Document d'Habilitation à Diriger des Recherches

Université de Méditerranée

Produire des Logiciels Sûrs

Contribution pour la construction de systèmes enfouis

Jean-Louis LANET

Le 9 mars 2004

Composition du Jury :

Joseph Sifakis	Président
Egon Börger	Rapporteur
Thomas Jensen	Rapporteur
Pierre Paradinas	Rapporteur
Daniel Le Metayer	Rapporteur
Gilles Barthe	Examinateur
Traian Muntean	Examinateur
Peter Ryan	Examinateur

A mon père qui aurait été fier

Remerciements

La majorité de mes recherches sont le fruit d'efforts collectifs. En particulier, le développement du vérifieur GemClassifier est le résultat des travaux effectués en collaboration avec Ludovic Casset, Antoine Requet et Lilian Burdy. Qu'ils trouvent ici, ainsi que tous ceux avec lesquels j'ai travaillé sur d'autres sujets, l'expression de ma reconnaissance.

Je tiens à souligner ma reconnaissance à Monsieur le Professeur Pierre Paradinas pour les conseils et encouragements qu'il m'a prodigués durant les années passées dans son département et sans qui ce travail n'aurait pas pu voir le jour.

Je suis particulièrement reconnaissant à Egon Börger Professeur à l'université de Pise, et Thomas Jensen Directeur de Recherche au CNRS, Daniel Le Metayer de la société Trusted Logic qui me font l'honneur de rapporter sur les résultats présentés dans ce manuscrit.

Je remercie Traian Muntean, Professeur à l'Université de Méditérannée, pour m'avoir encouragé à soutenir cette habilitation.

Je suis très honoré de la présence à ce jury de Peter Ryan, Professeur à Newcastle, Joseph Sifakis Directeur de Recherche au CNRS. Et je les remercie d'avoir bien voulu examiner ce travail.

Je ne saurais oublier les remarques très pertinentes et très constructives d'El Jefe que je remercie sincèrement.

Enfin je remercie Pascaline, Pauline et Antoine pour leur soutien au quotidien, leur patience et leur compréhension.

Tables des matières

Avant Propos 9

1. Introduction 11
   1.1 Systèmes sûrs et les besoins en qualité 11
   1.2 Petits objets portables de sécurité et besoin de vérification 13
   1.3 Construction correcte de système enfouis 14
   1.4 La détection de flux illicites 15
   1.5 Le test d'applications Java Card 15
   1.6 Contexte de ces recherches 16
   1.7 Une synthèse de la recherche autour des cartes à puce 17

2. Evitement de faute pour systèmes enfouis 19
   2.1 Positionnement 19
   2.2 Principe de décomposition d'une machine virtuelle 20
   2.3 Spécification de l'algorithme de vérification de FACADE 25
   2.4 Le vérifieur de GemClassifier 33
   2.5 Plan d'évaluation et collecte de métriques 40
   2.6 Conclusions 44

3. Elimination de fautes dans les systèmes enfouis 47
   3.1 Détection de flux illicites 47
      3.1.1. Positionnement 48
      3.1.2. Non interférence 49
      3.1.3. Modèle des dépendances sûres et son implémentation 51
      3.1.4. Conclusions et extension 55
   3.2 Test d'application 55
      3.2.1. Techniques de test : positionnement 56
      3.2.2. Le contexte Java et UML 57
      3.2.3. Modèle UML testable 57
      3.2.4. Génération d'objectif de test à partir de schéma de test 58
      3.2.5. Conclusion sur la méthode 59
      3.2.6. Une alternative : JACK 60

4. Perspectives 63
   4.1 Vérification statique embarquée et système de type 63
      4.1.1. Flux illicites 63
      4.1.2. Contrôle des ressources 64
      4.1.3. Sécurité des OS 64
   4.2 Technique de preuve et de test 64
   4.3 La certification un vrai vecteur ou mirage ? 65

Conclusions 67

Références 68

Bibliographie annexe 73

Table des figures

Figure 1 Instructions élémentaires de la machine virtuelle 21
Figure 2 Représentation de l'état de notre modèle 22
Figure 3 Instruction **sinc** de la machine virtuelle défensive 22
Figure 4 Catégories des instructions de flots de contrôle 23
Figure 5 Prédicat garantissant le confinement à l'intérieur d'une méthode 23
Figure 6 Instruction **sinc** avec un test statique du flot de contrôle. 23
Figure 7 Propriétés de la frame pour les instructions y accédant. 24
Figure 8 Interpréteur abstrait offensif de l'instruction **sinc** 24
Figure 9 Instruction **sinc** dans le dernier raffinement 24
Figure 10 Sémantique statique des instructions de FACADE 28
Figure 11 Exemple d'un programme FACADE 28
Figure 12 Inférence de type pour le programme FACADE 29
Figure 13 Vérification dans la carte du code FACADE 30
Figure 14 Propriété de bon typage pour l'instruction **JumpIf** 31
Figure 15 Spécification du vérifieur au niveau le plus abstrait. 31
Figure 16 Vérification partielle de la méthode 32
Figure 17 Spécification de l'instruction **JumpIf** 33
Figure 18 Spécification informelle de l'instruction **aaload** 36
Figure 19 Réécriture de l'instruction **aaload** 37
Figure20Modélisation abstraite de l'instruction **aaload** 38
Figure 21 Cycle de développement complet 39
Figure 22 Métriques sur le développement formel : complexité 42
Figure 23 Métriques sur le développement formel : performance du logiciel 43
Figure 24 Métriques sur le développement formel : coût et qualité 43
Figure 25 Non interférence entre des entrées secrètes et des sorties publiques 49
Figure 26 Canal implicite : recopie bit par bit 50
Figure 27 Canal caché par non terminaison 50
Figure 28 Canal caché par opération partielle 50

Avant Propos

Ce mémoire retrace une grande partie des activités de recherche que j'ai mené depuis ma thèse. Ces travaux peuvent être vus comme des approches à la fiabilité du logiciel en apportant des réponses soit aux problèmes de garantie d'exécution dans les systèmes temps réel, soit à la validation et la vérification de programmes séquentiels. J'ai choisi de présenter plus spécifiquement ce dernier aspect car il représente mes plus récentes recherches et surtout les plus abouties en particulier dans le domaine des cartes à puce.

Mes travaux de recherche ont commencé en 1984 au laboratoire d'études avancées de la société Elecma, la division électronique de la Snecma. Mes premiers travaux ont porté sur la conception de systèmes sûr de fonctionnement dans le domaine des calculateurs de contrôle commande pour turbo réacteur *(i.e.* système temps réel dur). A ce titre, j'ai participé à la conception et au développement d'un calculateur réparti en anneau : système d'exploitation, application distribuée et protocole de reconfiguration dynamique. Je me suis intéressé à l'ordonnancement temps réel réparti et aux problèmes de décision d'ordonnançabilité en ligne c'est-à-dire de pouvoir accepter ou refuser de nouvelles tâches sans remise en cause des tâches précédemment acceptées. Le critère d'optimisation était l'équilibrage de charge sous contraintes de précédence et la duplication de tâches nécessaire à la sauvegarde des contextes d'exécution nécessaire pour pallier la défaillance d'un processeur. Différents algorithmes ont été simulés sous les hypothèses de l'application distribuée (tailles, communication, relations…) et les problèmes de transition lors d'un changement de mode issu d'une reconfiguration abordés. Nous avons étudié les relations entre les contextes d'exécution anciens et nouveaux. Ces travaux ont abouti à un démonstrateur essayé sur banc moteur hydromécanique avec démonstration de la reconfiguration et de son absence d'impact sur l'application de régulation moteur. Ces travaux ont fait l'objet de mon DEA, de ma thèse et d'un certain nombre de publications [Lan-92], [Lan-95c], [Bic-96a] et [Lan-96].

Depuis 1996, au sein du laboratoire de recherche de Gemplus je dirige une équipe de chercheurs et nous travaillons principalement sur la sûreté de fonctionnement par évitement et par élimination de fautes. Les méthodes formelles sont le pivot de mes trois axes de recherche :

- Evitement de faute par construction correcte de logiciel tant au niveau du système d'exploitation qu'au niveau applicatif,
- Elimination de faute par automatisation de la production de suite de test à partir de modèles,
- Techniques d'analyse pour la certification de logiciel.

J'ai rejoint l'Institut de Recherche en Informatique et Automatique en 2003 pour participer à la valorisation des travaux de recherche de l'institut propre à ce domaine. Je contribue également au projet de recherche Everest[1] sur les environnements de vérification et la sécurité du logiciel.

[1] http://www-sop.inria.fr/everest/

1. Introduction

Mon travail de recherche a porté sur la production de logiciels sûrs. La sûreté de fonctionnement est un enjeu majeur des systèmes enfouis car la défaillance d'un élément peut avoir des répercussions dramatiques sur le système soit d'un point de vue économique soit d'un point de vue humain. Ce sont aujourd'hui les industries liées à la sécurité qui sont les plus sensibles à la qualité du logiciel. Elles représentent des niches pour des technologies basées sur la modélisation, l'évitement de faute par conception ou par élimination. Cependant d'autres domaines liés à des marchés de masse peuvent être sensibles aux gains potentiels de ces technologies si ces dernières peuvent démontrer leur aptitude à passer l'échelle de programmes de taille importante et démontrer aussi leur viabilité économique. Nous avons principalement investigué des technologies comme la preuve, le test ou l'analyse statique avec comme pivot la modélisation des systèmes.

J'aborde dans ce document le domaine du génie logiciel et principalement la validation et la vérification de logiciel. Les domaines d'application sont les systèmes enfouis et particulièrement les calculateurs de contrôle commande [Lan-95a], [Lan-95b], [Bic-96b] et [Hub-96] et les petits systèmes portables de sécurité comme les cartes à puce [Gri-99], [Lan-00], [Lan-01], [Cas-02a] et [Lan-02b]. Dans ce dernier, les techniques formelles s'appliquent particulièrement bien alors que dans le cas des systèmes de contrôle commande nous vérifions principalement des propriétés non fonctionnelles liées aux caractéristiques temps réel du processus contrôlé.

Dans le domaine de la conception correcte de systèmes, nous avons démontré la correction de l'implémentation d'un vérifieur de byte code, la correction de la spécialisation d'un optimiseur de byte code et la correction du mécanisme de vérification d'un langage dédié à la carte à puce. Nous nous sommes aussi intéressé à la vérification de séquence d'ordonnançabilité et la vérification de protocoles comme DVB [Lan-97], T=1 [Lan-98a] et [Lan-98b], T=CL et une spécialisation de TCP/IP pour la carte à puce.

Dans le cadre de l'évitement de faute, nous avons voulu intégrer le test dans une approche plus générale de conception objet qu'est la méthode UML pour le test de conformité. La certification de programme a porté sur la détection de flux illicite d'information dans le cadre de Java Card et aussi sur l'audit sécuritaire de code.

Nous avons choisi de décrire ici notre thème majeur de travail l'évitement de faute en s'appuyant sur la construction correcte d'un vérifieur de byte code et l'axe complémentaire l'évitement de faute en décrivant la vérification de flux illicites et le test d'application Java Card. Nous détaillons ces sujets et donnons une description du contexte de ces travaux. Mais auparavant nous rappelons le contexte général dans lequel se placent ces travaux : la sûreté de fonctionnement des systèmes enfouis et particulièrement les petits objets portables de sécurité.

1.1 Systèmes sûrs et les besoins en qualité

Les systèmes embarqués ont des besoins en qualité différents et donc des méthodes et outils adaptés. Il existe différentes techniques allant de l'audit de code à la vérification formelle. Nous

pouvons considérer deux types de systèmes embarqués où la non-qualité à des impacts industriels très importants :

- les systèmes mettant en jeux des vies humaines (aéronautique, centrale de production d'énergie, système ferroviaire, etc.), ces systèmes sont généralement très complexes, avec des coûts de développement élevés, un faible volume de production et un encadrement de certification très stricte. Dans ce type d'application, l'adoption de technologies nouvelles est plus facile mais rarement réalisée car la taille des logiciels est élevée.
- les systèmes pour lesquels une conséquence économique est catastrophique comme un produit de masse (automobile, électroménager, etc.). Ces produits à faible marge est généralement de complexité plus faible.

Pour réaliser des systèmes sûrs de fonctionnement, deux techniques complémentaires que sont la prévention des fautes et la tolérance aux fautes sont en général utilisées [Avi-01].

La prévention consiste à réduire la présence de fautes soit par des techniques d'élimination d'erreur soit par des techniques d'évitement et, par conséquent à identifier les actions les plus appropriées à mener pour l'amélioration du processus de développement du système. Parmi ces techniques nous pouvons citer l'utilisation de composants fiables, de méthodologies de développement du matériel et du logiciel, et des méthodologies de fabrication. Ces techniques d'évitement, bien que nécessaires, n'éliminent jamais toutes les fautes. De plus, parvenu à un certain stade, l'augmentation des mesures d'évitement a un effet marginal sur la probabilité de présence de fautes résiduelles. Ainsi, l'évitement de fautes n'éliminant pas les défaillances de composants même s'il retarde leur occurrence, il est nécessaire de développer les systèmes en les dotant de mécanismes de tolérance aux fautes. Si la tolérance aux fautes est un mécanisme acceptable dans un environnement (*e.g.* calculateurs aéronautique) il peut devenir totalement inacceptable si le coût de fabrication devient le premier critère commercial. Dans ce contexte, seules les techniques d'évitement ou d'élimination sont possibles.

L'évitement des fautes cherche à prévenir la présence de fautes ou l'occurrence de pannes touchant la structure du système. Ceci est obtenu par exemple en contraignant la démarche de conception par des règles régissant celle-ci ou l'utilisation de techniques garantissant par construction l'absence de fautes. Parmi les méthodes possibles, il est possible d'utiliser des spécifications formelles associées à des techniques de validation par la preuve et à la génération automatique de code à partir des spécifications.

L'élimination des fautes cherche à détecter la présence de fautes puis à les localiser dans le but de les extraire ensuite. Les diverses techniques de test fonctionnel ou structurel répondent par exemple à ces objectifs. De nombreux travaux de recherche ont porté sur l'utilisation de modèles pour générer de manière automatique des suites de test.

Ces deux techniques ont un point commun l'utilisation de modèles et de différentes techniques de vérification et de génération de code. Si l'utilisation de techniques formelles dans le milieu académique est bien répandue, il n'en est pas de même dans l'industrie (sauf peut-être dans le domaine de la vérification de circuits). Rares sont les industries pour lesquelles l'utilisation de techniques formelles est intégrée dans leur processus de développement. Au mieux ces techniques restent dans des laboratoires d'études avancées, au pire elles ne font que passer. Le contexte de la carte à puce est différent de celui que l'on rencontre généralement. La défaillance d'une carte à puce n'a pas de conséquences catastrophiques pour l'individu, contrairement aux systèmes de transport ou d'énergie. Par contre les pertes financières peuvent être très importantes. Aujourd'hui nous sommes convaincus que ces méthodes sont techniquement applicables à la carte à puce, cependant il n'est absolument pas évident qu'elles soient économiquement intéressantes. Il faut se

rappeler qu'aujourd'hui les attaques contre les cartes à puce sont essentiellement des attaques physiques (attaques en courant, en temps, électromagnétisme…) et que les contre mesures actuelles sont essentiellement liées à ce type d'attaque. Les méthodes formelles pourront résoudre les attaques logiques contre des fautes d'implémentation. Comme aujourd'hui les contre-mesures à ces attaques ne sont pas prioritaires, il faut convaincre que le surcoût est acceptable. Or il n'existe aucune étude publique, précise et chiffrée, sur l'utilisation de ces techniques dans un domaine proche de la carte.

1.2 Petits objets portables de sécurité et besoin de vérification

Les travaux mentionnés dans ce document ont été réalisés dans un environnement industriel : le laboratoire de recherche de Gemplus, et ces travaux portent essentiellement sur les cartes à puce. Cependant, ces travaux peuvent être élargis aux modules de protection du contenu comme Embassy Wave[2] (EMBedded Application Security SYstem) destiné au marché des ordinateurs personnels ou bien aux téléphones portables et donc à tous petits systèmes enfouis ayant deux caractéristiques la mobilité et la sécurité. Ce composant propose une architecture où le client possède dans son PC un élément de sécurité qui réalise toutes les fonctionnalités de la carte, du commerce électronique à la gestion de droits (DRM), support pour une plate-forme TCPA, etc.

La carte à puce doit être considérée comme un mécanisme de confiance apportant à l'usager une représentation électronique dans un système informatique. Dans ce contexte, le fournisseur de solutions doit garantir à l'usager des propriétés comme la validité de l'application embarquée (de même que son support le système d'exploitation) l'intégrité de ses données, la disponibilité des services, la confidentialité de ces informations et éventuellement des traitements.

Les cartes à puce ne sont pas que de simples objets de stockage, ce sont des ordinateurs aux capacités réduites offrant un niveau de sécurité élevé permettant la manipulation de données confidentielles. Elles incorporent des fonctions de cryptographie comme le triple DES, RSA, AES ou des algorithmes de courbes elliptiques [Joy-03]. Elles permettent de construire des protocoles élaborés, en vue de protéger les données de la carte ainsi que les communications. La carte à puce est donc un sous-élément de sécurité d'un système plus vaste et sa défaillance peut entraîner l'obtention illégale de biens ou de services.

