interface*). Ce mécanisme permet d'assurer la sécurité de l'objet en effectuant des vérifications pour les accès aux objets du tas. Un objet appartient à un unique propriétaire qui peut être ou bien une instance d'applet ou bien le système. Le propriétaire d'un objet est celui qui était actif quand l'objet a été crée et un objet ne peut être accédé que si le contexte de son propriétaire est le contexte actuel courant. La question des contextes est posée pour la DSU par le fait que l'accès aux instances est au coeur même du fonctionnement de la mise à jour des données. Le mécanisme de DSU doit prendre en considération les différents contextes des objets.

Nous proposons une approche permettant une mise à jour sûre du tas de la machine virtuelle Java Card en nous basant sur les critères suivants :

- **Consistance dans les versions des instances :** Ce critère permet de s'assurer qu' une méthode en cours d'exécution lors du processus de la mise à jour n'utilise pas plusieurs versions différentes d'une instance.
- **Cohérence des dépendances entre les instances :** Ce critère permet de s'assurer que l'application des STFs n'introduit pas des erreurs dans les relations entre les instances.
- **Préservation des contextes :** Ce critère permet de garantir que la mise à jour du tas ne compromet pas l'isolation des applications en s'assurant que pour les nouvelles versions des instances, l'appartenance aux différents contextes ayant crées les instances appartenant à la classe mise à jour soit préservée.

VIII.2 Approche pour la mise à jour dynamique du tas

VIII.2.1 Le modèle formel du tas

Les objets Java Card sont alloués dans le tas. Nous proposons de modéliser le tas comme un graphe orienté $G = (V; E)$ ou :

- V représente l'ensemble de noeuds (instances). Un objet est défini par le nom de la classe dont il est instance, une référence et par l'identifiant de son contexte (context ID).
- E représente l'ensemble des arcs. Il représente les relations existantes entre les objets. Chaque arc e est défini par trois information et nous notons : $e = (s; t; l)$ où :
  - s est le noeud source de e ;
  - t est le noeud cible de e ;
  - $l = \{héritage, dépendence\}$ est un ensemble d'étiquettes sur les arcs pour spécifier le type de relation existant entre les objets. On dit qu'une classe A dépend d'une classe B si la classe A utilise un champ ou une méthode de la classe B.

Nous ajoutons à notre modèle un noeud spécial appelé noeud des blocs d'activation (frames). Il représente les méthodes en cours d'exécution dans l'application. Un arc étiqueté "utilisé-par" existe entre chaque méthode et chacune des instances qu'elle utilise. La Figure VIII.1 représente le modèle formel du tas de la machine virtuelle. La figure représente un noeud spécial représentant une pile des cadres d'activation des méthodes ($m1$, $m2$ ..., mn). Les instances de classes créées dans le tas ($O1$, $O2$ ..., On) sont reliées aux méthodes qui les utilisent. Les instances sont reliées entre elles par des liens pouvant être marqués pour indiquer les relations d'héritage ou de dépendance.

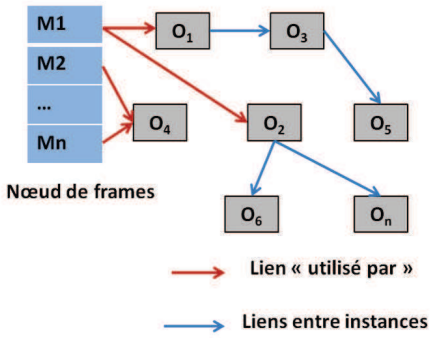

Figure VIII.1 : Modèle du tas

VIII.2.2 Les étapes de l'approche

La modification des instances dans le tas passe par la définition des fonctions de transfert pour les classes dont les objets seront mis à jour. Une STF concernant une classe est appliquée sur chaque instance de la classe durant la mise à jour. La STF retourne une instance initialisée, le nouvel objet prend ainsi l'identité de celui sur lequel la STF est appliquée. Les STFs sont écrites par le programmeur. Selon le corps des STFs, celles-ci sont divisées en trois catégories :

- Les STFs à valeurs par défaut : Il s'agit de mettre à jour ou d'initialiser les instances avec des valeurs par défaut.
- Les STFs à calcul simple : Il s'agit de mettre à jour ou d'initialiser les instances après un calcul qui n'utilise pas un autre objet à mettre à jour.

– Les STFs avec accès aux autres instances : Ce type de STFs contient des références vers des objets concernés par la mise à jour.

Dans notre modèle, une STF ft est un triplet (I, C, Oc) où :
– I est le nom de la classe des instances à mettre à jour ;
– C est le corps de ft ;
– Oc est l'ensemble des objets utilisés par ft.

La Figure VIII.2 représente les trois étapes de notre approche : (1) l'introspection du tas, (2) l'ordonnancement des STFs et (3) l'application et la propagation des mises à jour. Étant donné :
– G, un graphe représentant un modèle du tas ;
– P, un correctif (ou un DIFF file) contenant une description des mises à jour à effectuer (classes modifiées, champs ajoutés...) ;
– F, un ensemble de STFs $\{ STF_1, STF_2 \ldots, STF_n \}$;

le but de notre approche est d'obtenir un nouveau tas correspondant à la nouvelle application en appliquant les STFs. Une mise à jour dynamique du tas est considérée comme une transformation d'une modélisation du tas en entrée (graphe G) vers un modèle G', de telle façon que les critères cités comme objectifs soient vérifiés.

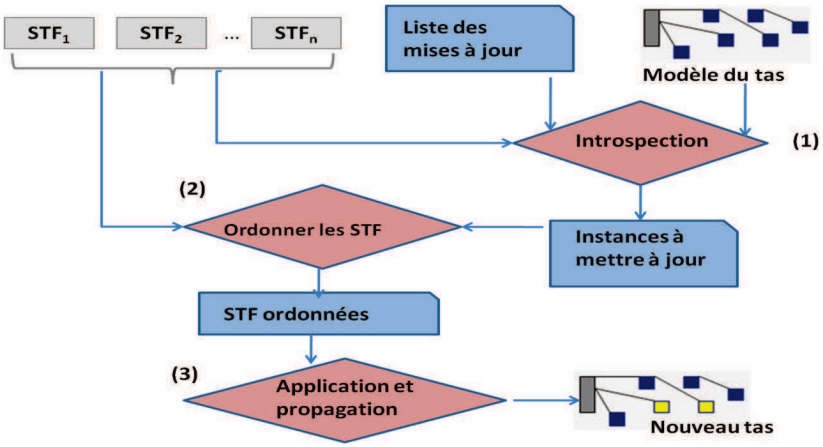

Figure VIII.2 : Les étapes de l'approche

VIII.2.2.1 Étape 1 : Introspection du tas

Cette étape prend des informations à partir du fichier DIFF sur les classes à mettre à jour puis parcourt le graphe représentant le tas pour trouver les instances concernées par la mise à jour. Cette première tâche est implémentée dans EmbedDSU. La parcours est effectué par 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 des racines 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 machine virtuelle. Notre approche nécessite de compléter cette introspection initiale par une recherche supplémentaire permettant d'avoir des informations sur :
– Les instances de classes éventuellement utilisées par les fonctions de transfert d'état ;
– L'existence de liens entre les fonctions de transfert d'états. Nous nous intéressons au cas ou une STF dédiée à la mise à jour d'instances qui sont utilisées par d'autres STFs ;

– Les différents liens qui peuvent exister entre les instances à mettre à jour (héritage, utilisation) ;
– Les contextes des différentes instances concernées par la mise à jour.

Les deux premières informations seront utilisées par l'étape suivante et servent à garantir la cohérence des relations entre les objets dans le tas. Les informations sur les contextes seront utilisées lors de l'application de la mise à jour pour vérifier que pour chaque instance, l'ancienne et la nouvelle version appartiennent au même contexte, pour établir la préservation de l'isolation des applets.

VIII.2.2.2 Étape 2 : Ordonnancement des classes et des STFs

L'ordre de la mise à jour des classes et de l'exécution des STFs sur les instances est nécessaire pour préserver la cohérence du tas. Nous proposons un algorithme (Algorithme 6) pour calculer ces ordres de façon automatique en se basant sur les informations contenues dans le modèle du tas par l'étape d'introspection et les informations véhiculées par les STFs elles mêmes. Le graphe en entrée est préalablement construit par une étude des liens de dépendance entre les classes.

Dans un premier temps (jusqu'à la ligne 12 sur l'algorithme), on visite les sommets du graphe par un parcours en largeur. Pour cela, nous utilisons la file $file\_parcours$ qui contiendra les noeuds lors du parcours. Un noeud est retiré de la file puis inséré dans la liste C_ord représentant l'ordre de la mise à jour des classes. Ensuite il est examiné pour vérifier s'il correspond au nom d'une classe correspondant à une des STFs (*FTC (f1)*) contenue dans la liste des STF en entrée. Dans ce cas, la STF est ajoutée à la liste ordonnée des STF F_ord (initialement vide []). Le symbole symbole : : représente la construction de liste.

L'algorithme enfile ensuite les voisins du noeud en cours pour les examiner. Cette partie finit quand la file est vide. On obtient une liste de l'ordre de la mise à jour des classes et une liste de l'ordre des STFs lui correspondant. Dans la deuxième partie de l'algorithme, on examine les liens existants entre les classes à mettre à jour et celles utilisées dans les corps des STF (STF_use). Á partir de ces liens, on construit une liste de contraintes sur F_ord. Cette liste ($list\_contr$) est prise en compte ensuite par un mécanisme de programmation avec résolution des contraintes pour valider l'ordre vis-à-vis des relations entre les classes et traiter les références récursives si elles existent dans ($list\_contr$).

VIII.2.2.3 Étape 3 : Synchronisation et propagation

Étant donné les listes ordonnées des classes à mettre à jour et des STF ordonnées, du tas et de la liste des instances à mettre à jour obtenue par introspection, cette étape consiste à effectuer la mise à jour selon l'ordre calculé en tenant en compte des contextes ainsi que des méthodes en cours d'exécution utilisant les instances. L'application d'une STF sur une instance x est déterminée en fonction de l'ensemble des méthodes utilisant x. Nous proposons à cet effet un mécanisme de synchronisation :

– La synchronisation est nécessaire à maintenir la consistance relative à l'utilisation d'une instance par une méthode. Cette notion permet de garantir qu'une méthode en cours d'exécution, n'utilise pas deux versions différentes d'une même instance. Ceci est basé sur les liens entre les les instances et le noeud des blocs d'activation du modèle. A cet effet, la mise à jour d'une instance est déférée s'il existe une méthode qui a utilisé, avant le lancement de la mise à jour, la version initiale de l'instance.
– La propagation concerne la mise à jour des classes dépendantes de la classe mise à jour avec les nouvelles informations (nouvelles références). Pour ces classes dépendantes, l'objectif de cette partie 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.