Rendre les cartes plus sûres nécessite de travailler dans différentes directions (matériel, cryptographie, qualité, méthodologie…) dont deux d'entre elles peuvent tirer parti des techniques formelles : le processus de développement (incluant le test) d'une part et le processus de certification d'autre part. Pour le développement il est certain que la complexité des systèmes ne fait que croître, de même que le nombre de fonctionnalités, au rythme de l'accroissement des capacités mémoire des puces. Pour le processus de certification, plus le niveau visé sera important plus la difficulté en terme de modélisation sera élevée.

La différenciation entre chaque fabricant se faisant généralement au niveau de la sécurité, chacun revendique un certain niveau d'assurance sécurité [CC-99], qui est validé par la soumission d'un produit à un laboratoire indépendant. Ce processus de certification est avant tout un moyen de gagner des parts de marché. Cependant dans certains pays (Allemagne, Hongrie…) un certain niveau de certification est obligatoire, en cas d'utilisation de clé de signature. Le schéma de

[2] http://www.wave.com/

certification concernant les produits de sécurité est actuellement les Critères Communs. L'utilisation des méthodes formelles intervient à partir du niveau cinq (EAL5) pour la modélisation de la politique de sécurité. Au niveau le plus élevé (EAL7) il est nécessaire que l'architecture de haut niveau soit décrite de manière formelle.

Le vecteur de la certification est intéressant pour l'utilisation des méthodes formelle de par son caractère obligatoire. Cependant nous sommes persuadés que la motivation principale doit être un critère d'amélioration de la qualité ou de la productivité. Cette amélioration peut être soit dans le développement correct soit dans le test de nos systèmes.

Nous allons présenter notre approche pour ces différents besoins que sont la construction correcte, la certification et le test avant d'aborder le contexte dans lequel se sont déroulés ces travaux.

1.3 Construction correcte de système enfouis

Eviter les faute en concevant a priori le bon système est un moyen de garantir que ce dernier répond bien aux spécifications. Ceci est d'autant plus crucial pour les fonctions de sécurité quand ces dernières deviennent complexes dans leur compréhension, leur implémentation et leur vérification. Pour les cartes à puce, certains modules sont très coûteux à tester (développement de plusieurs milliers d'applications de test) et la conséquence d'une vulnérabilité peut être grave.

Dans ce contexte des cartes à puce ouvertes, la possibilité d'autoriser le chargement de code en dehors de l'espace de confiance du fabricant de la carte induit de nouveaux risques d'atteinte à la sécurité. En effet, il n'y a aucune raison d'estimer que le nouveau code chargé a été développé de manière à ne pas interférer avec les applications existantes. Dès lors, un des principaux problèmes pour le déploiement de nouvelles applications est de pouvoir fournir l'assurance que ces nouvelles applications ne risquent pas de compromettre l'intégrité de la carte ou la confidentialité des autres données qui y sont stockées. Dans le cas de Java, la politique de sécurité est répartie dans plusieurs composants : l'interpréteur, le vérifieur, les librairies de base. Cette politique définit des propriétés devant être respectées par tout programme. Par exemple, il n'est pas possible de construire un pointeur à partir d'une valeur entière, Java étant un langage fortement typé.

Un point clef de cette politique de sécurité est le vérifieur de byte code. Cette partie de la machine virtuelle analyse statiquement les programmes chargés afin de s'assurer de leur innocuité. De plus, afin d'effectuer cette analyse, il vérifie la syntaxe (binaire) des programmes chargés. La construction correcte d'un tel vérifieur est d'une importance capitale, elle permet d'assurer la sécurité du système dans sa globalité. Nos travaux portent sur les méthodes de conception formelle du logiciel. Nous voulons apporter une preuve mathématique de la conformité de l'implémentation d'un vérifieur vis-à-vis de sa spécification en générant le code à partir de la spécification et en démontrant chaque étape de raffinement. Notre objectif est de formaliser le vérifieur de byte code pour le langage Java Card complet, et de montrer qu'une implémentation générée à partir de cette formalisation remplit les contraintes de la carte à puce.

Nous nous intéressons à l'intégration de cette technologie dans le processus de développement industriel et nous intégrons aussi les contraintes industrielles afin de garantir non seulement la qualité du logiciel mais aussi la pertinence économique d'un tel développement. Nous proposons une manière originale de spécifier un vérifieur de byte code à partir d'une machine virtuelle défensive.

1.4 La détection de flux illicites

Les cartes ouvertes multi-applicatives permettent l'ajout de fonctionnalités après la fabrication et l'émission de la carte. Cependant les contraintes de sécurité (confidentialité, intégrité et disponibilité) ne doivent pas être dégradées par l'ajout de programmes dont l'origine n'est pas a priori contrôlée. Nous proposons un cadre pour la certification d'application Java Card. Dans cette approche nous définissons une politique de sécurité multi-niveaux adaptée à ce type de carte en associant des niveaux de confiance à chaque entité et une relation d'ordre entre ces niveaux pour autoriser ou interdire des flux d'informations. Le contrôle de flux est réalisé en vérifiant que la propriété de dépendance causale est respectée par le code des applications. Pour faciliter cette vérification nous définissons des conditions suffisantes à respecter. Nous montrons que pour vérifier la sécurité il est suffisant de s'intéresser aux états d'une abstraction du programme Java où les valeurs des variables ont été remplacées par des niveaux. Finalement pour appliquer cette approche nous avons élaboré une technique de vérification compositionnelle qui permet de n'analyser que la nouvelle applet à charger dans une carte.

Pour analyser une applet nous construisons automatiquement le modèle SMV à partir du byte code Java Card et nous générons les propriétés de sécurité à vérifier. Nous utilisons ensuite le vérificateur de modèle SMV pour vérifier que les propriétés sont satisfaites par le modèle. Si elles ne le sont pas, le vérificateur fournit un contre exemple (une trace d'exécution) permettant de comprendre dans quelles conditions cette propriété est invalidée. Nous nous intéressons particulièrement à affiner le résultat par un traitement fin des instructions conditionnelles (réduction de l'incertitude) et le traitement des exceptions.

Cette recherche de canaux cachés dans un système s'inscrit dans le cadre de nos travaux sur l'élimination des fautes en utilisant des techniques d'analyse statique, d'abstraction et de vérifieur de modèle.

1.5 Le test d'applications Java Card

L'évitement de faute n'est pas toujours possible et nous devons aussi prendre en compte les techniques d'élimination des fautes. Au niveau des applications, il n'existe pas actuellement de possibilité de synthèse de code d'une spécification formelle vers le langage Java. Auquel cas, la phase de test est obligatoire puisque la translation d'un modèle abstrait vers une implémentation est manuelle et brise la chaîne de confiance dans le développement.

Ces travaux ont pour objectif de diminuer le coût de la phase de test en automatisant leur génération et en définissant une abstraction pour améliorer la généricité des suites de test. Quelle que soit la manière dont on souhaite tester une application (test fonctionnel ou de conformité, test aléatoire, test structurel …), il faut exécuter les tests sur une implantation de l'application : programme exécutable, prototype physique. Ces tests exécutables sont appelés tests concrets et doivent mettre en évidence les erreurs fonctionnelles des composants testés. C'est après l'exécution des tests que nous pourrons décider de l'arrêt du test et ce, selon certains critères. Les applications carte à puce sont écrites en Java, mais peuvent être portées sur des plates-formes bas coût écrites en C ou sur des plates-formes basées sur la technologie .NET. Dans ce cas, il faut tester des composants ayant des fonctionnalités identiques mais fonctionnant sur des technologies différentes nécessitant de définir un ensemble de tests concrets pour chaque technologie. Une amélioration consiste à ajouter un niveau d'abstraction définissant des tests abstraits.

Dans le cadre du test fonctionnel, l'ingénieur doit s'appuyer sur la spécification du composant, si elle existe, pour synthétiser des tests. Cette spécification peut être exprimée à l'aide de différents langages allant de la langue naturelle à des langages de spécification tels qu'UML. Nous proposons d'utiliser des objectifs de tests pour vérifier la conformité des applications par rapport à leur spécification. Cette approche tire profit des techniques de génération automatique optimisée, de techniques de combinaisons d'objectifs de test et de résolution du choix de données par des solveurs de contraintes.

1.6 Contexte de ces recherches

Comme nous l'avons mentionné précédemment, ces recherches ont été effectuées dans le cadre d'un laboratoire de recherche industriel. Ce laboratoire a maintenu des partenariats très étroits avec des laboratoires publiques afin d'acquérir et de transférer les technologies qu'il avait identifié comme vitales. Ces partenariats se sont concrétisés soit par le financement de post doctorant en accueil chez nos partenaires académiques pour adapter des technologies existantes aux besoins de notre domaine d'application, soit par l'encadrement de thèses industrielles dans notre laboratoire afin de développer des axes de recherches spécifiques.

La modélisation de politique de sécurité et la vérification de leur implémentation dans des applications carte à puce nous a amené à nous rapprocher de l'ONERA dont le modèle des dépendances sûres nous a semblé pertinent. Cette collaboration dans le cadre du projet PACAP[3] leur a permis de proposer un modèle adapté à notre support d'exécution et nous avons réalisé un prototype d'analyse statique implémentant la vérification de ce modèle.

L'optimisation de la phase de test est un problème récurrent pour les industriels. L'INRIA propose différents prototypes (TGV, STG, UMLAUT, CADP, BZTT, CASTING) génériques. Nous avons travaillé avec l'équipe VERTECS en finançant une post doctorante [Bou-00a] afin de prendre en compte dans UMLAUT les contraintes spécifiques de la carte. Nous avons élaboré une méthodologie pour la modélisation et la génération de schéma de test par le biais d'une thèse CIFRE [Mar-01]. Ces résultats qui furent exploités dans le projet RNTL COTE en partenariat avec l'IMAG.

Les techniques de développement rigoureuses comme la méthode B ont été introduites dès 1997 à Gemplus pour la modélisation de composants de sécurité et pour la modélisation de cibles de sécurité dans le cadre de certification. La confidentialité de ces travaux a très fortement limité la collaboration. Néanmoins en 1999 nous avons relevé un défi consistant à implémenter dans une carte un composant de sécurité connu par la communauté scientifique comme trop gourmand en ressources : le vérifieur de byte code. Ceci a été fait dans le cadre d'un projet européen Matisse en partenariat avec le CNRS (équipe de T. Muntean à Marseille dans le cadre d'une thèse co-dirigée [Cas-02b]), Aabo Academi (équipe de K. Serre à Turku) et l'Université de Southampton (équipe de M. Butler). Non seulement nous avons réussi à implémenter ce composant mais en plus nous l'avons prouvé correcte par rapport à sa spécification. Ce travail a été présenté à l'équipe de Sun Microsystem dédiée à Java Card, Visa International et BankSys. Il a fait l'objet de dépôts de brevets et de plusieurs présentations dont Java One [Cas-02e], FME [Cas-02d], DSN [Bur-02a],…

[3] http://www.gemplus.com/smart/r_d/publications/case-study/smv-model.html

Le dernier développement réalisé dans mon équipe fut le prototype JACK, un environnement destiné à la preuve de programme Java, comprenant un générateur d'obligation de preuve pour B et Simplify ainsi qu'un visualisateur de lemmes. Ce travail a été présenté à l'INRIA qui reconnu la qualité du travail et a signé un accord de transfert avec Gemplus.

Certains de ces travaux furent techniquement de belles réussites, d'autres n'ont pas obtenus les résultats escomptés ou ne furent pas transférés à la R&D de Gemplus. Nos travaux sur le test même s'ils ont permis certaines avancées nous laissent un goût d'inachevé. C'est certainement le domaine de recherche qui avait suscité le plus d'intérêt de la part de la R&D. Les outils et techniques n'étaient peut être pas suffisamment mûrs lorsque nous avons abordé ce thème. Désormais nous avons une bonne vision des différentes technologies de ce domaine et notre défi est de promouvoir la recherche d'une nouvelle technologie pouvant gérer efficacement données et traitement.

1.7 Une synthèse de la recherche autour des cartes à puce

Depuis avril 2003, je réalise une synthèse des besoins de recherche des différents acteurs de l'industrie de la carte à puce et j'analyse l'offre technologique de l'INRIA. J'ai donc dans un premier temps cherché les points de rupture c'est-à-dire les barrières à une plus grande utilisation de la carte. En amont des points de rupture viennent les menaces qui pèsent aujourd'hui sur cette industrie. Outre l'arrivée de substituts à la carte à puce, le rôle même que peut jouer la carte dans le futur et les fonctionnalités qui lui seront laissées ne sont pas garantis. De fortes pressions existent afin de voir la carte à puce comme un simple media de stockage sécurisé de clés. Rien ne permet de penser aujourd'hui que les services proposés dans la carte ne puissent pas être implémenté dans l'environnement d'accueil à coût plus faible, pour le même niveau de sécurité. Une autre option est au contraire une augmentation de ses fonctionnalités afin d'être plus accessible par des applications distribuées, d'être plus facilement programmable avec un niveau de sûreté et de sécurité au moins équivalent. Dans cette option, en rendant la carte transparente au développeur d'application et en ne nécessitant plus d'expertise pour développer il est possible de diminuer les coûts de développement et de déploiement. Cette vision n'est pas contradictoire avec la production de carte bas coût et des travaux sur les langages spécialisés, la compilation à la volée, les systèmes d'exploitation modulaires (reconfiguration statique et/ou dynamique) peuvent être une alternative aux besoins de compacité des cartes à faibles ressources matérielles.

Parmi les industriels s'intéressant à l'informatique embarquée, les industriels de la carte à puce sont ceux qui sont allé le plus loin dans l'intégration de Java dans des systèmes très contraints (nous ne parlons pas ici des PDA et téléphones portables) avec des capacités de communication, de chargement sécurisé après émission, de cryptographie etc. Celui qui maîtrisera ces technologies dans un tel environnement se verra ouvrir d'autres marchés de masse très similaires de l'informatique diffuse (l'informatique domestique, automobile, les Java Toys etc.) n'utilisant que certaines de ces fonctions. En effet, il semble établi que dans un environnement d'informatique ambiante ces très petits systèmes eux aussi connaîtront une évolution équivalente à celle de la carte à puce : de systèmes dédiés, fermés, non connectés et programmés en langage de bas niveau ou synthétisé dans un composant, ils seront amenés à devenir des systèmes ouverts, connectés et essentiellement bâtis autour d'architectures logicielles (machine virtuelle, pile de communication, composants logiciels certifiés, etc.) plus puissantes et évolués. Ce point est très important et il ne faut pas viser uniquement la carte à puce et ses besoins de sécurité mais aussi et surtout la maîtrise de la plate-forme et des outils et méthodes pour développer et déployer des applications.

Durant cette année j'ai donc recherché les différentes technologies présentes à l'INRIA ou en cours de développement qui peuvent provoquer des changements de contexte du marché. J'ai identifié différentes technologies depuis la conception modulaire et sécurisée d'OS, la spécialisation de modèle et de code, la biométrie, l'ordonnancement sous contrainte d'énergie, etc. J'ai rédigé un document de synthèse sur ce travail et fait des propositions pour la création d'une structure de recherche dédiée.

.

2. Evitement de faute pour systèmes enfouis

La conception et la réalisation d'un système sûr de fonctionnement dans le domaine des systèmes enfouis passent par des techniques de tolérance aux fautes, d'évitement de fautes ou d'élimination. Dans un système aussi contraint que la carte seule la combinaison d'élimination et d'évitement est réalisable afin de tendre vers un système exempt de faute. Nous montrons dans ce chapitre comment mettre en œuvre les techniques d'évitement depuis une approche très abstraite de preuve de concept jusqu'à la réalisation complète d'un système. Ce travail de recherche aborde un des éléments de sécurité des machines virtuelles Java : le vérifieur de byte code. La politique de sécurité de Java impose une vérification de la correction des applications avant leur exécution. Cette correction assure entre autre que le programme ne manipule pas des références, que la pile ne déborde pas etc. La correction de l'implémentation d'un tel mécanisme est évidemment très importante puisque le vérifieur est le point d'entrée de tout programme dans le système. Le premier travail donne un cadre générique au raffinement d'une machine virtuelle défensive en deux composants un vérifieur statique et un interpréteur offensif. Cette solution originale permet de conserver la cohérence du programme vérifié et du programme exécuté. Le second travail consiste à définir la sémantique statique d'un langage intermédiaire dédié pour les cartes à puce et de spécifier formellement le processus de vérification. Enfin la dernière partie montre l'application des travaux précédents à la génération d'un vérifieur de byte code pour Java Card à partir d'une spécification formelle.

2.1 Positionnement

Les premiers travaux sur la modélisation de la machine virtuelle Java débutent avec Cohen [Coh-97] qui propose une formalisation d'une machine défensive avec le système de preuve ACL2. Dans cette formalisation, le modèle de chaque instruction contient la sémantique opérationnelle qui vise à décrire le comportement de l'instruction mais aussi la sémantique statique qui exprime les tests et les conditions sous lesquelles l'instruction peut s'exécuter. Les instructions n'ont pas toutes été formalisées ni certaines fonctionnalités (interfaces, chargeur de classe,…). Ce travail a été ensuite étendu par la gestion du multi-tâche et une preuve de la sûreté du typage [Sto-01].

Stata et Abadi [Sta-98] ont été les premiers à proposer l'utilisation de règles de typage pour modéliser un vérifieur de byte code. Ces règles sont utilisées pour préciser le comportement de chaque instruction décrivant les entrées, le contexte d'exécution et elles donnent les modifications apportées par chaque instruction sur le contexte.

Freund et Mitchell [Fre-98] reprennent le même cadre pour étudier l'initialisation des objets et ils complètent le sous-ensemble de Java qu'ils considèrent en intégrant le traitement des sous-routines. Enfin, ils ajoutent le traitement des objets, des classes, des interfaces, des tableaux et des exceptions dans [Fre-99]. Dans leurs études, Freund et Mitchell définissent la sémantique opérationnelle et statique de chacune des instructions du langage qu'ils modélisent. Ils donnent également la preuve formelle de la cohérence de leur système.

Un des modèles formels les plus complets de la machine virtuelle Java est celui de Qian [Qia-98], l'auteur considère un important sous-ensemble du byte code et a pour objectif de prouver la correction de l'exécution d'après son typage statique. Ensuite, il propose la preuve d'un vérifieur

qui peut être déduit des spécifications de la machine virtuelle dont il propose le modèle formel. Dans une publication plus récente [Qia-00], les auteurs décrivent une implémentation correcte de presque tous les aspects d'un vérifieur de byte code Java. Ils considèrent le problème de la vérification comme une analyse de flot de données et ont pour objectif de formellement décrire les spécifications et ensuite d'extraire le code correspondant au vérifieur en utilisant l'outil Specware.

Dans le projet Bali, Push *et al.*[Pus-98] prouvent une partie de la machine virtuelle Java avec le prouveur Isabelle/HOL. Ils définissent également un sous ensemble de Java, µjava, et prouvent des propriétés sur ce sous ensemble. Plus précisément, ils formalisent le système de typage et la sémantique de ce langage en utilisant les travaux de Qian, ils fournissent les spécifications d'un vérifieur et prouve alors sa correction. Dans un travail plus récent [Nip-01], Nipkow présente la spécification formelle du vérifieur de byte code Java Card en Isabelle. Son idée est de fournir la preuve générique de l'algorithme de vérification puis de l'instancier avec une machine virtuelle particulière. L'algorithme de vérification de Rose [Ros-98] a été prouvé correct grâce à sa formalisation et à sa preuve en Isabelle [Kle-00].

Börger, Schmid et Stärk proposent une modélisation du langage Java et de la machine virtuelle Java à l'aide des ASMs [Sta-01]. Ils proposent notamment la modélisation itérative de sous-ensembles de Java en augmentant le sous-ensemble à chaque nouvelle extension. La spécification du langage Java est organisée en une série de cinq couches : le noyau impératif, la modularité, l'orientation objet, les exceptions et le système multi-tâches. Cette approche modulaire permet d'étudier les différents aspects du langage. Ils définissent également la spécification d'un compilateur et d'un vérifieur de byte code. Ils prouvent la correction et la complétude du compilateur par rapport à la spécification du langage Java et à la spécification du byte code correspondant [Bor-98]. Il est alors possible d'obtenir la propriété reliant la vérification d'un programme à son exécution : tout programme Java compilé est alors exécuté sans erreur de typage à l'exécution et son comportement est correct. Cette étude qui présente une formalisation cohérente des relations entre le langage Java et le byte code Java, est conséquente et il ne manque que le chargement dynamique de classe et le ramasse miette. Toutes les machines sont exécutables grâce à l'outil AsmGofer. Le problème de l'initialisation des objets par la machine virtuelle a fait l'objet d'un travail ayant permis de clarifier la spécification ambiguë de Sun [Bor-99].

On voit que la modélisation d'élément de la machine virtuelle Java a fait l'objet de nombreux travaux et particulièrement l'élément clé de la sécurité de Java, le vérifieur de byte code Java. Cependant aucun de ces travaux n'a démontré le lien entre la sémantique statique et la sémantique opérationnelle, ni généré le code d'un vérifieur à partir de sa spécification formelle. Or pour Gemplus il est très important de garantir que le code implémenté dans la carte à puce est conforme à sa spécification et que l'exécution offensive du code sera conforme à celle attendue.

Nous montrons comment nos travaux se sont articulés à partir d'un modèle générique de conception, une preuve de faisabilité et la réalisation concrète du premier vérifieur fonctionnel de byte code Java embarqué dans une carte à puce, dont le code est prouvé conforme à sa spécification.

2.2 Principe de décomposition d'une machine virtuelle

Il n'existe pas de description précise des fonctionnalités du vérifieur de byte code. Il existe des descriptions de la machine défensive Java. Cette machine défensive contient la sémantique statique et opérationnelle de chaque instruction, incluant implicitement les tests que doit faire un vérifieur de byte code ainsi qu'une explication succincte sur le principe de l'analyse de flot de données réalisée par le vérifieur.

Décomposition : Nous avons proposé dans [Cas-99] d'utiliser le mécanisme de raffinement de la méthode B afin d'obtenir une description formelle du vérifieur de byte code, à partir de la description formelle de la machine défensive. L'idée générale consiste à dériver à partir de la machine défensive un vérifieur et son interpréteur associé. Ces deux parties d'un système sont distinctes mais liées entre elles grâce au processus de raffinement. Le problème est de raffiner une exécution en deux exécutions consécutives avec un invariant liant les deux exécutions. L'équivalence que nous obtenons est une équivalence pour la propriété de typage. En effet un programme mal typé est refusé par un vérifieur de type et jamais exécuté alors que la machine défensive l'exécute jusqu'au point où l'erreur de typage est détecté.

Pour établir la conception d'un tel raffinement nous nous sommes basés sur un langage à dix instructions décrit par [Fre-98]. Nous utilisons une variation de ce sous ensemble en spécialisant les instructions *istore* et *iload* lesquelles manipulent des variables locales de type entier.

Inc Incrémente l'entier au sommet de pile	**Push** met un entier au sommet de la pile
Pop Retire un élément de la pile.	**If L** sauté au label L ou à l'instruction suivante suivant la valeur au sommet de la pile.
Istore x Enlève un entier du sommet de la pile et le range dans la variable locale x.	**Iload x** met au sommet de la pile le contenu de la variable x.
Halt achève l'exécution du programme.	**New** σ alloue un nouvel objet non initialisé de type σ au sommet de la pile.
Init σ initialise l'objet de type σ au sommet de la pile.	**Use** σ exécute une opération sur un objet initialisé de type σ.

Figure 1 Instructions élémentaires de la machine virtuelle

Dans la spécification de la machine défensive, nous décrivons le comportement de chacune des instructions en B. Lors de cette modélisation, la propriété que nous démontrons est que le flot de contrôle est correct : la vérification se faisant méthode par méthode, chaque saut d'une instruction vers une autre se fait à l'intérieur du domaine de la méthode en cours de vérification ou d'exécution, par la machine virtuelle défensive. Pour cela, dans la modélisation, nous nous dotons d'une variable, nommée *apc* qui décrit l'évolution du compteur d'instruction de la méthode (*abstract program counter*). Elle indique quelle instruction est en cours d'exécution par la machine virtuelle défensive. La propriété que nous assurons est que cette variable *apc* reste dans le domaine de la méthode. Nous mettons cette propriété en invariant de la machine abstraite de plus haut niveau.

La partie opérationnelle de la machine abstraite décrit le comportement de chacune des instructions du langage. Dans cette spécification, nous utilisons du B événementiel : chaque instruction est spécifiée à l'aide de l'instruction B **SELECT** qui se déclenche si la garde est vérifiée. Cela nous permet de définir les instructions de la méthode en cours d'exécution par la machine virtuelle défensive, comme une succession d'événements, la garde de chaque instruction étant le nom de l'instruction elle-même.

L'état est définit par :

- *apc* le compteur de programme qui indique quelle instruction sera exécutée.
- *frame_type*, contient le type des variables locales,
- *stack_types* est une séquence contenant les éléments de la pile, cette séquence est bornée par *max_stack*.

SETS

TYPE = { *byte, short, int, returnAddress, reference*}

VARIABLES
apc, *frame_type*, *stack_type*
INVARIANT
a*pc* ∈ *opcode_locations* ∧
frame_type ∈ 0..*max_locals*-1 +→ *TYPE* – {*int* } ∧
stack_type ∈ **seq**(*TYPE* – {*int*}) ∧
size(*stack_type*) <= *max_stack*

Figure 2 Représentation de l'état de notre modèle

Nous expliquons notre modélisation sur l'instruction *sinc* qui incrémente le contenu de la variable locale de type entier dont l'index est donné par le premier paramètre. Nous ne modélisons pas la sémantique relative aux valeurs. Pour avoir une sémantique complète de l'instruction il faut rajouter dans le **THEN** l'incrément du pointeur de programme.

DEFINITIONS
succ_pc(x) == x + 1 + *parameters_size*(*BYTE_to_OPCODE*(*method*(x)) ;
parameter(x, y) == *method*(x+y)
OPERATIONS
op_sinc ==
SELECT *BYTE_to_OPCODE*(*method*(pc)) = *SINC* **THEN**
IF
BYTE_to_unsigned(parameter(pc, 1)) ∈ 0..*max_locals*-1 ∧
frame_type(*BYTE_to_unsigned*(parameter(pc, 1))) = *short* ∧
succ_pc(*pc*) ∈ *opcode_locations*
THEN
pc := *succ_pc*(*pc*)
END
END

*Figure 3 Instruction **sinc** de la machine virtuelle défensive*

Dans cet exemple, le contenu de la clause **SELECT** signifie que l'opération sera activée lorsqu'une instruction de type *sinc* sera reconnue. Ensuite l'ensemble des tests contenus dans la clause **IF** aux tests effectués lors de l'exécution. Les définitions sont des macro commandes destinées à être insérées, elles facilitent ainsi la lecture des spécifications. La fonction *succ_pc* donne pour chaque *pc* de la méthode le prochain *pc* valide avec la prise en compte de la taille des paramètres. Avant d'effectuer l'opération l'interpréteur défensif vérifie que :

- le prochain byte code est valide : succ_pc(*pc*) ∈ *opcode_locations*
- l'index donné par le premier paramètre est valide : *BYTE_to_unsigned*(parameter(pc, 1)) ∈ 0..*max_locals*-1
- et il est du type attendu : *frame_type*(*BYTE_to_unsigned*(parameter(pc, 1))) = *short*

Afin de faciliter l'expression des différents byte code, nous avons réuni les byte codes dans des groupes représentatifs. Par exemple pour les propriétés sur le flot de contrôle l'ensemble des byte codes a été réparti en trois catégories.

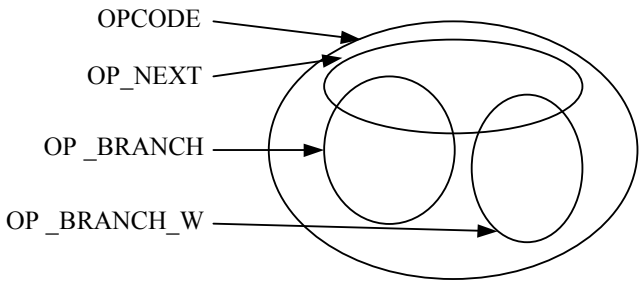

Figure 4 Catégories des instructions de flots de contrôle

L'ensemble OP_NEXT contient les byte codes faisant avancer le *pc* jusqu'au suivant après leur exécution. Nous avons placé dans OP_BRANCH les byte codes modifiant le pc avec un offset arbitraire.

Il est donc possible d'exprimer la sémantique statique pour les byte codes ensemble par ensemble contrairement au travail réalisé dans [Cas-99] où nous exprimions les contraintes de chaque instruction. Les contraintes sur le flot de contrôle sont exprimé par un prédicat B comme indiqué ci-dessous. Ce prédicat assure le confinement de l'exécution, il n'est donc plus nécessaire de réaliser ce test lors de l'exécution.

flow_checked = TRUE
⇒
. ∀*pc*.(*pc* ∈ **dom** (*method*) ∧ *BYTE_to_OPCODE* (*method* (*pc*)) ∈ *OP_NEXT*)
⇒ *pc*+1+*parameters_size*(*BYTE_to_OPCODE* (*method* (*pc*))) ∈ *opcode_locations*)) ∧
. ∀*pc*.(*pc* ∈ **dom** (*method*) ∧ *BYTE_to_OPCODE* (*method* (*pc*)) ∈ *OP_SINGLE*)
⇒ *pc* + 1 + *BYTE_to_signed* (parameter(*pc*,1)) ∈ *opcode_locations*)) ∧
. ∀*pc*.(*pc* ∈ **dom** (*method*) ∧ *BYTE_to_OPCODE* (*method* (*pc*)) ∈ *OP_SINGLE_W*)
⇒ *pc* + 1 + *BYTE_to_signed* (parameter (*pc*,1), parameter (*pc*,2)) ∈ *opcode_locations*))

Figure 5 Prédicat garantissant le confinement à l'intérieur d'une méthode

Lors du premier raffinement de l'opération *sinc*, les tests sur le flot de contrôle disparaissent et sont remplacés par le test de la variable *flow_checked*.

op_sinc ==
SELECT *BYTE_to_OPCODE* (*method* (*pc*)) = *SINC* ∧
flow_checked = TRUE
THEN
IF
BYTE_to_unsigned (parameter (*pc*, 1)) ∈ 0..*max_locals*-1 ∧
frame_type (*BYTE_to_unsigned*(parameter (*pc*, 1))) = *short* ∧
THEN
pc := succ_pc (*pc*)
END
END

Figure 6 Instruction **sinc** *avec un test statique du flot de contrôle.*

Les propriétés de la pile et de la frame sont exprimées de la même manière. Par exemple, la définition pour les instructions qui accèdent à la frame et vont à l'instruction suivante est donnée figure suivante. Comme la fonction *frame_type* n'est définie que pour les variables utilisables, il faut que la frame suivante soit incluse dans la frame courante garantissant ainsi que soit les variables sont inchangées soit perdues.

Static_frame_checked = TRUE
⇒
(∀pc.((pc ∈ opcode_location ∧ opcode (pc) ∈OP_NEXT_FRAME_READ
BYTE_to_unsigned (method(pc+1)) ∈ 0..max_locals-1 ∧
frame_type_s (pc)(BYTE_to_unsigned (method (pc+1)))=frame_type_used (opcode (pc)) ∧
frame_type_s (succ_pc(apc)) ⊆ frame_type_s (pc)))

Figure 7 Propriétés de la frame pour les instructions y accédant.

Désormais il ne reste plus que la sémantique opérationnelle dans les opérations, définissant ainsi un interpréteur offensif n'exécutant aucun test. Cet interpréteur ne travaille encore que sur des types et non sur des valeurs.

op_sinc ==
SELECT
frame_checked = TRUE ∧
stack_checked = TRUE ∧
flow_checked = TRUE
THEN
pc := succ_pc(*pc*)
END

*Figure 8 Interpréteur abstrait offensif de l'instruction **sinc***

Il faut désormais introduire dans le dernier raffinement la notion de valeur pour la pile et la frame. Notre invariant de collage stipule que chaque variable typée doit avoir une valeur.

frame_value ∈ 0..*max_locals*-1 +→ **INT** avec comme invariant de collage

dom(*frame_type*) ⊆ **dom**(*frame_value*)

Ceci nous permet de définir un dernier raffinement très proche d'une implémentation en langage impératif.

op_sinc =
SELECT *frame_checked* = TRUE ∧
stack_checked = TRUE ∧
flow_checked = TRUE
THEN
VAR *oldfvalue*, *newfvalue* **IN**
oldfvalue := *frame_value*(parameter(1)) ;
newfvalue ← sadd (*oldfvalue*, parameter(2)) ;
frame_value(parameter(1)) := *newfvalue*
END ;
pc := *pc* + 3
END

*Figure 9 Instruction **sinc** dans le dernier raffinement*

La clause **VAR** introduit des variables locales dans la spécification. Nous utilisons dans ce raffinement l'opérateur de séquence (;). La sémantique opérationnelle montre que la valeur de la variable locale pointée par le premier paramètre est poussée au sommet de pile. Puis à l'aide de l'opération *sadd* nous additionnons la valeur du second paramètre. La nouvelle valeur est stockée à nouveau dans la frame et le pointeur de programme est incrémenté de manière à pointer après les deux paramètres de l'instruction *sinc*.

Le mécanisme de raffinement garantit que chaque opération raffinée peut s'exécuter dans un état correspondant à celui de la machine abstraite et que les opérations ainsi raffinées se comportent comme les opérations abstraites. Donc démontrer que cette spécification est un raffinement valide de l'interpréteur garantit la correction du vérifieur et de l'interpréteur offensif. La grande différence entre l'interpréteur défensif et l'interpréteur abstrait est qu'il n'y a pas de correspondance stricte entre les opérations déclenchées par l'interpréteur défensif et l'interpréteur offensif. Si la méthode est vérifiable alors les machines abstraites et concrètes ont le même comportement, sinon la machine défensive exécutera le code jusqu'à la détection d'erreur alors que la machine offensive ne l'exécutera jamais.

Cette étude montre qu'il est possible d'utiliser la méthode B afin de raffiner une machine défensive en un vérifieur et un interpréteur associé. Ces travaux ont continué à Gemplus sur la méthodologie de preuve par agrégation et la spécification complète d'un interpréteur offensif [Req-03]. D'autre part une méthodologie sur la démonstration du raffinement du vérifieur et sur la preuve des invariants de boucle par [Cas-01].

Un travail plus complet mené à l'INRIA par l'équipe *Everest* qui a étendu notre principe. Dans [Bar-01] Barthe *et al.* propose une formalisation de la machine virtuelle défensive Java Card. Cette première formalisation leur sert de point de départ pour montrer la correspondance entre la machine défensive et l'interpréteur offensif Java Card associé au vérifieur de byte code. L'ensemble des instructions Java Card en utilisant le système de preuve Coq est développé dans [Bar-02].