Algorithme 6 Ord_STF

Entrées: STF_list,$G =< V, E >$, S
1 **Sorties** *F_ord, F_ord, list_cont*
Variables : file_parcours, F_ord, C_ord, list_contr ;
$F\_ord \leftarrow []$; $C\_ord \leftarrow []$;
enfiler(S,file_parcours) ;
2 **Tant que** $(\neg(vide(file\_parcours))$ **faire**
3 x ← defiler(file_parcours) ;
$C\_ord \leftarrow x :: C\_ord$;
4 **Pour chaque** $STF\ f1 \in STF\_list$ **faire**
5 **Si** *FTC(f1)* = x **alors**
6 $F\_ord \leftarrow f1 :: F\_ord$;
7 **finsi**
8 **Fchq**
9 **Pour chaque** $y \in voisin(x)$ **faire**
10 enfiler(y,file_parcours) ;
11 **Fchq**
12 **Ftq**
13 $list\_cont \leftarrow []$;
14 **Pour chaque** $STF\ f1 \in F\_ord$ **faire**
15 **Pour chaque** $STF\ f2 \in F\_ord \setminus \{f1\}$ **faire**
16 **Si** $f1 \in STF\_use\ f2$ **alors**
17 $list\_cont \leftarrow (f1; f2) :: list\_cont$;
18 **finsi**
19 **Fchq**
20 **Fchq**

VIII.3 Conclusion

Nous avons présenté dans ce chapitre notre contribution concernant la mise à jour des données. Après avoir étudié le système EmbedDSU, nous avons constaté que la mise à jour des données dans ce système soulève quelques points : premièrement, nous avons remarqué que cette partie passe par l'utilisation des fonctions de transfert d'états avec valeurs par défaut. Ceci réduit l'expressivité des mises à jour pour le système. Nous avons également constaté que l'application de la mise à jour passe par un algorithme de type *garbage collector* qui ne prend pas en considération les différents types de liens entre les objets ou bien entre les objets et les méthodes en exécutions. Nous avons constaté également que la vérification des contextes n'est pas prise en compte. Nous avons proposé une approche pour la mise à jour des instances dans le tas de la machine virtuelle permettant de garantir trois critères : la consistance dans l'utilisation des instances ; la cohérence des dépendances entre les instances et la préservation des contextes. Notre approche se base sur un algorithme permettant d'ordonnancer l'application de la mise à jour et d'obtenir un tas consistant. L'approche prend également les fonctions de transfert écrites par l'utilisateur. Ce travail est en cours. L'objectif à très court terme est de finir l'implémentation de la solution et de la comparer avec la solution existante en terme d'efficacité. Une analyse formelle pour établir les trois critères étudiés est également en perspective.

Quatrième partie

Conclusion

Chapitre IX

Conclusion

"In literature and in life we ultimately pursue, not conclusions, but beginnings." Sam Tanenhaus.

En réponse au besoin croissant d'évolution des systèmes en parallèle du besoin de haute disponibilité, la mise à jour dynamique propose des techniques permettant de faire évoluer les applications et/ou systèmes sans les arrêter en préservant leurs disponibilités et l'état d'exécution de ceux-ci. La mise à jour dynamique est utilisée dans des applications critiques. Les différents travaux autour de la mise à jour dynamique ont montré la nécessité d'établir la correction formelle de celle-ci. En effet, de par son utilisation dans des applications critiques, des techniques rigoureuses sont requises pour montrer que l'application de la DSU n'introduit pas d'erreurs dans les systèmes mis à jour. Ce besoin en rigueur est apporté par l'utilisation des méthodes formelles.

IX.1 Contributions

Dans notre travail, nous avons étudié la correction formelle de la mise à jour dynamique dans un système de DSU pour les applications Java Card. Les contributions de cette thèse se situent au niveau de la correction de la mise à jour du code, de la recherche des points sûrs et de la mise à jour des données.

Correction de la mise à jour du code

Nous avons établi la correction du code mis à jour par deux propriétés : la sûreté de typage et la correction comportementale. Pour la première propriété, nous avons d'abord proposé une sémantique formelle des opérations de mise à jour. La sémantique définie permet de spécifier, pour chaque opération de mise à jour, l'ensemble des conditions permettant de garantir une application sûre de l'opération. Nous avons ensuite, sur la base de cette sémantique, énoncé et démontré des lemmes aboutissant à établir la propriété de la sûreté de typage des programmes mis à jour. Nous avons également proposé une méthode pour établir les propriétés comportementales des codes mis à jour. Cette méthode est principalement basée sur la proposition d'un calcul de transformation de prédicats par les opérations de mise à jour. Pour un programme à mettre à jour annoté par des opérations de mise à jour provenant d'un fichier DIFF, la transformation des prédicats proposée permet d'aboutir à une spécification. L'équivalence entre la spécification calculée et la spécification désirée par le programmeur permet d'établir la correction comportementale de la mise à jour du code. Nous pensons que les approches proposées peuvent être exploitées par d'autres systèmes de mise à jour particulièrement les systèmes opérants au niveau bytecode. Une autre utilisation possible de nos propositions consiste en l'étude des comportement des applications suite à l'injection

de code dans le but de l'instrumenter ou bien d'en étudier des propriétés de sécurité.

Correction de la recherche des points sûrs

Concernant l'aspect de la recherche des points sûrs, nous avons étudié la correction en proposant d'établir trois critères : l'absence d'inter blocage, la sûreté d'activation ainsi que la garantie de mise à jour. Nous avons montré l'utilisation du model checking pour premièrement détecter une situation d'inter blocage dans la version initiale du module de recherche SUP. Nous avons proposé une solution permettant d'éviter les inter blocages. Nous avons ensuite utilisé le model checking pour établir les propriétés de correction. Notre vérification est basée sur la spécification des propriétés en logique temporelle et une modélisation du module : les fonctions principales du module et les différentes données utilisées. La vérification que nous avons effectuée représente le premier cas d'étude pour l'utilisation de techniques et outils existants de model checking pour modéliser et vérifier des propriété de correction relatives à la correction d'un module de SUP pour Java Card.

La mise à jour des données

Le troisième aspect que nous avons traité concerne la mise à jour des données. La mise à jour des instances dans le tas passe par l'application de Fonctions de Transfert d'Etat (FTE). Le système EmbedDSU se base initialement sur les fonctions de transferts à valeurs par défaut. L'ordre de l'application des FTE ainsi que les différentes relations existantes parmi les instances ne sont pas prises en compte dans cette solution. Dans ce travail, nous avons proposé une approche sûre pour la mise à jour dynamique du tas basée sur une modélisation formelle de celui-ci et sur un algorithme d'ordonnancement des classes à mettre à jour et des FTEs. La solution est en cours d'implémentation.