Avec ce travail nous avons établi un mécanisme générique pour lier la sémantique statique et la sémantique opérationnelle d'un langage. Ce cadre original garantit la consistance entre la spécification d'un vérifieur statique et un interpréteur offensif gage d'une implémentation optimale d'un langage fortement typé dans un système contraint comme la carte à puce. Nous avons appliquer ce travail à un langage de programmation dédié à des petits systèmes enfouis en partenariat avec l'université de Lille destiné à être implanté dans un prototype de recherche.

2.3 Spécification de l'algorithme de vérification de FACADE

Lors de ces travaux en 1999, il était admis dans la communauté scientifique qu'embarquer un vérifieur de type dans une carte à puce était un problème sans solution. Dans sa thèse, Grimaud [Gri-00] a proposé de concevoir un langage fortement typé simple à vérifier en utilisant la technique du PCC (Proof Carrying Code) [Nec-97]. Dans un premier temps nous avons spécifié formellement la sémantique statique du langage FACADE [Gri-99] et donné le principe de l'algorithme de la vérification puis nous avons démontré la validité de l'algorithme dans le cadre de ce langage [Cas-00].

La technique du PCC consiste à ajouter une preuve de la sûreté du programme au programme lui-même. Cette preuve est constituée d'informations additionnelles permettant de garantir des propriétés de sûreté sur le programme. Ces informations additionnelles, également appelées preuves, peuvent être générées par le développeur de l'application, appelé producteur de code. Une

fois cette preuve générée, elle est transmise avec le code au client final, le consommateur de code. Celui-ci peut alors vérifier la preuve et le code en s'assurant que les deux correspondent bien et que les propriétés de sûreté sont respectées. Comme vérifier la preuve sur un programme est plus simple que la générer, l'algorithme de vérification s'adapte parfaitement aux appareils ayant de fortes contraintes en mémoire et en puissance de calcul.

Définition de la sémantique du langage FACADE : FACADE est un langage intermédiaire appartenant dédié aux cartes à puce. L'utilisation d'un tel langage permet de faciliter la vérification de l'innocuité des applications. C'est une architecture complète car l'auteur définit les traducteurs (compilateurs certifiant) adéquats pour transformer du code Java Card, Mel ou C# vers FACADE. Ensuite, il spécifie le compilateur à la volée permettant de transformer ce code intermédiaire en une représentation efficace pour l'exécution. L'innocuité des applications est donc assurée par la vérification de leur bon typage. Il est essentiel que l'algorithme soit correctement implémenté mais aussi que la sémantique du langage, nécessaire à sa vérification soit bien établie.

Notre contribution à ce travail fut la définition de cette sémantique et la formalisation de l'algorithme de vérification dont la preuve de correction fut apportée ultérieurement.

Le langage FACADE est un langage à objet et donc toute information est associée à un type qui est définit par des classes. La hiérarchie de classe est donc une structure d'arbre extensible permettant lors du chargement d'application d'étendre le système de type. Chaque classe ne possède qu'une seule super classe. Une classe particulière est *CardObject* est au sommet de la hiérarchie et représente la racine du graphe d'héritage. Ce graphe est donc défini par un semi-treillis comme un triplet $L = (V, \subseteq, \cap)$ où V est l'ensemble des types, $\subseteq$ un ordre partiel défini sur V, et $\cap$ une opération binaire, l'unification définie sur V. Deux éléments $i, j \in V$ sont dits incomparables si ni $i \subseteq j$ ni $j \subseteq i$. Dès lors leur plus grand père commun est CardObject que nous noterons *Top.* Une opération est donc sûre en terme de typage si toutes les opérations quelle effectue respectent la hiérarchie de type. Il est nécessaire que chaque opération soit appelée avec les bons types de paramètres et que le paramètre de retour soit bien celui attendu. La phase de vérification doit donc garantir que les types utilisés soient ceux définis par la sémantique statique de chaque opération élémentaire.

Ce langage comprend cinq instructions qui sont :

- *Jump labelId*, un saut inconditionnel dans le programme, *labelId* doit appartenir à la table des labels et le pointeur de programme associé *pp* doit être valide,
- *JumpIf Var labelId*, un saut conditionnel dans le programme, la variable *Var* doit être de type booléen, *labelId* doit appartenir à la table des labels et le pointeur de programme associé *pp* doit être valide,
- *JumpList Var nbcase {labelId*$_1$*, labelId*$_2$*..., labelId*$_{nbcase}$*}*, une sorte de choix conditionnels multiples avec une liste de sauts dans le programme, la variable *Var* doit être de type entier, *labelId*$_i$ doivent appartenir à la table des labels et le pointeur de programme associé *pp* doit être valide, le nombre de labels doit être inférieur ou égal à *nbcase*,
- *Return VarRes*, pour sortir d'une méthode et éventuellement retourner un résultat, le type du résultat doit être celui définit dans la signature de la méthode. Tout programme doit se terminer par cette instruction,
- *Invoke VarRes Var methodId {var1, var2,...},* pour appeler la méthode *methodId* sur la variable *Var* avec les paramètres *var1*, *var2*,…. et en retournant le paramètre *VarRes*. Pour être valide, *methodId* doit être déclarée dans la classe de *Var*, les types des paramètres d'appel et de retour doivent être ceux attendus par *methodId.*

Les programmes P sont donc des fonctions d'adresses mémoires vers des instructions. Le domaine de P est l'ensemble des adresses relatives valides. Pour simplifier la présentation nous travaillerons sur des programmes P, dont le domaine est l'ensemble des adresses (*pp*, program point) pour lesquelles il existe une instruction valide. On prend comme hypothèse qu'une vérification structurelle s'est assurée de cette translation. Un programme est donc modélisé par une suite d'instruction, des variables locales typées *Local*, et des variables de travail non typées *Temp*. Il existe une fonction L qui associe des types à chaque variables locales et une fonction T qui calcule pour chaque variable de *Temp*, le type associé à chaque point de programme *pp*. Un programme est bien typé si il existe L et T tels que $L,T \Rightarrow P$.

Lors du chargement il existe une représentation (*classDesc*) décrivant chaque classe et pour chaque classe une description de chaque méthode (*methodDesc*) par sa signature.

$$\frac{P[pp] = \text{Return } VarRes \quad VarRes : methodDsc[ThisMethod][ReturnType]}{L,T,pp \Rightarrow P}$$

$$\frac{P[pp] = \text{Jump } labelId \quad labelId \in \text{Dom}(P) \quad pp+1 \in Dom(P)}{L,T,pp \Rightarrow P}$$

$$\frac{P[pp] = \text{JumpIf } Var\ labelId \quad Var : Bool \quad labelId \in \text{Dom}(P) \quad pp+1 \in \text{Dom}(P)}{L,T,pp \Rightarrow P}$$

$$\frac{P[pp] = \text{JumpList } Var\ nbCase\ labelList \quad Var : Int \quad nbcase : Nat \quad nbcase \geq \text{Card}(label) \quad \forall i\,(i \in labelList \Rightarrow i \in \text{Dom}(P)) \quad pp+1 \in \text{Dom}(P)}{L,T,pp \Rightarrow P}$$

$$\frac{\begin{array}{c} P[pp] = \text{Invoke } VarRes\ Var\ MethodId\ tabVar \quad MethodId : methodDesc_{classDesc} \quad VarRes \in T \\ T_{pp+1} = T_{pp}[VarRes \rightarrow methodDesc[MethodId][ReturnType]] \\ \forall i, (i \in 1..methodDesc[MethodId][nbvar] \Rightarrow tabVar[i] : methodDesc[MethodId][i]) \quad pp+1 \in \text{Dom}(P) \end{array}}{L,T,pp \Rightarrow P}$$

$$P[pp] = \text{Invoke } VarRes\ Var\ MethodId\ tabVar$$
$$MethodId : methodDesc_{classDesc}$$
$$T_{pp+1} = T_{pp}$$
$$VarRes \in L$$
$$VarRes : methodDesc[MethodId][ReturnType]$$
$$\forall i, (i \in 1..methodDesc[MethodId][nbvar] \Rightarrow tabVar[i] : methodDesc[MethodId][i])$$
$$\frac{pp+1 \in \text{Dom}(P)}{L, T, pp \Rightarrow P}$$

Figure 10 Sémantique statique des instructions de FACADE

Il existe une différence dans la sémantique de l'instruction *Invoke* suivant le type de variables locales qu'elle utilise. Considérons l'exemple suivant décrivant un programme FACADE qui exécute une simple boucle. Il utilise deux variables *i* et *v* en tant que variables temporaires, le type de ces variables est donc inconnu (*Top*) avant l'exécution du programme.

pp	Label	Instruction	Commentaires
1		`Invoke i,0,likeIt`	`i initialisé à 0`
2		`Jump label0`	
3	`label1`	`Invoke i, i, add, 1`	`incrément de i`
4	`label0`	`Invoke v, i, LtOrEq, 100`	`test si inf. ou égal à 100`
5		`JumpIf v, label1`	`si vrai retour à 3`
6		`Return void`	`sinon retour`

Figure 11 Exemple d'un programme FACADE

Génération hors carte des labels :

Cet algorithme est une adaptation du PCC aux propriétés de typage. L'inférence de type correspond partiellement au calcul des *labels* et est réalisée hors de la carte. Cette partie permet de générer les informations de typage aux points de jonction. La génération des *labels* conduit au calcul de la table de typage *TT*. Cette table correspond aux informations de typage des variables *Temp* pour chaque instruction de la méthode. Ainsi, pour chaque point du programme *pp*, nous disposons du type des variables temporaires, dont le type peut évoluer au cours de l'exécution du programme. Les descripteurs de labels correspondent aux informations de typage pour une instruction particulière pointée par *pp*, c'est-à-dire le type de chaque variable temporaire *Temp*.

Avec l'application FACADE doit être fournie la table des labels qui est extraite de la table *TT*. Ce sous-ensemble de *TT* contient tous les *descripteurs de labels* concernés par des destinations de saut. Un descripteur de label est donc une table de type [*pp*, TT_{pp}]. En effet, les destinations de saut sont des points du programme où le typage des variables temporaires peut être différent étant donné qu'il existe au moins deux chemins pour y accéder, chacun des chemins pouvant avoir stocké un type différent dans les variables temporaires. Ce sont ces *labels* qui vont permettre de conclure le processus de vérification sur la carte.

Pas	*pp*	Label	Instruction	$TT_{pp}[i,v]$
1	1		`Invoke i, 0, likeIt`	*{Top,Top}*
2	2		`Jump label0`	*{Int,Top}*
3	4	`label0`	`Invoke v,i, LtOrEq, 100`	*{Int,Top}*
4	5		`JumpIf v, label1`	*{Int,Bool}*
5	3	`label1`	`Invoke i, i, add, i`	*{Int,Bool}*
6	4	`label0`	`Invoke v, i, LtOrEq, 100`	*{Int,Bool}*
7	5		`JumpIf v, label1`	*{Int,Bool}*
8	6		`Return void`	*{Int,Bool}*

Figure 12 Inférence de type pour le programme FACADE

La première colonne représente les pas dans le calcul de l'inférence de type et la dernière colonne les types avant exécution de l'instruction. Le résultat de l'exécution abstraite de l'instruction est donc visible à la ligne suivante.

Lors du premier pas de calcul de l'inférence, *i* et *v* sont de type *Top*. Dans le descripteur de méthode, est indiqué le type de retour de la méthode `likeIt` comme *Int*. Après l'exécution de cette méthode, la variable *i* prend le type *Int*. En **2**, l'exécution du saut n'affecte pas le type des variables. Lors du pas **4** un branchement conditionnel divise l'analyse en deux partie, la première est réalisée immédiatement la seconde (*i.e.* le retour) lors du pas **8**. Au pas **6**, nous revenons sur une instruction déjà visitée (pas **3**) mais avec un type différent pour TT_{pp}. Il faut donc faire l'unification des deux états ce qui donne {*Int,Top*}, car dans notre semi-treillis de types *Top* est le plus petit parent commun de *Top* et *Bool*. Au pas **7**, les états sont égaux, nous avons donc atteint un point fixe. Il est possible d'exécuter le dernier pas.

Nous avons donc deux labels : (3,[*Int*,*Bool*]) et (4, [*Int*, *Top*]). Il est possible comme montré dans [Gri-99] de réduire cette information en la compressant, ce qui nous donne la table *labelDesc* suivante : ({3,4},{*Int,Top*}).

La table des *labels* est transmise à la carte avec le code FAÇADE. La présence de ces informations permet à l'algorithme de vérification d'avoir une complexité linéaire, ce qui facilite son exécution par la carte à puce. De plus, si la présence des *labels* augmente la taille du code au chargement, elle ne modifie en rien la taille du code nécessaire à l'exécution ou à l'interprétation. En effet, ces informations ne sont utiles que pour la phase de vérification et peuvent ensuite être effacées de la carte.

Principe de l'algorithme de vérification : Le processus de vérification a toujours le même objectif : s'assurer que l'exécution d'une application ne menace pas la plate-forme qui l'exécute. Cela signifie que le système étant dans un état correct au départ, chaque opération, chaque instruction, conduit le système dans un nouvel état correct. Dans le cadre de FAÇADE, un état correct correspond à un état où les variables sont correctement typées. Il faut définir un système de transitions avec les fonctions de transfert adéquates, permettant de passer d'un état à un autre. Chaque fonction de transfert correspond aux contraintes et au comportement relatif à une instruction du langage FAÇADE. L'ensemble de ces fonctions de transfert, c'est-à-dire le système de transitions, définit la sémantique statique du langage. En particulier, il doit être assuré que :

- toutes les instructions ont le bon type d'arguments avant l'exécution de l'instruction,
- toutes les instructions utilisent et modifient les variables locales en fonction des règles de typage définies sur le langage.

Vérifier qu'un programme est correctement typé se traduit par vérifier que toutes les opérations effectuées par le programme sont légales d'après les règles de typage considérées. En particulier, il faut que tous les paramètres des méthodes appelées aient le bon type, et qu'aucune conversion de type non autorisée soit effectuée. Pour vérifier cette propriété de typage, il est nécessaire de calculer séquentiellement les informations de typage pour chaque point du programme et de vérifier la cohérence des types inférés sur le programme et des types fournis par les *labels* ajoutés au code. C'est ce qui est effectué par la carte à puce. On voit dans l'exemple suivant que les informations à sauvegarder dans la carte sont plus faibles car on ne maintient que la valeur de *TT* pour le *pp* courant. Cette table est maintenue en mémoire volatile pour des raisons d'efficacité.

Pas	pp	Instruction	labels	$T_{pp}[i,v]$
1	1	`Invoke i,0,likeIt`		*{Top,Top}*
2	2	`Jump label0`		*{Int,Top}*
3	3	`Invoke i, i, add, 1`	*{Int,Top}*	*{Int,Top}*
4	4	`Invoke v, i, LtOrEq, 100`	*{Int,Top}*	*{Int,Top}*
5	5	`JumpIf v, label1`		*{Int,Bool}*
6	6	`Return void`		*{Int,Bool}*

Figure 13 Vérification dans la carte du code FACADE

L'algorithme se déroule jusqu'au pas **3** comme décrit lors du calcul des labels. Pour ce pas là, puisqu'il existe pour le *pp* courant un descripteur de labels, ce dernier surcharge la valeur de TT_{pp}. Il en va de même au pas **4** dont l'exécution abstraite modifie le type de *v*. Dès lors au pas 5, la pré-condition du `JumpIf` est remplie.

L'algorithme de PCC adapté au langage FAÇADE est de complexité linéaire. L'adaptation faite du PCC est même plus performante que le PCC lui-même. En effet, la technique du PCC nécessite un espace mémoire de taille exponentielle pour le pire cas de la vérification. Hors, dans le cas de FAÇADE, la taille mémoire est en *O(mn)* où *m* est le nombre d'opérations du programme et *n* le nombre de variable temporaire T. Cette complexité en taille de mémoire correspond à la taille maximale de la table des *labels*. Nous pouvons alors proposer la définition du typage correct qui est la suivante :

Définition : *Une méthode est correctement typée si et seulement s'il existe une table de type TT satisfaisant les règles de typage pour cette méthode.*

La définition signifie que la méthode, et donc par association le programme, respecte les règles du typage du langage FAÇADE. Nous avons utilisé la méthode B pour spécifier l'algorithme et apporter une preuve de correction de la vérification.