IX.2 Perspectives

Les principales perspectives de recherche qui apparaissent à l'issue de cette thèse sont catégorisées en trois axes :

- Extension de nos trois contributions au niveaux code, données et recherche de points sûrs ;
- Conception d'un formalisme général sur la base de nos contributions ;
- Etude de propriétés avancées.

La mise à jour du code

Nous avons effectué une formalisation des opérations de mises à jour qui nous a permis d'énoncer et de démontrer la propriété de sûreté de typage pour les programmes mis à jour. La formalisation est effectuée dans l'assistant de preuve Coq. Nous envisageons dans un premier temps d'étendre la formalisation à des aspects non traités du langage bytecode comme par exemple les tableaux et les exceptions. Nous comptons également compléter le calcul de plus faible précondition vers les opérations de suppression d'instructions. La formalisation de la sémantique formelle des opérations de mise à jour et des démonstrations concernant la propriété de la sûreté de typage permettra d'aboutir à un vérifieur certifié sans erreur. Au niveau des spécification des propriétés comportementales des programmes mis à jour, la limite des ressources des cartes à puce soulève le problème pour appliquer de telles vérifications sur la carte. La réflexion dans ce sens constitue une autre piste pour des travaux futurs.

La recherche de points sûrs

Nous avons établi la correction formelle du module de la détection de SUP en utilisant le model checking. Une possible extension de l'utilisation du model checking consiste en la vérification des propriétés de correction des différentes sémantiques pour la mise à jour des données. Au niveau de la formalisation du fonctionnement, un travail est en cours pour une formalisation fonctionnelle qui aboutira à établir par preuve de théorèmes les propriétés de correction.

La mise à jour des données

Nous avons présenté une approche pour la mise à jour sûre du tas basée sur l'ordonnancement automatique des fonctions de transfert d'états. Actuellement, nous étendons le système EmbedDSU pour intégrer l'approche proposée. Le système EmbedDSU se base initialement sur les fonctions de transferts à valeurs par défaut, l'ordre de leurs exécutions n'est pas pris en compte. La mise à jour des instances est basée sur une approche globale, l'approche que nous proposons permet d'éliminer les problèmes de cohérence des versions des instances vis-à-vis des méthodes qui les utilisent. La suite de notre travail dans l'immédiat est d'implémenter l'approche proposée et d'effectuer une comparaison avec la version actuelle en terme de ressources. Nous envisageons également d'étudier la correction formelle de l'approche proposée.

Vers un formalisme fonctionnel général

Nous prévoyons d'étendre également notre formalisation pour inclure une formalisation fonctionnelle pour les aspects données et points sûrs en vue d'obtenir un formalisme fonctionnel permettant de raisonner sur les trois aspects de la mise à jour dynamique. Une partie de la formalisation concernant la recherche de points sûrs est déjà effectué dans Coq.

Propriétés avancées

Un autre point dans nos perspectives consiste à étudier des propriétés de correction plus avancées. Il s'agit de prendre en considération les points suivants : l'arrachage de la carte et l'effacement sûrs des données. En effet, la processus de mise à jour peut être interrompu par l'arrachage de la carte. L'arrachage lors de la mise à jour dynamique peut conduire la carte à un état inconsistent. Nous envisageons d'élargir les propriétés de correction en prenant en compte l'effet de l'arrachage sur l'état du système. Le deuxième point est relatif aux anciennes versions des codes et données. En effet, le processus de mise à jour dans EmbedDSU utilise le mécanisme de recopie en modification pour les codes et les données. Les applications Java Card sont utilisées dans des applications comportant des données sensibles et personnelles. Les processus de mise à jour dynamique doivent veiller à la sûreté des données mises à jour mais également à l'utilisation que les entités malveillantes peuvent tirer des anciennes versions des codes et données. Nous envisageons d'étudier les propriétés de corrections relatives aux versions initiales des programmes.

Annexe : Les règles sémantiques de suppression d'instructions

$$\frac{\begin{array}{l} Dlt\_inst\ \ goto\ L\ \ (i+1) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, goto\ L, i+1) \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1, L \in DOM(BC)\ PC\_MAX -- \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Dlt\_inst\ \ (pop\ \ (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = \\ Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, pop, i+1) \\ S_i = t.S_0 \rightarrow \\ (M2)S_{i+1} = \\ Effects\_STK(M2[i+1], t.S_0) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX -- \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Dlt\_inst\ \ (putfield(A,f,t)\ \ (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, putifield(A,f,t), i+1) \\ S_i = A.t.S_0 \rightarrow \\ (M2)S_{i+1} Effects\_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX - 3 \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Dlt\_inst\ \ (getfield(A,f,t)\ \ (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, getifield(A,f,t), i+1) \\ S_i = A.S_0 \rightarrow \\ (M2)S_{i+1} Effects\_STK(M2[i+1], A.S_0) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX - 3 \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Dlt\_inst\ \ (neg\ \ (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, neg, i+1) \\ S_i = int.S_0 \rightarrow \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX -- \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Dlt\_inst\ \ (load\ x\ \ (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, load\ x, i+1) \\ (M1)S_{i+1} = t.S_0 \rightarrow \\ (M2)S_{i+1} Effects\_STK(M2[i+1], S_0) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX -- \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Dlt\_inst\ \ (const\ a\ \ (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, const\ a, i+1) \\ S_i = S_0 \rightarrow \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX -- \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

$$\frac{\begin{array}{l} Dlt\_inst\ \ (invokevirtuel(A,l,t)\ \ (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, invokevirtuel(A,l,t), i+1) \\ S_i = tn_1.tn_2 \ldots tn_n.S_0 \rightarrow \\ (M2)S_{i+1} Effects\_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX - 3 \end{array}}{<F_i,S_i,SD_i,M1,i> \rightarrow <F_{i+1},S_{i+1},SD_{i+1},M2,i+1>}$$

Bibliographie

[1] Robert S. Fabry. How to design a system in which modules can be changed on the fly. In Raymond T. Yeh and C. V. Ramamoorthy, editors, *Proceedings of the 2nd International Conference on Software Engineering, San Francisco, California, USA, October 13-15, 1976.*, pages 470–476. IEEE Computer Society, 1976.