Preuve de l'algorithme de vérification : ce que nous souhaitons démontrer c'est qu'un tel algorithme ne peut accepter de programmes mal typés même si **le programme et la preuve** ont été altérés. Nous ne nous attachons pas à démontrer que l'implémentation est correcte mais seulement le principe de vérification. La définition *correctement typé* nécessite d'être formalisée pour chaque instruction en suivant les règles de la sémantique statique. Nous exprimons un invariant qui est vrai si et seulement si il existe une table de types TT pour chaque instruction du programme qui satisfasse les règles de typage. La figure suivante montre partiellement cet invariant pour l'instruction `JumpIf`. En (1) la table des types pour le premier label est forcément à Top car les variables ne sont initialisées que lors de l'entrée dans la méthode. En (2) nous spécifions que la variable évaluée lors du test est un booléen ou un sous-type. L'instruction suivante doit être dans le corps de la méthode (3) de même que la destination du saut (4). Le type des variables pour

l'instruction suivante (5) ou pour l'instruction pointée par le saut (6) doit être compatible avec la table de types de l'instruction. L'invariant décrit la sémantique statique pour l'ensemble des instructions.

$$
\begin{array}{l}
WellTyped \in \mathbf{BOOL} \wedge \\
((wellTyped = \mathbf{TRUE}) \\
\Leftrightarrow \exists TT.(TT \in \mathbf{dom}(opcodes) \rightarrow LABEL\_DSC \wedge \\
\quad TT(1) = T\_VARS \times \{Top\} \wedge \qquad (1) \\
\quad \forall pp.(pp \in \mathbf{dom}(opcodes) \\
\qquad \Rightarrow \\
\qquad (opcodes(pp) = JumpIf \\
\qquad \Rightarrow \\
\qquad Bool \in ge(VarType(jumpVar(pp), TT(pp))) \wedge \qquad (2) \\
\qquad pp + 1 \in \mathbf{dom}(opcodes) \wedge \qquad (3) \\
\qquad jumpTarget(pp) \in \mathbf{dom}(opcodes) \wedge \qquad (4) \\
\qquad TT(jumpTarget(pp)) \in compatibles(TT(pp)) \wedge \qquad (5) \\
\qquad TT(pp + 1) \in compatibles(TT(pp))) \wedge \qquad (6)
\end{array}
$$

*Figure 14 Propriété de bon typage pour l'instruction **JumpIf***

L'algorithme de vérification doit nous permettre de s'assurer que la propriété suivante est garantie.

Théorème : *tout programme mal typé est rejeté.*

Pour ce faire, nous définissons au plus haut niveau la fonction de vérification comme retournant un booléen ayant la valeur vraie si notre règle de typage est vérifiée. La pré condition nécessite que les labels fournis pointent des instructions qui appartiennent à la méthode en cours de vérification.

```
result ← check_type(labels) = PRE
   labels ∈  NATURAL  +↔ LABEL_DSC ∧
   dom(labels) ⊆ dom(opcodes)
THEN
    ANY res WHERE
      res ∈ BOOL ∧
      (res = TRUE ⇒ wellTyped = TRUE)
    THEN
         result := res
    END
END
```

Figure 15 Spécification du vérifieur au niveau le plus abstrait.

Cette machine est raffinée par une boucle appelant une machine effectuant un pas de vérification jusqu'à avoir parcouru toute la méthode ou bien rencontré une erreur de typage.

Cette nouvelle machine n'effectue qu'un seul pas. On définit donc une fonction `partiallyChecked` décrivant qu'une partie de la méthode jusqu'au point de programme *pp* a été analysée, est bien typée et vérifie notre invariant. Nous introduisons la variable `abstract_labels`, qui est une fonction des adresses vers des descripteurs de labels. Cette fonction correspond à la table TT lorsque la dernière adresse a été vérifiée. Cette variable est mise à jour de manière à toujours satisfaire `partiallyChecked.` Chaque instruction validée ajoute ses informations de typage (`Tpp`) nouvellement inférées à `abstract_labels`.

```
step_result ← check_instruction = CHOICE
   step_result := FALSE
OR
   step_result := TRUE || pp := pp + 1 ||
   IF pp < methodSize THEN
     ANY new_labels WHERE
        new_labels ∈ LABEL_DSC ∧
        (pp + 1 ∈ dom (labels)
         ⇒ new_labels = labels(pp+1)) ∧ (opcodes(pp) = JumpIf
         ⇒
             Bool ∈ ge (VarType (jumpVar (pp), abstract_labels (pp))) ∧
             JumpTarget (pp) ∈ dom (labels) ∧
             labels (jumpTarget (pp)) ∈ compatibles (abstract_labels (pp)) ∧
             new_labels ∈ compatibles (abstract_labels(pp)) )
      THEN
        abstract_labels (pp + 1) := new_labels
      END
   ELSE ANY new_labels WHERE
        new_labels ∈ NATURAL +↔ LABEL_DSC ∧
        new_labels = abstract_labels ∧
        partiallyChecked (pp + 1, new_labels) = TRUE
      THEN
        abstract_labels := new_labels
      END
   END
END;
```

Figure 16 Vérification partielle de la méthode

Il faut désormais indiquer le traitement effectué au niveau de chaque instruction. Ceci est fait par une autre machine. Dans cette ultime machine, nous introduisons `Tpp` le descripteur de labels pour l'instruction courante, nous décrivons tous les tests effectués par chaque instruction et nous indiquons comment le label courant est modifié. Pour chaque instruction il faut différencier la cas où un label est fourni pour la prochaine instruction ou pas. Dans le cas où aucun label est fourni, la table Tpp est mise à jour uniquement si l'instruction est `invok` et dans ce cas le type de la variable de retour est mis à jour. Si un label est fourni il faut s'assurer que ce dernier est compatible avec le label courant après modification par l'instruction. Si c'est le cas alors `Tpp` est remplacé par le label fourni. La figure suivant montre les tests pour l'instruction `JumpIf`.

```
res ← check_jumpif = PRE
   pp ∈ dom(opcodes) ∧ opcodes(pp) = JumpIf
THEN
   SELECT
      jumpTarget(pp) ∈ dom(labels) ∧
      labels(jumpTarget(pp)) ∈ compatibles(Tc) ∧
      pp + 1 ∈ dom(opcodes) ∧
      Bool ∈ ge(VarType(jumpVar(pp), Tc)) ∧
      pp + 1 ∈ dom(labels) ∧
      labels(pp + 1) ∈ compatibles(Tc)
   THEN
      Tc := labels(pp + 1) || res := TRUE || pp := pp + 1
   WHEN
      jumpTarget(pp) ∈ dom(labels) ∧
      labels(jumpTarget(pp)) ∈ compatibles(Tc) ∧
      pp + 1 ∈ dom(opcodes) ∧
```

```
    Bool ∈ ge(VarType(jumpVar(pp), Tc)) ∧
    pp + 1 ∉ dom(labels)
  THEN res := TRUE || pp := pp + 1
  ELSE res := FALSE
  END
END;
```

*Figure 17 Spécification de l'instruction **JumpIf***

Les tests pour chaque instruction sont intégrés dans le raffinement de la vérification. Les détails complets de cette modélisation sont donnés dans [Cas-00]. Cette preuve de correction de l'algorithme du PCC adapté à la vérification de code FAÇADE repose sur plusieurs points :

- la correction de l'énoncé de la propriété de typage : cette propriété de haut niveau est le cœur du modèle. Cependant, sa formalisation est complexe car elle sous-entend un invariant qui donne la bonne propriété de typage pour toutes les instructions du langage. Si cette formalisation facilite la preuve de la machine de haut niveau, elle reporte la difficulté sur la machine spécifiant la vérification d'une instruction. La preuve reste dans le modèle, et la délocalisation permet de ne se focaliser que sur un problème à la fois, rendant le déroulement de la phase de preuve plus simple,
- la difficile implémentation du modèle : un tel modèle est difficilement implantable. Les structures de données sont complexes à raffiner. L'avantage de ce style de modèle est de pouvoir utiliser à loisir tout le pouvoir d'expression de B pour spécifier des propriétés de haut niveau comme par exemple si un programme est accepté, alors il est bien typé. C'est une propriété importante pour le vérifieur car nous assurons ainsi qu'il n'accepte jamais des programmes mal typés,
- la preuve interactive : Du fait du choix de modélisation les lemmes générés par l'*Atelier B* sont plus complexes. Il faut donc avoir recours au prouveur interactif de l'outil pour parvenir à terminer les preuves et à prouver ainsi le modèle. En fait, au cours de la phase de preuve interactive, le premier travail est d'identifier l'origine du lemme. Du fait de l'automatisation de la génération, comprendre le lemme n'est pas facile car il n'est pas toujours proposé de façon naturelle par l'outil. Dans cette première partie de la preuve interactive, il est utile de reformuler le modèle pour obtenir des lemmes plus facilement compréhensibles.

La conclusion sur cette étude est que B est parfaitement utilisable pour obtenir la preuve de concept d'un algorithme. En utilisant l'expressivité de B, l'algorithme peut être formalisé et des propriétés de haut niveau peuvent être prouvées.

Après avoir montré une façon originale de lier les sémantiques statiques et dynamiques et démontrer la correction de l'algorithme de vérification pour un langage spécifique, nous nous attachons à prouver la correction d'une implémentation d'un tel vérifieur. Le défi est bien sûr de réaliser une telle implémentation dans une carte à puce, chose qui n'avait jamais été faite lors de ces travaux, mais aussi de valider une méthodologie de développement. En effet le passage à l'échelle et l'adoption d'une technologie par une organisation passe non seulement par une réalisation concrète mais aussi par une évaluation détaillée du processus de développement.

2.4 Le vérifieur de GemClassifier

Une adaptation de la technique du PCC au langage Java a été proposée par Rose [Ros-98]. Cette adaptation est désormais utilisée par la KVM de Sun Microsystem. L'algorithme de vérification de Rose a été prouvé correct grâce à sa formalisation et à sa preuve en Isabelle [Kle-00]. Cependant il existe toujours une rupture dans la chaîne de confiance entre la spécification formelle et

l'implémentation qui peut en être faite. Nous proposons donc de générer le code à partir de la spécification en ayant eu soin d'étendre la modélisation précédente au vérifieur complet de Java. De plus, nous nous intéressons à la vérification dans son intégralité c'est-à-dire en incluant le vérifieur de structure qui à notre connaissance n'a jamais été traité. Ce travail a fait l'objet de la thèse de Ludovic Casset [Cas-02b].

Un vérifieur de byte code comprend deux parties relativement distinctes que sont la vérification de structure et la vérification de type. Ces deux parties sont distinctes par plusieurs aspects :

- d'un point de vue purement fonctionnel, ce sont les deux étapes successives de la vérification qui pourraient dans l'absolu être totalement séparées,
- d'un point de vue algorithmique, l'une est découpée en douze composants à traiter séparément, chaque composant nécessitant une analyse syntaxique d'un flot d'octets ; l'autre est constituée d'un traitement linéaire d'un ensemble de byte codes,
- du point de vue de la modélisation les deux vérifieurs diffèrent dans l'utilisation qui est fait de la méthode B.

Nous avons construit un modèle unique pour le vérifieur puisque dans notre architecture, le vérifieur de type s'appuie sur le vérifieur de structure aussi bien pour lui fournir des propriétés que des services dont il a besoin.

Le vérifieur de type est complètement modélisé en B sauf en ce qui concerne les allocations mémoire. Pour les accès aux composants, une interface contenant un modèle nécessaire au vérifieur de type a été modélisée. Cette interface est ensuite raffinée et complétée pour fournir non seulement les services au vérifieur de type mais aussi spécifier les tests du vérifieur de structure. Cette seconde partie n'est pas modélisée intégralement en B. Ceci est dû au fait que comme la vérification de structure comporte une vérification syntaxique d'un flot d'octets, il est difficile d'en fournir une représentation abstraite. Certains composants ont été complètement modélisé jusqu'à une interface qui permet de lire un octet dans un fichier. Nous avons ainsi montré la faisabilité d'une modélisation s'appuyant sur des briques de base de très bas niveau, mais d'autres composants n'ont pas été modélisés et ont été implémentés directement en C. Par contre, la partie concernant certains tests internes et la totalité des tests externes a été modélisée. Les tests internes font partie intégrante de la vérification de structure, les tests externes suivent le même schéma de raffinement que le vérifieur de type.

Le vérifieur de type : il a pour but de vérifier les règles de typage du langage Java Card. De plus, il assure qu'aucun débordement de la pile n'aura lieu à l'exécution et qu'il ne sera pas dépilé plus d'éléments que la pile n'en contient. Dans le cadre de notre développement, nous avons choisi d'intégrer la vérification de composant *Reference Location* à la vérification de type, ces deux étapes pouvant être menées parallèlement car elles nécessitent toutes deux une analyse du byte code.

La vérification est menée méthode par méthode. A l'intérieur d'une méthode, le byte code est vérifié de façon linéaire grâce à la technique du PCC. La complexité du vérifieur est donc linéaire en fonction de la taille du code ; même si la recherche d'informations permettant de vérifier le typage dans le fichier CAP peut être complexe.

Le vérifieur de type est modélisé intégralement en B. Il est composé d'un modèle abstrait raffiné par un modèle concret. Le modèle abstrait comprend les boucles de haut niveau et la spécification complète de chaque test à effectuer sur chaque byte code. Le modèle concret l'implémente en s'appuyant sur les services fournis par le vérifieur de structure et une machine de base modélisant la mémoire volatile de travail (RAM).

Le vérifieur de structure : il a pour but de vérifier la correction syntaxique du fichier chargé, d'assurer les propriétés et les interfaces nécessaires au vérifieur de type et d'assurer la validité des propriétés de bonne formation du fichier, notamment celles concernant les relations entre composants. Nous nommons tests internes, la vérification syntaxique et la vérification de propriétés internes à un composant et tests externes, les vérifications de propriétés entre composants.

Le vérifieur de structure ayant pour premier objectif, d'assurer l'analyse syntaxique des composants, nous avons mis en place une architecture où chaque composant est modélisé par une machine. Cette machine décrit les propriétés et les services d'accès à ce composant ainsi que les tests internes de ce composant.

La modélisation des douze composants du fichier CAP suit ainsi le même modèle. Il contient un ensemble de variables abstraites modélisant les informations contenues dans le composant, un invariant qui définit les propriétés des variables, une opération spécifique qui réalise l'analyse syntaxique et effectue les tests internes et des opérations de lecture permettant d'accéder aux informations dans le cas où l'analyse syntaxique aurait réussi. Pour ce faire, la machine contient une variable booléenne indiquant si les tests internes ont réussi, cette variable est initialisée à faux, est mise à jour par l'opération de test interne et est en pré condition des opérations de lecture.

L'interface du composant ne fournit que les propriétés et les variables nécessaires à la modélisation du vérifieur de type et des tests externes. Lors du raffinement de cette machine, la spécification est complétée pour permettre de décrire complètement le composant et les tests liés aux informations qu'il contient. Le raffinement se fait par collages successifs pour terminer sur une représentation élémentaire du fichier, c'est-à-dire une fonction qui associe à une adresse dans la mémoire, l'octet présent à cette adresse. Ici on ne peut pas parler de modèle concret, ni de modèle abstrait puisque la chaîne de raffinement avec les collages successifs vers une représentation concrète constitue la spécification du composant.

Modélisation : Nous n'entrerons pas dans le détail de la modélisation ni du vérifieur de structure ni du vérifieur de type. Nous renvoyons le lecteur intéressé à la lecture de la thèse de Ludovic Casset. Nous nous intéresserons juste à la partie formalisation. En effet, la plupart des erreurs rencontrées au cours de ce développement sont venue d'une mauvaise interprétation de la spécification informelle.

La méthodologie utilisée est traditionnelle pour les utilisateurs de la méthode B, elle consiste à réécrire la spécification informelle dans un format proche de B. Ainsi pour l'instruction *aaload*, le sommet de pile doit contenir une référence sur un tableau (*arrayref*) et un index dans cette même table (*index*). Après l'exécution de l'instruction *aaload*, les deux éléments précédents sont enlevés de la pile et remplacés par la valeur du tableau à l'index donné. L'inconvénient avec toute spécification informelle c'est que les informations essentielles sont souvent éclatées dans le document. Par exemple pour l'instruction *aaload*, le contenu de la pile n'explicite pas le typage requis ou d'une manière incomplète. La figure suivante donne *in extenso* la définition informelle de Sun.

Aaload
Load reference from array
Stack
…, *arrayref*, *index* ⇒
…, value
Description
The *arrayref* must be of type reference and must refer to an array

whose components are of type reference. The index must be of type short. Both *arrayref* and index are popped from the operand stack. The reference value in the component of the array at index is retrieved and pushed onto the top of the operand stack.
Runtime If *arrayref* is null, *aaload* throws a *NullPointerException*.
Exceptions Otherwise, if index is not within the bounds of the array referenced by *arrayref*, the *aaload* instruction throws an *ArrayIndexOutOfBoundsException*

Figure 18 Spécification informelle de l'instruction ***aaload***

La réécriture définit quels types sont attendus en entrée par l'instruction et comment cette même instruction modifie les types dans la pile. La figure suivante indique dans la première partie la sémantique opérationnelle de l'instruction. Il doit y avoir au moins deux éléments sur la pile pour que l'instruction puisse s'exécuter. Si la pile contient une référence sur un tableau d'objets et un *short*, alors l'instruction *aaload* enlève ces deux éléments de la pile et retourne le contenu du tableau à l'index indiqué par le short. Dans le second cas, la pile contient une référence *null* et un *short*. Dans ce dernier cas, l'instruction *aaload* retourne une référence *null*. Si la pile contient d'autres séquences de types que les deux précédentes, l'instruction ne peut pas s'exécuter et une erreur est détectée. La seconde partie de la spécification concerne la sémantique statique c'est-à-dire les tests qui doivent être vérifiés pour permettre l'exécution de l'instruction. Ces tests vérifient qu'il y a bien au moins deux éléments dans la pile et que ces deux éléments sont du type attendu, en accord avec la sémantique statique de l'instruction *aaload*.