[2] M. Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information science, University of Pennsylvania, USA, 2001.

[3] D. Gupta. *On-line software version change*. PhD thesis, Indian Institute of Technology, India, 1994.

[4] Dominic Duggan. Type-based hot swapping of running modules. *Acta Informatica*, 41(4) :181–220, 2005.

[5] Agnś Cristelle Noubissi. *Mise á jour dynamique et scurisée de composants systéme dans une carte á puce*. PhD thesis, University of Limoges, France, 2011.

[6] Danny Weyns, M. Usman Iftikhar, Didac Gil de la Iglesia, and Tanvir Ahmad. A survey of formal methods in self-adaptive systems. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, C3S2E '12, pages 67–79, New York, NY, USA, 2012. ACM.

[7] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems*, WOSS '04, pages 28–33, New York, NY, USA, 2004. ACM.

[8] Jeff Kramer and Jeff Magee. Dynamic configuration for distributed systems. *IEEE Trans. Softw. Eng.*, 11(4) :424–436, April 1985.

[9] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. De Wit. A dynamic reconfiguration run-time system. In *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*, pages 66–75, April 1997.

[10] Luis Gabriel Ganchinho de Pina. *Practical Dynamic Software Updating*. PhD thesis, Université de Lisbonne, Institut superieur technique, 2016.

[11] I. lee. *Dymos : A dynamic modification system*. PhD thesis, University of Wisconsin , Madison, USA, 1983.

[12] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Institut Royal de Technologie, Suède, 2003.

[13] Luís Pina, Luís Veiga, and Michael Hicks. Rubah : DSU for java on a stock JVM. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, October 2014.

[14] Y. Murarka. *Online Update of Concurrent Object Oriented Programs.* PhD thesis, Indian Institute of Technology, India, India, 2010.

[15] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for c. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 72–83, New York, NY, USA, 2006. ACM.

[16] Andrew Baumann, Jeremy Kerr, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Module hot-swapping for dynamic update and reconfiguration in k42. In *In 6th Linux.Conf.Au*, 2005.

[17] Jeff Arnold and M. Frans Kaashoek. Ksplice : Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 187–198, New York, NY, USA, 2009. ACM.

[18] A. Orso, A. Rao, and M.J. Harrold. A technique for dynamic updating of java software. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 649–658, 2002.

[19] Jevgeni Kabanov and Varmo Vene. A thousand years of productivity : the jrebel story. *Software : Practice and Experience*, 44(1) :105–127, 2014.

[20] Hajar Ikhlef. Génération efficace de graphes d'appels dynamiques complets. mémoire pour le grade de maître ès sciences. département d'informatique et de recherche opérationnelle, faculté des arts et des sciences, université de montréal,2011.

[21] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic java classes. In Elisa Bertino, editor, *ECOOP 2000 : Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 337–361. Springer Berlin Heidelberg, 2000.

[22] Shi Zhang and Linpeng Huang. Research on dynamic update transaction for java classes. *Frontiers of Computer Science in China*, 1(3) :313–321, 2007.

[23] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 403–417, New York, NY, USA, 2003. ACM.

[24] Gavin Bierman, Matthew Parkinson, and James Noble. Upgradej : Incremental typechecking for class upgrades. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming : 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings*, pages 235–259, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[25] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus : A powerful live updating system. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 271–281, Washington, DC, USA, 2007. IEEE Computer Society.

[26] Stephen Magill, Michael Hicks, Suriya Subramanian, and Kathryn S. McKinley. Automating object transformations for dynamic software updating. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, October 2012.

[27] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis : Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4), August 2007.

[28] Christopher M. Hayden. *Clear, corret and efficient dynamic software updates.* PhD thesis, University of Maryland, USA, 2012.

[29] Kristis Makris and Rida A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In Geoffrey M. Voelker and Alec Wolman, editors, *2009 USENIX Annual Technical Conference, San Diego, CA, USA, June 14-19, 2009*. USENIX Association, 2009.

[30] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates : A vm-centric approach. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, 2009.

[31] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. A survey of dynamic software updating. *Journal of Software : Evolution and Process*, 25(5) :535–568, 2013.

[32] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 32–32, Berkeley, CA, USA, 2005. USENIX Association.

[33] Simon Holmbacka, Wictor Lund, Sébastien Lafond, and Johan Lilius. Lightweight framework for runtime updating of c-based software in embedded systems. In *Presented as part of the 5th Workshop on Hot Topics in Software Upgrades*, Berkeley, CA, 2013. USENIX.

[34] Weifeng Lv, Xiaochuan Zuo, and Lei Wang. Dynamic software updating for onboard software. In *Intelligent System Design and Engineering Application (ISDEA), 2012 Second International Conference on*, pages 251–253, Jan 2012.

[35] Weizhong Qiang, Feng Chen, Laurence T. Yang, and Hai Jin. Muc : Updating cloud applications dynamically via multi-version execution. *Future Generation Computer Systems*, 74 :254 – 264, 2017.

[36] Gísli Hjálmtýsson and Robert Gray. Dynamic c++ classes : A lightweight mechanism to update code in a running program. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '98, pages 6–6, Berkeley, CA, USA, 1998. USENIX Association.

[37] Jérémy Buisson and Fabien Dagnat. Recaml : Execution state as the cornerstone of reconfigurations. *SIGPLAN Not.*, 45(9) :27–38, September 2010.

[38] Stephen Gilmore, Dilsun Kéirlí, and Christopher Walton. Dynamic ml without dynamic types. Technical report, ECS-LFCS-97-378, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1997.

[39] Kristis Makris. *Whole-program Dynamic Software Updating*. PhD thesis, Arizona State University, USA, 2009.

[40] M. Wahler and M. Oriol. Disruption-free software updates in automation systems. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, pages 1–8, Sept 2014.

[41] A.C. Noubissi, J. Iguchi-Cartigny, and Jean-Louis. Lanet. Convergence osgi-javacard : fine grained dynamic update. In *Eurosmart Smart Card Security Conference and Java Card, e-Smart10, Sophia Antipolis, France.*

[42] Junqing Chen and Linpeng Huang. Dynamic service update based on osgi. In *Software Engineering, 2009. WCSE '09. WRI World Congress*, volume 3, pages 493–497, May 2009.

[43] Meik Felser, Rüdiger Kapitza, Jürgen Kleinöder, and Wolfgang Schröder-Preikschat. Dynamic software update of resource-constrained distributed embedded systems. In Achim Rettberg, Mauro C. Zanella, Rainer Dömer, Andreas Gerstlauer, and Franz J. Rammig, editors, *Embedded System Design : Topics, Techniques and Trends*, volume 231 of *IFIP – The International Federation for Information Processing*, pages 387–400. Springer US, 2007.

[44] Jianhua Liu and Weiqin Tong. A framework for dynamic updating of service pack in the internet of things. In *2011 International Conference on Internet of Things (iThings/CPSCom),*

and 4th International Conference on Cyber, Physical and Social Computing, pages 33–42, Oct 2011.

[45] Christopher M. Hayden, Edward K. Smith, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. Evaluating dynamic software update safety using systematic testing. *IEEE Trans. Software Eng.*, 38(6) :1340–1354, 2012.

[46] Luís Pina and Michael Hicks. Tedsuto : A general framework for testing dynamic software updates. In *Proceedings of the IEEE 8th International Conference on Software Testing, Verification and Validation*, ICST '16. IEEE, April 2016.

[47] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng.*, 22(2) :120–131, February 1996.

[48] Christopher M. Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster. Specifying and verifying the correctness of dynamic software updates. In *Proceedings of the 4th International Conference on Verified Software : Theories, Tools, Experiments*, VSTTE'12, pages 278–293, Berlin, Heidelberg, 2012. Springer-Verlag.

[49] Min Zhang, Kazuhiro Ogata, and Kokichi Futatsugi. An algebraic approach to formal analysis of dynamic software updating mechanisms. In Karl R. P. H. Leung and Pornsiri Muenchaisri, editors, *19th Asia-Pacific Software Engineering Conference, APSEC 2012, Hong Kong, China, December 4-7, 2012*, pages 664–673. IEEE, 2012.

[50] Gabrielle Anderson and Julian Rathke. Dynamic software update for message passing programs. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems*, volume 7705 of *Lecture Notes in Computer Science*, pages 207–222. Springer Berlin Heidelberg, 2012.

[51] G. Anderson. *Behavioural Properties and Dynamic Software Update for Concurrent Programs*. PhD thesis, University of Southampton, UK, 2013.

[52] Stefanie Rinderle, Manfred Reichert, and Peter Dadam. Correctness criteria for dynamic changes in workflow systems : A survey. *Data Knowl. Eng.*, 50(1) :9–34, July 2004.

[53] G. Stoyle. *A Theory of Dynamic Software Updates*. PhD thesis, University of Cambridge, UK, 2006.

[54] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 13–24, New York, NY, USA, 2009. ACM.

[55] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 37–49, New York, NY, USA, 2008. ACM.

[56] Y. Murarka and U. Bellur. Correctness of request executions in online updates of concurrent object oriented programs. In *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, pages 93–100, Dec 2008.

[57] Jérémy Buisson, Fabien Dagnat, Elena Leroux, and Sébastien Martinez. Safe reconfiguration of coqcots and pycots components. *Journal of Systems and Software*, 122 :430 – 444, 2016.

[58] A.C. Noubissi, J Iguchi-Cartigny, and Jean-Louis Lanet. Hot updates for java based smart cards. In *2011 IEEE 27th International Conference on Data Engineering Workshops (ICDEW)*, pages 168–173, April 2011.

[59] Min Zhang, Kazuhiro Ogata, and Kokichi Futatsugi. Towards a formal approach to modeling and verifying the design of dynamic software updates. In Jing Sun, Y. Raghu Reddy, Arun

Bahulkar, and Anjaneyulu Pasala, editors, *2015 Asia-Pacific Software Engineering Conference, APSEC 2015, New Delhi, India, December 1-4, 2015*, pages 159–166. IEEE Computer Society, 2015.

[60] Nathaniel Charlton, Ben Horsfall, and Bernhard Reus. Formal reasoning about runtime code update. In *ICDE Workshops*, pages 134–138. IEEE, 2011.

[61] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing dynamic software updating. pages 13–23, 2003.

[62] Ophir Frieder and Mark E. Segal. On dynamically updating a computer program : From concept to prototype. *J. Syst. Softw.*, 14(2) :111–128, February 1991.

[63] Masatomo Hashimoto. A method of safety analysis for runtime code update. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, volume 4435 of *Lecture Notes in Computer Science*, pages 60–74. Springer Berlin Heidelberg, 2007.

[64] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. Opus : Online patches and updates for security. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, 2005.

[65] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. *SIGOPS Oper. Syst. Rev.*, 41(3) :327–340, March 2007.

[66] Y. Murarka, U. Bellur, and R.K. Joshi. Safety analysis for dynamic update of object oriented programs. In *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*, pages 225–232, Dec 2006.

[67] Austin Anderson and Julian Rathke. Migrating protocols in multi-threaded message-passing systems. In *Proceedings of the 2Nd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, pages 8 :1–8 :5, New York, NY, USA, 2009. ACM.

[68] Junqing Chen, Linpeng Huang, Siqi Du, and Wenjia Zhou. A formal model for supporting frameworks of dynamic service update based on osgi. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 234–241, Nov 2010.