La troisième partie décrit les modifications effectuées par l'instruction elle-même. Ces modifications concernent l'évolution de la pile et/ou des variables locales. Dans l'exemple suivant, les deux éléments se trouvant au sommet de la pile sont supprimés de la pile et un nouvel élément est ajouté au sommet de la pile. Le type de ce dernier élément est fonction du type du second élément dans la pile, avant l'exécution de l'instruction. Notons que dans ce cas précis, les variables locales ne sont pas modifiées. Cependant, d'autres instructions du langage les modifient. La quatrième partie de notre format de spécification contient les tests qui permettent d'assurer les propriétés en sortie de l'instruction. Dans notre exemple, il n'y a pas de tels tests de post-modification. Enfin, la cinquième et dernière partie indique les catégories d'exceptions qui peuvent être lancées par la dite instruction.

aaload
[…, refarray class, short] => […, ref class]
[…, null, short] => […, null]
Pre-modification tests:
1. The stack must contain at least two elements ***(1)***
2. The two topmost elements of the stack have to be of types compatibles with refarray class and short. ***(2)***

Modifications: ***(3)***
The two topmost elements of the stack are removed. If the second element was a refarray type, then a reference of the same class is pushed onto the stack. Otherwise a type null is pushed.
Post-modification tests:
None
Throws ***(4)***
- NullPointerException
- ArrayOutOfBoundException

- SecurityException

*Figure 19 Réécriture de l'instruction **aaload***

Chaque instruction du langage Java Card est réécrite suivant ce format. Ce document devient le document de référence pour la modélisation. Ainsi, à partir de la description informelle, nous obtenons la spécification formelle du vérifieur de type.

Il est désormais envisageable de réaliser la formalisation de cette spécification en plusieurs étapes de raffinement jusqu'à l'obtention du code exécutable. Cette méthodologie de raffinement est celle utilisée pour le développement B. La traduction de l'informel vers le formel est une étape cruciale car l'obtention du code et la construction des preuves se fait à partir de cette spécification formelle. Ainsi, si une erreur se glisse dans la traduction, la phase de preuve peut être dans l'incapacité de la détecter. Nous pensons en particulier aux définitions des types attendus par les instructions. Ces définitions sont des postulats de base dans la modélisation. Une erreur dans leur écriture et au final, le vérifieur acceptera ou rejettera des programmes respectivement faux ou corrects, ce qui n'est pas le comportement attendu du vérifieur.

La figure suivante montre la modélisation en B de l'instruction *aaload* basée sur le modèle de FACADE. Le cheminement du modèle abstrait suit notre réécriture de la spécification ainsi, ***(1)*** indique qu'il doit y avoir au moins deux éléments sur la pile. ***(2)*** montre que l'élément au sommet de la pile doit être un *short* et que le deuxième élément à partir du sommet de la pile doit être soit une référence sur un tableau, soit un *null*. Nous devons différencier ces deux derniers cas, car des traitements différents sont attendus pour chacun des cas. Parce que l'instruction *aaload* a deux traitements différents, la clause B **SELECT … THEN … WHEN … THEN … ELSE … END** est la plus adaptée à cette situation. Cette clause permet de spécifier une action gardée par une condition. Dans notre cas, tous les **SELECT** sont déterministes et la spécification globale est déterministe. L'opération retourne un résultat : une valeur est donnée à la variable contenant le résultat dans chaque cas. De plus, la pile n'est modifiée que dans les deux cas corrects acceptés par l'instruction : deux éléments sont supprimés de la pile et un nouveau est ajouté.

Dans le **SELECT** et dans le **WHEN**, le développeur indique les pré-conditions qui permettent l'exécution de l'instruction. Dans notre exemple, les pré-conditions indiquent qu'il doit y avoir au moins deux éléments sur la pile, que l'élément au sommet de la pile est un *short* et que le second élément à partir du sommet de la pile est soit une référence sur un tableau d'objets, soit *null*. Dans les clauses **SELECT** et **WHEN**, nous ajoutons les tests pour le traitement des exceptions ***(4)***. En fait, il s'agit d'une condition additionnelle qui vise à vérifier que si l'instruction est dans un handler d'exception, nous devons alors vérifier que pour tous les labels de preuve donnés pour chaque handler, les types des variables locales sont compatibles avec les types contenus dans les descripteurs des variables locales associés aux labels et qu'il n'y a pas de références non initialisées dans les variables locales. Cette condition additionnelle vient du fait que certaines instructions peuvent lancer des exceptions. Il faut donc tenir compte de ce comportement exceptionnel lors de leur modélisation puis de leur vérification. Ces conditions apparaissent en fait dans la clause THROWS de la spécification informelle. Dans cette clause, les exceptions que peut lancer l'instruction considérée sont énumérées.

Enfin, dans les clauses **THEN** et **ELSE**, nous donnons le comportement de l'instruction. Dans les deux cas corrects, la variable retournée par l'opération est mise à vrai (**TRUE**), indiquant que la vérification s'est bien passée. De plus, la pile est modifiée suivant la branche dans laquelle nous nous trouvons. Ainsi, dans les deux cas, deux éléments sont supprimés de la pile. Ensuite, dans un cas, une référence est ajoutée sur la pile et dans l'autre cas un *null* est ajouté sur la pile. Le troisième et dernier cas correspond au cas d'erreur : il n'y a pas assez d'éléments sur la pile, le typage des

éléments est incorrect ou alors la vérification du handler d'exception n'est pas correcte. L'instruction *aaload* retourne alors faux (**FALSE**) et ne modifie ni la pile, ni les variables locales. L'information qu'une erreur a été découverte est propagée et le processus de vérification est arrêté. Dans les cas où tout s'est bien passé, le processus de vérification continue par la vérification de l'instruction suivante

```
bb ← verify_aaload =
BEGIN
   SELECT
        2 ≤ size(stack) ∧                                                    (1)
        last(stack) = c_short ∧                                              (2)
        last(front(stack)) = c_refarray ∧
        (pc ∈ dom(exception_handler)                                         (4)
        ⇒
        ∀label.(label ∈ exception_handler(pc)
              ⇒ COMPATIBLE(loc_var, loc_var_descriptor(label))) ∧
        c_uref ∉ ran(loc_var))
   THEN
        bb    := TRUE ||
        stack := front(front(stack)) ← c_ref                                 (3)
   WHEN
        2 ≤ size(stack) ∧                                                    (1)
        last(stack) = c_short ∧                                              (2)
        last(front(stack)) = c_null ∧
        (pc ∈ dom(exception_handler)                                         (4)
        ⇒
        ∀label.(label ∈ exception_handler(pc)
              ⇒ COMPATIBLE(loc_var, loc_var_descriptor(label))) ∧
        c_uref ∉ ran(loc_var))
   THEN
        bb    := TRUE ||
        stack := front(front(stack)) ←c_nul                                  (3)
   ELSE
        bb := FALSE
   END
END
```

*Figure20Modélisation abstraite de l'instruction **aaload***

Le modèle formel du vérifieur de byte code embarqué est destiné à produire une implémentation formelle en C. Dans notre méthodologie de développement, il nous a fallu prendre en compte non seulement le développement formel, et en particulier le passage des besoins informels aux spécifications formelles mais également sa traduction et son intégration sur une plate-forme à fortes contraintes.

Ainsi si le développement formel à l'aide de la méthode B permet d'obtenir une implémentation formelle en B0, celle-ci doit encore être traduite en C puis compilé pour le processeur cible de la carte à puce. Les processus de traduction et de compilation ne sont pas des processus qui ont été formalisés. Si la compilation a été qualifiée, ce n'est pas le cas de la traduction pour laquelle nous avons développé notre propre traducteur. Pour ces raisons, nous ne pouvons entièrement nous reposer sur le développement formel pour assurer la correction du vérifieur. Il faut utiliser d'autres techniques. La figure suivante décrit les différentes phases du développement, de l'expression des besoins fonctionnels du vérifieur au code embarqué de ce même vérifieur.

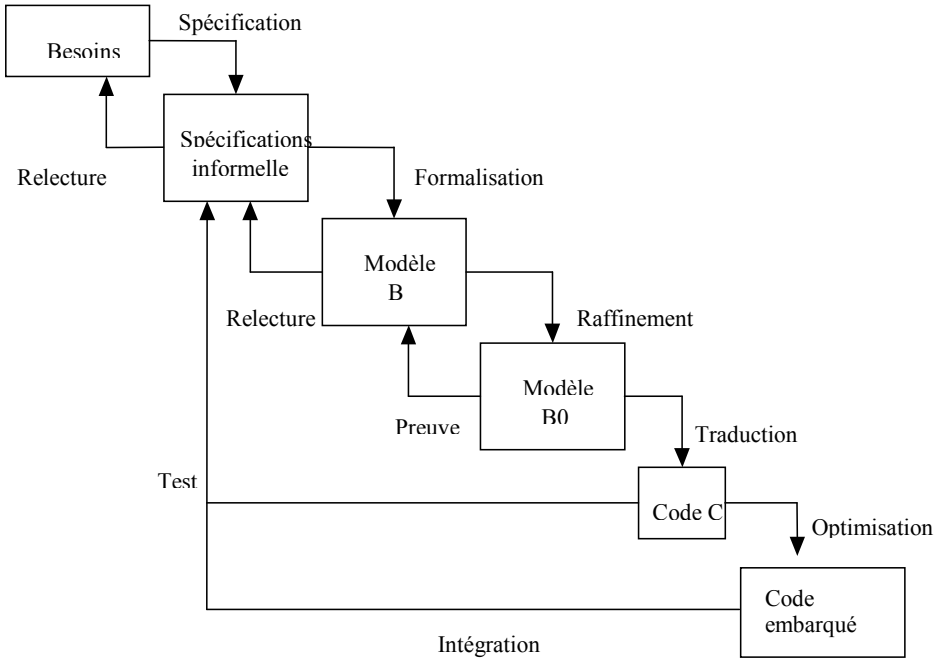

Figure 21 Cycle de développement complet

Le développement formel à lui tout seul n'assure pas une totale correspondance entre les besoins informels et le code produit. Cela est dû principalement aux activités en amont et en aval du processus de développement formel qui restent des activités manuelles. Il faut donc, en particulier dans une activité industrielle, tenir compte de cet aspect et compléter le développement formel par d'autres techniques comme le tests et la relecture de code.

La relecture des spécifications formelles écrites en B permet de s'assurer manuellement que la traduction des besoins informels en spécifications formelles s'est faite correctement. C'est une étape importante car par la suite, le développement formel s'appuie sur cette traduction. Une erreur à ce niveau peut donc avoir des répercussions sur tout le développement.

Pour s'assurer que la traduction de l'implémentation formelle vers du code C puis vers du code compilé pour le processeur cible est correcte, nous ajoutons une phase de test. En fait, les tests peuvent être effectués à deux moments différents. Le premier se situe après la génération du code C issu de l'implémentation formelle. Le second sur le code installé dans la carte à puce. Cela permet de tester différentes parties du processus de développement comme la phase traduction et la phase d'intégration. Notons que les cas de tests ne sont pas générés à partir des spécifications formelles mais de l'expression des besoins fonctionnels. Cela nous permet en plus, de nous assurer de la conformité du code généré par rapport aux spécifications informelles.

L'intégration des parties de code développées formellement avec des parties de codes classiques est un point très important car il est illusoire d'imager développer formellement l'ensemble d'un système. Il faut d'abord modéliser les interactions entre ces deux développements. L'avantage

d'utiliser B est la facilité avec laquelle nous pouvons utiliser les machines de base pour réaliser cette interface entre le développement formel et informel. Ces machines contiennent la spécification abstraite des blocs de base, c'est-à-dire la liste des variables, de leurs propriétés et de leurs services. Cette spécification abstraite est alors utilisée par les différents modèles du vérifieur de type et de structure. Cependant, ces machines de bases ne sont pas raffinées suivant la méthodologie B. En fait, une implémentation directe est fournie (en C ou en un autre langage). Il n'y a alors pas de preuve que les implémentations de ces machines de base correspondent effectivement à leurs spécifications. Cependant, ces machines de bases représentent des blocs logiciels très précis dont les implémentations ont déjà fait l'objet de la plus grande attention comme la gestion de la mémoire ou de la communication. Ces blocs ne sont de toutes les façons pas développables à l'aide de la méthode B. Nous assurons la correspondance par un processus classique la relecture de code.

L'un des principaux avantages de l'utilisation de la méthode B est la génération automatique de code qui conclut la phase d'implémentation. En fait, à la fin du processus de raffinement, la machine B obtenue est une implémentation écrite dans un sous-ensemble du langage B, le B0. Pour aborder cette génération de code, nous avons conçu un traducteur de code rudimentaire qui ne prend en compte que les implémentations en B0 du modèle formel. Ce qui n'est pas le cas de traducteurs comme celui développé par ClearSy. Pour une traduction plus simple, nous ajoutons les types des variables sous la forme d'assertions dans le code B0. Cela permet au traducteur de choisir le meilleur type C pour la variable à traduire. Cette simple considération nous permet en particulier de restreindre l'espace mémoire d'une variable à son strict nécessaire. Ce travail a été décrit en détail dans [Bos-00].

Conclusions : Le développement B du vérifieur de byte code Java Card est une réussite du point de vue de la modélisation : nous avons atteint l'objectif de disposer d'un modèle et de son implémentation correspondante. En utilisant une méthodologie de développement formel nous avons fourni une implémentation adaptée pour carte à puce, issue d'un modèle formel. La modélisation doit être utilisée avec précaution car les outils et les méthodes demandent encore des améliorations. Les parties du système où l'utilisation des méthodes formelles est nécessaire doivent être clairement identifiées. Ainsi, les interfaces entre les parties prouvées et non prouvées sont cruciales pour le système lui-même et doivent être définies avec le plus grand soin. Cette interaction entre les parties prouvées et non prouvées permet de réduire les coûts de développement, de réutiliser du code déjà développé et de réduire ainsi les risques tout en augmentant la confiance que nous pouvons mettre dans du code de plus en plus complexe.

Nous ne reprendrons pas ici ces travaux qui ont fait l'objet de plusieurs publications [Cas-02d] et [Bur-02a] mais nous nous intéressons à un point peu abordé dans les travaux académiques, la valorisation des travaux et le transfert technologique.

2.5 Plan d'évaluation et collecte de métriques

Les techniques d'évitement de faute par construction correcte ont un grand impact dans la communauté scientifique mais reçoivent assez peu d'attention de la part du monde industriel. Le seul cas où les industriels adoptent ces technologies, c'est sous la contrainte d'une certification, et les techniques formelles sont plus vues comme un coût supplémentaire plutôt qu'un moyen d'améliorer la qualité du logiciel. Il faut donc convaincre que ces technologies sont à même de passer l'échelle mais aussi sont compatibles avec les contraintes économiques rencontrées par les industriels. Nous avons donc évalué ce processus de développement en soumettant plusieurs hypothèses que nous cherchons à invalider. Pour ce faire, il nous faut disposer d'une base

d'informations quantitatives sur le développement et pouvoir le comparer avec un développement traditionnel.

Plan d'évaluation : Ces hypothèses mesurent l'apport de la modélisation si elles ne sont pas invalidées. Nous définissons trois hypothèses pour évaluer l'impact et le résultat d'un développement formel sur un cas industriel :

- Hypothèse 1 : l'utilisation des méthodes formelles, et dans notre cas précis, l'utilisation de la méthode B, améliore la qualité du logiciel résultant de la modélisation.
- Hypothèse 2 : le surcoût d'un développement formel est acceptable.
- Hypothèse 3 : Le code généré par l'application de la méthode B n'entraîne pas une augmentation significative de l'occupation mémoire ou du temps d'exécution.

Les deux premières hypothèses nécessitent de pouvoir comparer un développement formel et un développement conventionnel. Nous devons mener ces deux développements par des équipes séparées et collecter les informations concernant les coûts et les erreurs trouvées dans les différentes étapes du développement.

Trois étapes ont été identifiées lors du développement :

- La translation de la spécification informelle vers un modèle formel abstrait, la vérification est faite par une revue de code,
- Le développement formel dont la validation est faite par raffinement et preuves,
- La translation de la spécification formelle détaillée vers un code exécutable, la vérification est faite par de la relecture et du test.

Durant la première phase les erreurs sont découvertes par la une revue effectuée par une personne externe au groupe de développeur. Le processus de revue est le premier moyen de s'assurer de correction de la formalisation, l'autre est le test. L'avantage de la revue sur le test est d'arriver plus tôt dans le cycle de développement et donc évite de perdre du temps dans une activité de preuve interactive. Durant cette phase 12 erreurs ont été relevées. La seconde phase est liée au processus de développement B. A chaque raffinement, le générateur d'obligation de preuve propose les lemmes liés au raffinement. Dans la méthodologie utilisée, nous ne réalisons que la preuve automatique suivie d'une revue des lemmes restants. Si ceux ci indiquent une erreur de raffinement alors cette erreur est corrigée. Mais nous ne réalisons pas de preuve interactive. Lorsque le dernier raffinement est atteint et que toutes les structures de données sont implémentables alors nous reprenons l'ensemble de lemmes avec le prouveur interactif. Certains nous avaient semblés justes alors qu'ils étaient erronés. Le processus de preuve peut alors être vu comme un outil de mise au point extrêmement puissant qui nous a permis de découvrir 29 lemmes faux. La dernière étape est la validation par le test. Cette étape nous assure dans une certaine limite que le passage du langage formel de bas niveau au langage exécutable est correct ainsi que le passage de la spécification informelle vers le modèle abstrait de notre système. Durant cette phase, 32 erreurs ont été découvertes liées à la formalisation (14) et au traducteur de code (9) et à des parties non modélisée (9).