[69] M. Solarski. *Dynamic upgrade of distributed software components*. PhD thesis, University of Berlin, Germany, 2004.

[70] Valerio Panzica La Manna. Dynamic software update for component-based distributed systems. In *Proceedings of the 16th International Workshop on Component-oriented Programming*, WCOP '11, pages 1–8, New York, NY, USA, 2011. ACM.

[71] Jiankun Wu, Linpeng Huang, and Dejun Wang. Asm-based model of dynamic service update in osgi. *SIGSOFT Softw. Eng. Notes*, 33(2) :8 :1–8 :8, March 2008.

[72] John Rushby. Formal methods and their role in the certification of critical systems. In Roger Shaw, editor, *Safety and Reliability of Software Based Systems*, pages 1–42. Springer London, 1997.

[73] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9) :8–24, 1990.

[74] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[75] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[76] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7) :1165–1178, 2008.

[77] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 26(1) :53–56, January 1983.

[78] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T Leavens, K Rustan, M Leino, and Erik Poll. An overview of jml tools and applications. *Electronic Notes in Theoretical Computer Science*, 80 :75–91, 2003.

[79] Marie-Laure Potet and Yann Rouzaud. Composition and refinement in the b-method. In Didier Bert, editor, *B'98 : Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science*, pages 46–65. Springer Berlin Heidelberg, 1998.

[80] Carroll Morgan. *Programming from Specifications (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1994.

[81] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[82] Nachum Dershowitz and Jean-Pierre Jouannaud. Handbook of theoretical computer science (vol. b),. chapter Rewrite Systems, pages 243–320. MIT Press, Cambridge, MA, USA, 1990.

[83] Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(4), 2009.

[84] Min Zhang, Kazuhiro Ogata, and Kokichi Futatsugi. Formalization and verification of behavioral correctness of dynamic software updates. *Electr. Notes Theor. Comput. Sci.*, 294 :12–23, 2013.

[85] Min Zhang, Kazuhiro Ogata, and Kokichi Futatsugi. Verifying the design of dynamic software updating in the ots/cafeobj method. In *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi*, pages 560–577, 2014.

[86] Kazuhiro Ogata and Kokichi Futatsugi. Proof scores in the ots/cafeobj method. In Elie Najm, Uwe Nestmann, and Perdita Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 2884 of *Lecture Notes in Computer Science*, pages 170–184. Springer Berlin Heidelberg, 2003.

[87] Deepak Gupta and Pankaj Jalote. On line software version change using state transfer between processes. *Softw. Pract. Exper.*, 23(9) :949–964, September 1993.

[88] International Organisation for Standardization. Iso7816, 1987.

[89] Christian Tavernier. *Les cartes à puce : théorie et mise en oeuvre*. Dunod, 2007.

[90] Samiya Hamadouche. Etude de la sécurité d'un vérifieur de byte code et génération de tests de vulnérabilité, mémoire de magister, université de boumerdes, algérie.

[91] Ahmadou Al Khary Séré. *Tissage de contremesures pour machines virtuelles embarquées*. PhD thesis, University of Limoges, France, 2010.

[92] Jean-Louis Lanet. Cartes á puce, support de cours, université de limoges, france, 2006.

[93] Damien Sauveron. *Etude et réalisation d'un environnement d'expérimentation et de modélisation pour la technologie Java Card TM. Application á la sécurité*. PhD thesis, Université Bordeaux I, France, 2004.

[94] Le site de multos. https ://www.multos.com/. Accessed :2016-09-30.

[95] Le site de basic card. https ://www.basiccard.com/. Accessed :2016-09-30.

[96] Sun Microsystems Inc. Java card^{TM} 2.1 virtual machine specification. 1999.

[97] Sun Microsystems Inc. Java card^{TM} 2.1 application programming interface specification. 1999.

[98] Sun Microsystems Inc. Java card^{TM} 2.1 runtime environment specification. 1999.

[99] Zhiqun Chen. *Java Card Technology for Smart Cards : Architecture and Programmer's Guide*. Addison-Wesley Professional, 2000.

[100] Sun Microsystems Inc. Java card^{TM} 2.2.2 virtual machine specification. 2006.

[101] Sun Microsystems Inc. Java card^{TM} 2.2.2 application programming interface specification. 2006.

[102] Sun Microsystems Inc. Java card^{TM} 2.2.2 runtime environment specification. 2006.

[103] Sun Microsystems Inc. Java card^{TM} 3.0.1 virtual machine specification, classical edition. 2009.

[104] Sun Microsystems Inc. Java card^{TM} 3.0.1 runtime environment specification, classical edition. 2009.

[105] Sun Microsystems Inc. Java card^{TM} 3.0.1 application programming interface specification, classical edition. 2009.

[106] Sun Microsystems Inc. Java card^{TM} 3.0.1 virtual machine specification, connected edition. 2009.

[107] Sun Microsystems Inc. Java card^{TM} 3.0.1 application programming interface specification, connected edition. 2009.

[108] Sun Microsystems Inc. Java card^{TM} 3.0.1 runtime environment specification, connected edition. 2009.

[109] Sun Microsystems Inc. Java card^{TM} 3.0.1 servlet specification, connected edition. 2009.

[110] Xavier Leroy. Java bytecode verification : An overview. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer, 2001.

[111] Xavier Leroy. On-card bytecode verification for java card. In Isabelle Attali and Thomas P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*, volume 2140 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 2001.

[112] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the java bytecode language. In Bjørn N. Freeman-Benson and Craig Chambers, editors, *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998.*, pages 310–327. ACM, 1998.

[113] Stephen N. Freund and John C. Mitchell. A formal framework for the java bytecode language and verifier. In Brent Hailpern, Linda M. Northrop, and A. Michael Berman, editors, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999.*, pages 147–166. ACM, 1999.

[114] Stephen N. Freund and John C. Mitchell. A type system for the java bytecode language and verifier. *J. Autom. Reasoning*, 30(3-4) :271–321, 2003.

[115] Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard P. Serpette, and Simão Melo de Sousa. A formal executable semantics of the javacard platform. In David Sands, editor, *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2028 of *Lecture Notes in Computer Science*, pages 302–319. Springer, 2001.

[116] Guillaume Dufay. *Vérification formelle de la plate-forme Java Card*. PhD thesis, Université de Nice - Sophia Antipolis, France, 2003.

[117] Antoine Requet, Ludovic Casset, and Gilles Grimaud. Application of the B formal method to the proof of a type verification algorithm. In *5th IEEE International Symposium on High-Assurance Systems Engineering (HASE 2000), 15-17 November 2000, Albuquerque, NM, USA, Proceedings*, pages 115–124. IEEE Computer Society, 2000.

[118] George C. Necula. Proof-carrying code. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97 : The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 106–119. ACM Press, 1997.

[119] Claude Marché and Nicolas Rousset. Verification of JAVA CARD applets behavior with respect to transactions and card tears. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India*, pages 137–146. IEEE Computer Society, 2006.

[120] Site de la plateforme why3. http ://why3.lri.fr/. Accessed :2018-02-25.

[121] Wojciech Mostowski. Formalisation and verification of java card security properties in dynamic logic. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering*, pages 357–371, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[122] Wojciech Mostowski. *Formal Development of Safe and Secure Java Card Applets*. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2005.

[123] David Harel. Dynamic logic. In *Handbook of philosophical logic*, pages 497–604. Springer, 1984.

[124] Fabian Bannwart and Peter Müller. A program logic for bytecode. *Electr. Notes Theor. Comput. Sci.*, 141(1) :255–273, 2005.

[125] Lilian Burdy, Marieke Huisman, and Mariela Pavlova. Preliminary design of bml : A behavioral interface specification language for java bytecode. In *International Conference on Fundamental Approaches to Software Engineering*, pages 215–229. Springer, 2007.

[126] June Andronick, Boutheina Chetali, and Olivier Ly. Using coq to verify java cardtm applet isolation properties. In David Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics*, pages 335–351, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[127] Werner Dietl, Peter Müller, and Arnd Poetzsch-Heffter. A type system for checking applet isolation in java card. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 129–150, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[128] June Andronick and Quang Huy Nguyen. Certifying an embedded remote method invocation protocol. In Roger L. Wainwright and Hisham Haddad, editors, *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, pages 352–359. ACM, 2008.

[129] Quang Huy Nguyen and Boutheina Chetali. Certifying native java API by formal refinement. In Josep Domingo-Ferrer, Joachim Posegga, and Daniel Schreckling, editors, *Smart*

Card Research and Advanced Applications, 7th IFIP WG 8.8/11.2 International Conference, CARDIS 2006, Tarragona, Spain, April 19-21, 2006, Proceedings, volume 3928 of *Lecture Notes in Computer Science*, pages 313–328. Springer, 2006.

[130] Agnés-Cristelle Noubissi, J. Iguchi-Cartigny, and Jean-Louis Lanet. Incremental dynamic update for java-based smart cards. In *2010 Fifth International Conference on Systems (ICONS)*, pages 110–113, April 2010.

[131] Agnés-Cristelle Noubissi, J. Iguchi-Cartigny, and Jean-Louis Lanet. Carte à puce, vers une durée de vie infinie. In *MajecSTIC : MAnisfestation des JEunes Chercheurs en Sciences et Technologies de l'Information et de la Communication*, 2009.

[132] Julien Iguchi-Cartigny. *Contributions á la sécurité des Java Card*. Mémoire d'habilitation à diriger des recherches, Université de Limoges, 2014.

[133] Arie van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10 :2002, 2001.

[134] Razika Lounas, Mohamed Mezghiche, and Jean-Louis Lanet. Towards a general framework for formal reasoning about java bytecode transformation. In Adel Bouhoula, Tetsuo Ida, and Fairouz Kamareddine, editors, *Proceedings Fourth International Symposium on Symbolic Computation in Software Science, SCSS 2012, Gammarth, Tunisia, 15-17 December 2012.*, volume 122 of *EPTCS*, pages 63–73, 2012.

[135] Razika Lounas, Mohamed Mezghiche, and Jean-Louis Lanet. An approach for formal verification of updated java bytecode programs. In Belgacem Ben Hedia and Florin Popentiu Vladicescu, editors, *Proceedings of the 9th Workshop on Verification and Evaluation of Computer and Communication Systems, VECoS 2015, Bucharest, Romania, September 10-11, 2015.*, volume 1431 of *CEUR Workshop Proceedings*, pages 51–64. CEUR-WS.org, 2015.

[136] Razika Lounas, Mohamed Mezghiche, and Jean-Louis Lanet. A formal verification of dynamic updating in a java-based embedded system. *International Journal of Critical Computer-Based Systems*, 7(4) :303–340, 2017.

[137] Raymie Stata and Martín Abadi. A type system for java bytecode subroutines. In David B. MacQueen and Luca Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 149–160. ACM, 1998.

[138] Razika Lounas, Nisrine Jafri, Axel Legay, Mohamed Mezghiche, and Jean-Louis Lanet. *A Formal Verification of Safe Update Point Detection in Dynamic Software Updating*, pages 31–45. Springer International Publishing, Cham, 2017.

[139] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[140] G. Holzmann. *The SPIN Model Checker : Primer and Reference Manual, 1st edn.* Addison-Wesley Professional, 2003.

[141] Le guide d'utilisation de l'outil modex. http ://spinroot.com/modex/MANUAL.html. Accessed :2016-06-05.

[142] Razika Lounas, Mohamed Mezghiche, and Jean-Louis Lanet. Mise à jour dynamique des applications java card. Une approche pour une mise à jour sûre du tas. In *Proceeding de la 3ème Conférence en IngénieriE du Logiciel*, CIEL'14, 2014.