L'ensemble des métriques concernant les coûts de développement, les origines des erreurs, la charge de travail pour le développement formel et conventionnel est détaillé dans [Cas-02c]. Nous fournissons et commentons ci-dessous certaines de ces métriques.

Le premier tableau synthétise les métriques concernant le développement. En particulier, il apparaît que le vérifieur de structure est plus gros que le vérifieur de type. Ceci est dû au fait que le vérifieur de structure contient de nombreux tests, très différents. Chacun de ces tests nécessite une spécification et une implémentation, ce qui a pour conséquence d'augmenter la taille des modèles et

du code C généré par la suite. Le vérifieur de type, quant à lui, peut être vu comme une seule entité incluant les règles de typage définies par le langage Java Card. Ainsi, il est possible de factoriser le code et d'éviter une expansion des modèles B et du code C généré. De plus, le vérifieur de structure contient des services d'accès aux données du fichier CAP qui sont utilisés par le vérifieur de type pour accéder notamment aux instructions des méthodes, aux définitions des classes et au type des champs. Cela explique également les différences apparaissant sur les nombres de composants B : le vérifieur de structure contient beaucoup plus de composants que le vérifieur de type.

	Structure	Typage	Utilitaires	Total
Nombre de lignes de B	35000	20000	3500	58500
Nombre de composants B	116	34	45	195
Nombre de POs générées	11700	18160	950	30810
Pourcentage de POs automatiquement prouvées	81 %	70 %	77 %	74 %
Nombre de machines de base utilisées	6	0	7	13
Nombre de lignes de C	7540	4250	860	12650

Figure 22 *Métriques sur le développement formel : complexité*

Ce tableau présente deux résultats remarquables : le premier concerne le nombre de lemmes ou obligations de preuve (POs). Les résultats montrent que le vérifieur de type génère bien plus de POs que le vérifieur de structure. La principale raison est le traitement par cas effectué par le vérifieur de type produisant ainsi un nombre considérable de lemmes. Le deuxième résultat concerne le nombre de lignes de C. Ce nombre est bien en dessous du nombre de lignes de B des modèles du vérifieur. Ce résultat vient du fait que la traduction du B vers du C ne prend en compte que les implémentations. Ainsi, les machines abstraites et les raffinements ne sont pas utilisés lors de la traduction. De plus, les clauses INVARIANT des boucles dans les implémentations ne sont pas traduites. Cela réduit drastiquement le nombre de lignes de B qui seront traduites en C.

La tableau suivant représente les résultats de l'exécution du vérifieur de byte code sur quelques applets. Ce sont des applets concrètes qui correspondent à des applications industrielles réelles déjà déployées sur les cartes à puce. La colonne *taille* contient la taille en octets des applets qui sont envoyées au vérifieur. Les colonnes *structure* et *type* contiennent le temps nécessaire pour exécuter chacune des phases de la vérification.

Le premier commentaire qui peut être extrait de ces résultats est que l'exécution du vérifieur de type est plus coûteuse en temps que celle du vérifieur de structure. En fait, le vérifieur de structure est une succession de tests. Certains sont simples, d'autres sont plus complexes, mais ce sont des vérifications de cohérence à effectuer en général une seule fois. Au contraire, le vérifieur de type effectue des tests pour chaque instruction. Afin de privilégier la faible consommation de mémoire volatile (RAM), il a été choisi de ne pas recourir au stockage temporaire dans cette mémoire de données dont le vérifieur de type peut souvent avoir besoin. En effet, pour réduire les besoins en mémoire, tant en RAM qu'en EEPROM, le code doit être parcouru à chaque fois qu'une information est demandée.

Applet	Taille (octets)	Vérifieur de Structure	Vérifieur de Type
Utils	4439	230 ms	1422 ms
pacapint	2375	50 ms	110 ms
Wallet	2762	100 ms	460 ms
TicTacToe	4988	130 ms	1372 ms
Applet 1	10394	611 ms	26 856 ms
Applet 2	22554	1161 ms	11 506 ms

Figure 23 *Métriques sur le développement formel : performance du logiciel*

La contrepartie évidente d'une telle approche est l'augmentation du temps d'exécution. Même si l'algorithme de vérification est linéaire, le temps nécessaire à la vérification des règles de typage est supérieur au temps de vérification de la cohérence des données contenues dans le fichier CAP. Si l'applet 1 nécessite près de 28 secondes pour être vérifiée, c'est parce que cette applet contient de nombreuses exceptions et que la vérification des exceptions est coûteuse en parcours de code. Une façon d'optimiser est d'essayer de faciliter l'accès aux données en stockant des données temporaires en mémoire volatile (RAM). Un compromis doit être trouvé entre l'utilisation de RAM et d'EEPROM et le temps d'exécution.

	Développement formel	Développement conventionnel
Coût du développement	12 semaines	12 semaines
Coût de la preuve	6 semaines	NA
Coût du test	1 semaines	3 semaines
Integration	1 semaines	2 semaines
Total	**20** semaines	**17** semaines
Nombre de fautes détectées par la relecture	13	24
Nombre de fautes détectées par la preuve	29	-
Nombre de fautes détectées par le test	32	71
Total	**74**	**95**

Figure 24 *Métriques sur le développement formel : coût et qualité*

Ces métriques indiquent que la première hypothèse est vraie. Pour la seconde, il existe une surcharge liée à la méthodologie cependant nous ne prenons pas en compte le cycle de vie complet du produit. En effet les retours clients ne sont pas pris en compte ici. Probablement, il existe plus d'erreurs résiduelles dans le développement conventionnel que dans le développement formel. Les métriques (non-fournies ici) concernant la taille et les performances montrent que le code formel est sensiblement plus gros, mais les performances sont équivalentes.

Résultats : Le résultat majeur de ce travail d'évaluation est la démonstration de la non-invalidation des hypothèses [Lan-02a] et donc une grande confiance sur l'amélioration de la qualité du logiciel à un coût raisonnable. Nous avons aussi montré l'intérêt pour un industriel d'inclure dans son processus de développement de logiciels des méthodes formelles. En particulier, nous avons mis l'accent sur la conformité du code produit avec la spécification initiale.

Mais la comparaison relève aussi des manques. Le premier, et le plus marquant d'entre eux, concerne l'activité de preuve. Cette activité est clairement identifiée comme la partie coûteuse du développement formel. Elle révèle une tendance à risque liée à des outils et une méthodologie

encore en cours d'évolution et d'amélioration. Cette méthodologie met en avant le comportement que doit avoir une équipe de développement face à la preuve, notamment quant à son organisation et à sa gestion :

- le découpage de la phase de preuve qui passe d'abord par une étape de correction du modèle et d'analyse des lemmes générés avant de se lancer dans la preuve des lemmes les uns après les autres,
- le développement de règles de preuve, un investissement et un coût en temps important sur les premiers développements, et donc plus encore sur le premier, mais sur lequel nous pouvons capitaliser.

Cette étude est encourageante quant à l'intérêt des méthodes formelles dans le domaine des cartes à puce. Pour parvenir aux résultats, il n'a pas été besoin d'optimiser totalement le prototype. Ainsi, pour améliorer le vérifieur, quelques optimisations ont déjà été identifiées : l'amélioration du modèle mémoire, l'élimination des tests redondant dans le vérifieur structurel, l'amélioration du traducteur de code pour prendre en compte des optimisations spécifiques. Enfin, la comparaison et l'analyse de la phase de preuve soulignent la nécessité d'améliorer les outils. Pour accompagner la méthodologie, les outils doivent subir des évolutions. Des améliorations sur la gestion des lemmes, de leurs hypothèses et de leurs preuves peuvent faciliter la phase de développement et de démonstration des lemmes, et avoir ainsi un impact significatif sur le développement en réduisant sa durée.

2.6 Conclusions

Ces travaux sur les techniques d'évitement de faute par construction correcte de logiciel nous ont permis d'aborder un problème industriel en partant d'une preuve de concept : la décomposition d'une machine virtuelle défensive en deux systèmes indépendants : un vérifieur et un interpréteur offensif en démontrant la correction du raffinement. Nous avons poursuivi par la correction de l'algorithme de vérification lui-même. Puis nous avons raffiné un tel vérifieur pour obtenir un code exécutable sur une carte à puce en démontrant que ce code est conforme à la spécification initiale. Au début de ce travail de recherche, nous avons proposé deux défis : la réalisation d'un vérifieur de byte code dans une carte à puce et la preuve de la correction de ce dernier.

Ajouter un vérifieur de byte code dans une Java Card permet à la carte d'assurer elle-même sa propre sécurité. Lorsque l'architecture de déploiement des cartes est discutée, l'indépendance de la carte vis-à-vis de sa propre sécurité permet entre autre de réduire les coûts d'architecture ainsi que le temps de déploiement. En effet, aujourd'hui, déployer de nouvelles applications sur une carte déjà dans les mains de l'utilisateur final requiert une importante infrastructure qui implique des centres de certification et des protocoles de cryptographie lourds et coûteux. Avec un vérifieur embarqué, l'infrastructure est plus réduite : il suffit d'envoyer l'application à la carte qui se charge elle-même de la vérifier et assure ainsi sa propre sécurité.

Maintenant qu'une implémentation du vérifieur de byte code est disponible, plusieurs ouvertures sont possibles : l'implémentation peut être améliorée et optimisée afin de l'intégrer dans une machine virtuelle. Cette intégration n'est pas triviale et doit prendre en compte plusieurs paramètres comme le chargement et l'édition de liens effectués par la carte, ainsi que le stockage des informations de typage nécessaires à la vérification par la carte. Une autre possibilité, est d'ajouter des capacités cryptographiques au prototype du vérifieur existant. Cela fournirait un vérifieur résistant aux attaques classiques, du fait de son intégration dans une carte à puce, qui pourrait être utilisé pour signer des applets.

Le processus de preuve appliqué durant le développement formel du vérifieur de byte code assure sa correction.. Nous avons acquis l'expérience pour intégrer du développement formel dans un développement traditionnel déjà existant, ainsi que l'évaluation du coût d'un développement formel et de l'intégration. L'utilisation des méthodes formelles n'est pas sans impact. Un compromis doit être trouvé pour obtenir la meilleure confiance dans le code pour le meilleur coût. Par exemple, nous montrons que la formalisation de chaque composant du fichier CAP n'est pas nécessaire. L'apport de la méthode B est marginale pour ce type de développement. Il faut se concentrer sur d'autres points comme le vérifieur de type où les apports des méthodes formelles sont plus importants.

Nous avons mis l'accent sur l'aspect méthodologique afin de fournir des données quantitatives sur le processus de développement formel afin d'aider des responsables de projets de choisir cette technologie. Nous avons aussi une méthodologie de développement adapté au domaine de la carte à puce afin de rendre un tel développement acceptable en terme de coût sans obérer la viabilité de la chaîne de confiance. Il reste bien entendu à valider la correction de notre traducteur de code mais qui n'est plus un travail de recherche mais d'ingénierie du logiciel.

Un point sur lequel il reste à travailler est l'intégration du processus de test dans un cadre où le code n'est pas produit pas une spécification formelle ce qui représente un autre volet de nos travaux.

3. Elimination de fautes dans les systèmes enfouis

3.1 Détection de flux illicites

La sécurité des systèmes de confiance repose souvent sur des cartes à puce. Ces dernières sont à même d'effectuer des traitements dans un environnement sécurisé comme le calcul de clés de session ou le stockage d'informations sensibles. Tant que le système de confiance est dans un environnement clos, la sécurité est bien maîtrisée. Cependant la demande aujourd'hui requiert une ouverture de ces systèmes à d'autres acteurs. Dès lors, les systèmes de confiance doivent intégrer dans leur environnement des applications pour lesquelles le niveau de confiance peut être plus faible. Les cartes multi-applicatives sont l'exemple même où le chargement de code peut être réalisé en dehors du périmètre de confiance, c'est-à-dire dans un environnement potentiellement hostile. Les cartes à puce basées sur la technologie Java autorisent un tel chargement de code. Différents mécanismes ont été définis afin d'assurer la ségrégation du code à l'intérieur de la carte (vérifieur de byte code, firewall, etc.) voir lors du chargement en s'assurant de la provenance de l'application.

Les fournisseurs d'application doivent respecter de nombreuses exigences de sécurité lors de leurs développements comme la confidentialité des clés secrètes, l'intégrité des données sensibles, etc. Ces exigences sont couvertes par de multiples mécanismes existants dans le silicium ou définis par la norme Java Card ou dans les spécifications de *Global Platform* comme le chargement sécurisé. Cependant la possibilité de partager des informations entre deux applications à l'intérieur de la carte entraîne de nouvelles exigences de sécurité comme le déni de service et le flot illicite d'information. Il est important en effet que des informations confidentielles puissent être partagées entre deux applications, mais le propriétaire de l'information doit pouvoir s'assurer que cette information ne sort pas du cercle de confiance qu'il a définit. La relation de confiance ne doit pas être transitive sinon de nouvelles attaques logiques deviennent possibles [Gir-99]. Pour l'utilisateur final ceci se traduit par une perte d'image de marque (le déni de service pour une banque) ou d'atteinte à la vie privée (transfert d'informations médicales ou bancaires à un tiers).

D'une manière plus générale, dans le cadre du chargement de code mobile on peut distinguer deux problèmes. Le premier concerne la protection du code mobile et de ses données contre un support d'exécution malveillant et le second la protection du système contre du code malveillant. Ce dernier peut être généralisé par la suspicion mutuelle de programme. Dans le cadre de la carte ouverte, il faut préserver l'intégrité et la confidentialité des données du système et des autres applications par rapport à un code non digne de confiance. Il faut s'assurer que le code mobile ne va pas corrompre les données ni laisser fuir vers le terminal ou l'application distante des données privées du système. La fuite d'information peut être résolue par des techniques d'analyses de flot d'information, alors que l'intégrité des données ne peut être assurée que par la preuve de la correction du code par rapport à cette propriété.

Les architectures de sécurité comme celle de Java utilise des concepts bien connus comme les domaines de protection, les contrôles d'accès, les permissions, le code signé… Le principe est basé sur la surveillance de l'exécution de l'interprétation du code qui permet de le stopper si une opération illégale est survenue. Cependant l'interpréteur doit gérer des informations spécifiques lors

de l'exécution. Parmi les techniques utilisées citons l'introspection de pile, pour laquelle il est vérifié dynamiquement les entités identifiées dans la pile d'appel afin de s'assurer de la compatibilité de leur domaine de sécurité par rapport au domaine de sécurité courant.

Il est évident que les solutions basées sur des mécanismes d'authentification est insuffisant car dès lors la confiance n'est basée que sur l'origine du code et non pas sa sémantique. De plus ceci nécessite le déploiement d'infrastructure de gestion et de révocation de certificats dont la mise en œuvre est très lourde.

Les techniques d'analyse statique sont donc celles qui s'imposent dans un tel contexte. C'est pour cette raison que la machine virtuelle Java dispose d'un vérifieur de type embarqué. Ce vérifieur s'assure que le code mobile peut être interprété d'une manière offensive tout en garantissant que la sémantique du langage est préservée. Améliorer cette approche par des analyses de types plus riches en assurant la prévention de flux illicites est certainement une direction prometteuse. Cependant lors de ce travail les capacités de calcul et de mémoire des cartes à puce que nous visions étaient trop faible pour utiliser une telle approche. Notre approche est basée sur une analyse non utilisable dans une carte mais sur un serveur permettant ainsi de garantir à un fournisseur de service que l'application qu'il s'apprête à offrir ne peut faire fuir les informations des autres applications. Nous proposons d'assurer la sécurité en contrôlant les dépendances entre les variables.

Dans un premier temps nous positionnons notre approche non seulement par rapport aux travaux connus à l'époque mais aussi par rapport à de plus récents développements. Ensuite nous définissons la propriété à vérifier pour que les dépendances soient contrôlées et nous donnons plusieurs conditions suffisantes de sécurité. Nous définissons ensuite une politique de sécurité multi-niveaux et nous montrons comment l'appliquer concrètement en utilisant une combinaison d'abstraction et de model checking pour vérifier son implémentation correcte à un ensemble d'application.

3.1.1. Positionnement

L'utilisation de techniques d'analyse statique pour raisonner sur le comportement et les propriétés de programmes ont été largement appliquées aux problèmes de sécurité. Les premiers travaux de Denning [Den-76] ont porté sur un mécanisme utilisé lors de la compilation pour contrôler qu'un programme ne contient pas de flots illicites. McLean [McL-92] décrit un cadre permettant de démontrer la non-interférence d'un module. Cette propriété de sécurité est démontrée à partir de la sémantique des traces. Parmi les travaux les plus récents, nous pouvons citer l'algorithme développé par Banâtre, Bryce et Le Métayer [Ban-94] pour l'analyse de programmes séquentiels. L'algorithme est dérivé de règles d'inférence d'une logique et démontré correct et complet par rapport à ce système d'inférence. Smith et Volpano [Smi-97] utilisent des systèmes de type pour garantir la non interférence pour un langage procédural. Ce travail peut être vu comme une re-formulation des travaux de Denning mais ils apportent en plus la démonstration de la correction de leur système. Dans [Smi-98], les auteurs étendent leurs travaux avec un système de type en prenant en compte le multi-threading.

Les travaux les plus proches des nôtres sont ceux de Myers et Liskov [Mye-97] qui proposent une technique d'analyse des flots d'informations pour les programmes impératifs en associant des niveaux de sécurité aux objets du programme. Avec ce niveau est associé pour chaque créateur possible un ensemble d'observateurs autorisés. Un flot d'information est autorisé de O1 à O2 si pour tous les créateurs de O1 la liste des observateurs de O2 est incluse dans O1. Les auteurs utilisent un solveur de contraintes pour vérifier que le niveau associé à chaque valeur est dominé par

le niveau associé de chaque variable qu'il affecte. Cette approche est plus efficace que l'utilisation d'un vérifieur de modèle mais ne permet de vérifier qu'un seul type de propriété de sécurité.

Récemment, Melo Da Souza [Mel-03] propose d'utiliser le mécanisme d'analyse statique du typage de Java pour vérifier une propriété de non-interférence en étendant les fonctionnalités du vérifieur. Les travaux de Bernardeschi *et al*. [Ber-02] sont très proches des nôtres, mais ne portent que sur un sous ensemble à 10 instructions. Ils réalisent la vérification à l'aide d'une abstraction sur les valeurs remplacées par des niveaux en gardant un contexte de sécurité pour évaluer l'entrée et la sortie d'une instruction conditionnelle.

Lorsque nous avons commencé ces travaux en 1999, il n'existait donc pas d'outil ni de prototype pour vérifier de telles propriétés de sécurité sur des ensembles d'application Java Card. Ce problème arrive actuellement dans les téléphones portables car il devient possible de charger sur son téléphone portable directement via Internet des applications qui si elles sont sûres en terme de typage (elles respectent la sémantique de Java) peuvent avoir des actions illégales sur les données des autres applications résidentes sur le mobile.

3.1.2. Non interférence

Afin de vérifier la confidentialité, il est nécessaire de s'assurer que les informations secrètes ne soient pas disséminées dans des variables facilement accessibles. Il faut donc classifier les données en au moins deux catégories, les données secrètes (*high level*) et les données publiques (*low level*). Il est possible comme nous le montrons plus loin d'utiliser des classifications intermédiaires représentées par un treillis. Il faut s'assurer que les programmes satisfassent la propriété de non interférence.

Cette propriété peut être définie par le fait qu'une donnée secrète ne doit pas interférer avec un comportement observable de l'application. Il faut naturellement définir ce que nous entendons par observable. La figure suivante illustre ce principe. Si nous considérons le programme P comme une boîte noire qui fournit des sorties publiques (y^{low}) et secrètes (y^{high}) à partir d'entrées publiques (x^{low}) et secrètes (x^{high}) alors le résultat publique (y^{low}) ne doit pas dépendre des entrées secrètes (x^{high}). Les sorties secrètes peuvent dépendre de variables publiques ou secrètes.

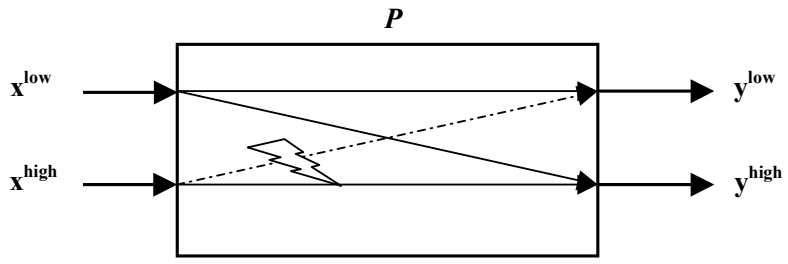

Figure 25 Non interférence entre des entrées secrètes et des sorties publiques

Il existe de nombreuses manières de créer des canaux illégaux entre des entrées secrètes et des sorties publiques. La plus simple, le canal explicite consiste à copier une variable secrète dans une variable publique. Ceci peut être facilement détecté par une classique analyse de flot. D'autres sont difficiles comme les canaux implicites. Le premier exemple consiste à copier bit par bit le contenu d'une variable secrète dans une variable publique en utilisant le résultat d'un test. Dans la figure

suivante, la copie est faite en ayant positionné la variable `mask` à une puissance de deux. Ce dernier sert de curseur pour la copie bit à bit.

```
while (mask !=0){
    if (PIN & mask !=0) ‘
      y := y|mask
    }
    mask :=mask/2
}
```

Figure 26 Canal implicite : recopie bit par bit

Une façon d'éviter un tel flux serait de renforcer la propriété afin qu'aucun changement sur une variable privée n'interfère sur le contenu d'une variable publique. Auquel cas ce flux serait détecté car toute modification de `PIN` entraîne une modification de `y`.

Cependant une telle propriété n'est pas nécessairement suffisante car si la copie est réalisée en une seule passe dans l'exemple précédent, il est plus subtil de l'exécuter en plusieurs exécutions. Auquel cas une simple exécution ne transmet qu'une partie du secret. L'exécution multiple d'un programme peut amener d'autres formes de flux. Dans la figure suivante, il est possible d'exploiter le fait de bloquer un programme pour transférer une partie de l'information. Le non blocage permet d'inférer une autre valeur partielle de la variable secrète.

```
while (PIN & mask ==0){}    // boucle infinie si zéro
y := y | mask               // mise à jour et terminaison
```

Figure 27 Canal caché par non terminaison

On voit bien que la propriété précédente n'est pas suffisamment forte car elle ne prend pas en compte la terminaison. En fait, pour les programmes Java il faut prendre en compte aussi les terminaison par exception. Par exemple le programme suivant utilise le comportement anormal de traitement d'exception arithmétique pour induire un comportement observable et donc faire fuir de l'information.

```
if (1/ (PIN & mask)) {}  //lancement d’exception si le bit
                           est nul
y := y | mask            // mise à jour
```

Figure 28 Canal caché par opération partielle

Ce programme produit une division par zéro si le bit observé (pointé par la variable `mask`) est nul. Pour s'en prémunir la propriété de non interférence définie plus haut ne suffit plus il faut prendre en compte aussi les terminaisons anormales.

Nous avons choisi d'utiliser une relation plus forte, la dépendance causale. Ce modèle comme celui de la non-interférence [Gog-84], garantit que les dépendances entre objets du système ne peuvent être utilisées pour établir des canaux de communication indirects. Lorsqu'un système est décrit en utilisant les constructions d'un langage de programmation ou de spécification il est possible de déterminer les dépendances syntaxiques. Par exemple, en examinant la construction `if (condition) then x := expression` il est possible de déduire qu'à cet instant la valeur de x dépend potentiellement de tous les objets figurant dans la condition et l'expression. Il s'agit

d'une approximation car certains éléments de la condition peuvent ne pas avoir d'effet sur la valeur de x. Pour assurer la sécurité du système il faut garantir qu'un attaquant (un utilisateur ou une entité logicielle) ne peut pas déduire de l'information à partir de l'observation de l'information à laquelle il a licitement accès. Le modèle des dépendances s'intéresse aux conditions susceptibles d'autoriser une attaque. C'est un modèle pessimiste car il détecte des failles de sécurité qui ne sont pas exploitables. Il faut donc analyser les failles détectées et s'assurer qu'elles représentent effectivement une opportunité pour un attaquant. Ce modèle permet de résoudre les différents cas décrits précédemment.

3.1.3. Modèle des dépendances sûres et son implémentation

Modèle des dépendances sûres.

Pour notre domaine d'application, les cartes à puce multi-applicative, nous utilisons un modèle basé sur l'évolution au cours du temps des valeurs d'objets. Ces objets sont les paramètres d'entrée des commandes envoyées à la carte, les valeurs de retour des commandes ainsi que les objets internes non observables. S'il existe un règlement de sécurité qui définit pour chaque sujet ce qu'il est autorisé à observer, il est possible de définir une propriété assurant que les dépendances ne peuvent pas être exploitées.

Nous montrons en réutilisant le modèle des dépendances sûres [Bie-92] que la valeur d'un objet de sortie du système ayant un niveau de sécurité donné ne dépend que de la valeur d'objets d'entrée dont les niveaux de sécurité lui sont inférieurs. Nous utilisons une politique de sécurité multi-niveaux en définissant des niveaux pour chaque acteur avec des niveaux intermédiaires pour les modules de partage d'information. La relation de flux entre les différents niveaux est utilisée pour autoriser ou interdire l'échange d'information entre les entités. Les différents niveaux et la relation de flux forment un semi-treillis borné avec le niveau privé comme borne supérieure et le niveau public comme borne inférieure.

Chaque fournisseur d'application possède un niveau de sécurité privé et, s'il partage des données ou des services avec un autre fournisseur d'application, un niveau spécifique est assigné aux méthodes et objets partagés. Ce niveau de sécurité est dominé par le niveau de sécurité de chaque fournisseur de service.

Technique d'analyse : Une application est constituée d'un ensemble fini d'applets, chacune comprenant plusieurs classes et plusieurs méthodes. Afin de limiter la complexité de l'analyse, nous l'avons décomposée en un ensemble d'analyses locales sur un sous-ensemble de méthodes d'une seule applet. La décomposition nécessite d'identifier les points d'entrée des graphes. Deux types de méthodes doivent être examinés : les méthodes d'interface proposant des services à l'interface de partage et les méthodes faisant elles-mêmes appel à des méthodes publiques et partageables d'autres applets. Pour un programme donné, nous considérons comme des valeurs d'entrée le résultat de l'invocation de méthodes externes ainsi que la lecture d'attributs d'objet. En sortie nous avons l'appel aux méthodes externes et la modification des attributs. Nous associons ensuite à tous les attributs et aux paramètres des méthodes appelantes des niveaux de sécurité. Nous restreignons ainsi l'analyse au contenu du flot d'information à l'intérieur d'une applet en contrôlant les niveaux des paramètres de sortie. Ces derniers deviennent donc des hypothèses pour la poursuite de l'analyse. Grâce à ce principe de décomposition il est possible de se focaliser sur l'ajout d'une applet à une configuration donnée. Tant que la politique de sécurité n'est pas modifiée (le treillis) il n'est pas nécessaire d'analyser à nouveau les applets déjà chargées. Une modification du treillis nécessite d'analyser à nouveau les méthodes dont les objets en entrée et/ou en sortie ont été affectés par ce changement.

Notre méthode d'analyse agit au niveau code binaire (byte code Java) et est basée sur trois éléments :

- Abstraction des valeurs et remplacement par des niveaux de sécurité,
- Insertion d'un invariant représentant une condition suffisante de la propriété de sécurité,
- Vérification d'un invariant sur un modèle du programme par un vérifieur de modèle.

Nous utilisons la modularité de SMV en construisant le graphe d'appel à partir des points d'entrée (appel par la méthode *process* et appel via l'interface de partage), en réalisant une abstraction du byte code de chaque méthode utilisée et en insérant les invariants nécessaires à la vérification.

Invariant de sécurité un invariant est associé à chaque variables de sortie. Pour le paramètre de retour de la méthode appelée le niveau calculé doit être dominé par le niveau de l'interaction défini par le treillis. Lorsqu'une méthode est invoquée alors le niveau calculé de chaque paramètre doit être dominé par le niveau de cette interaction. Lorsqu'un objet est modifié, le niveau calculé doit être dominé par le niveau de cet objet. Nous insérons automatiquement dans le modèle du programme ces invariants devant être vérifiés.

Les propriétés sont définies pour chaque exécution du programme. Nous utilisons pour cela la construction **assert** de SMV. Les propriétés sont des formules de la logique temporelle linéaire, les opérateurs sont **G** (toujours), **F** (finalement), **X** (suivant) et **U** (jusqu'à). Mais nous n'utilisons que l'opérateur **G**.

Construction du graphe d'appel : La méthode d'analyse consiste à évaluer les flux d'information pouvant être transmis à d'autres applets n'appartenant pas au même package. Il faut donc vérifier pour toutes les méthodes pouvant être activées au travers d'une interface partagée, que dans leur graphe d'appel elles invoquent une méthode appartenant à un autre package. Lors de la construction du graphe, nous analysons si une méthode possède une instruction de type invoke dans son byte code. Dès lors, nous parcourons cette nouvelle méthode et nous vérifions si elle réalise des appels. Les méthodes auxquelles nous nous intéressons dans la construction des graphes sont :

- les méthodes appartenant aux interfaces de partage**,**
- les méthodes appelant à travers l'ensemble de leurs invocations une méthode appartenant à une interface de partage (ceci correspond au byte code *invokeInterface*).

Le calcul du graphe d'appel se fait au travers de la construction d'un arbre n-aire. Les arêtes sont étiquetées par le type d'appel, après le nom de la méthode sont indiqués les paramètres d'appel suivi du paramètre de retour (si il existe). Le graphe d'appel obtenu est un arbre d'appel (les applications Java Card n'autorisent pas la récursivité). A chaque instruction *invokeSpecial*, il faut développer les sous-arbres, alors que les instructions *invokeStatic*, *invokeVirtual* et *invokeInterface* sont des nœuds terminaux du graphe. La construction des graphes d'appel ne prend pas en compte la possibilité de cycle dans le graphe car la récursivité n'est pas autorisée dans les applications Java Card.

Edition de lien : D'une manière générale, le problème, lorsque l'on est en présence d'une invocation de méthode ou d'interface est de trouver quelle portion de byte code il est nécessaire d'inclure dans le modèle. Ceci n'est pas toujours simple à déterminer statiquement à cause des problèmes causés par la surcharge et la redéfinition des méthodes ainsi que l'utilisation de références typées par des interfaces.

- Surcharge des méthodes : Un même nom de méthode peut être utilisé pour désigner plusieurs méthodes différentes d'une classe, à condition que leur signature diffère. Dans tous les traitements, il faudra donc utiliser non pas un simple nom de méthode,

mais, comme dans la JVM, une spécification de méthode qui est composée d'un nom de package, d'un nom de classe, d'un nom de méthode et d'un descripteur (c'est-à-dire de la signature de la méthode). Un exemple de spécification de méthode est par exemple : `com/gemplus/myPackage/MyClass/myMethod(I)V`, où le nom du package est `com/gemplus/myPackage`, le nom de la classe est `MyClass`, le nom de la méthode `myMethod` et le descripteur `(I)V` (la méthode prend un entier en paramètre et ne rend rien).

- Redéfinition des méthodes : Une sous-classe peut redéfinir une classe présente dans sa classe mère. Cette redéfinition est bien entendu appelée à la place de la méthode de la classe mère lorsque la méthode est invoquée sur une instance de la sous-classe. La difficulté vient de la manipulation de références typées avec une sur-classe d'une classe redéfinissant des méthodes. En effet, il est impossible de savoir à quelle classe exacte appartiendra l'objet dont la méthode sera manipulée. La classe exacte, et donc la méthode exacte appelée est déterminée à l'exécution par la JVM. Le byte code `invokevirtual` est utilisé pour ce type d'invocations.

- Pour invoquer, dans une méthode d'une instance d'une classe redéfinissant une méthode, la méthode redéfinie de la classe mère, il est possible d'utiliser le mot clé `super`. Par exemple, si la classe A définit la méthode `f()`, une sous classe B peut redéfinir cette méthode `f`, et à l'intérieur de cette redéfinition de f (et à cet endroit là seulement) peut invoquer la méthode f de A par `super.f()`. Il faut noter que le typage de B en A ne permet absolument pas d'invoquer sur un objet de classe B la méthode f de A. Signalons enfin que si l'accès à une méthode de la classe mère est possible par le mot clé super, l'accès à une méthode (doublement redéfinie) de la classe grand-mère n'est pas possible, une syntaxe du type `super.super.x()` étant illégale. Une exception cependant : si la classe mère ne définit pas elle-même la méthode redéfinie par sa classe fille, mais l'hérite d'une classe ancêtre. Dans ce dernier cas, c'est la méthode de la plus proche classe ascendante qui sera invoquée par `super.x()`.

- Les méthodes de classes se comportent différemment, et ne peuvent pas être redéfinies mais seulement cachées. Si une classe C redéfinit une méthode f, définie dans sa classe mère B, il est possible d'utiliser la syntaxe `A.f()` pour invoquer la méthode définie par A ou celle utilisée précédemment `super.f()`. Enfin, le transtypage est efficace dans le cas des méthodes de classes.

- Les méthodes signalées par le mot clé `final`, ainsi que les méthodes définies par une classe `final` ne peuvent pas être redéfinies par des classes filles.

- Les méthodes appelées sur des objets manipulés par l'intermédiaire d'une référence typée par une interface comportent le même type d'indétermination que les méthodes redéfinies. Il est impossible de savoir par avance la classe exacte de l'objet qui sera manipulé, et, de ce fait, la méthode exacte qui sera appelée. Le byte code `invokeinterface` est utilisé pour gérer ce type d'appels.

En ce qui concerne la génération de code SMV, il est nécessaire d'analyser les classes et les interfaces utilisées dans une applet afin de construire l'arbre d'héritage correspondant. Cette information est ensuite utilisée pour déterminer s'il peut y avoir une ambiguïté sur un appel de méthode, et, si oui, pour lister les méthodes potentiellement appelables et générer un appel de méthode non déterministe dans le modèle.

Elimination des sous routines : Les sous routines sont un moyen d'implémenter de manière efficace la construction `try-finally` de Java. Sans ce mécanisme il est nécessaire de n-pliquer le corps du traitement d'exception. Lors d'un appel à une sous routine, la première instruction que fait la sous routine est de sauvegarder l'adresse de retour dans une variable locale. Ainsi l'instruction de retour reprend cette adresse pour retourner au programme appelant. Or notre abstraction nous fait perdre la valeur de la variable pour lui substituer un niveau. Afin de ne pas mémoriser l'adresse de retour de sous routine dans une variable locale, nous déplions le traitement dans le corps du programme. Il est à noter que ceci peut poser problème si les sous routines sont imbriquées, par exemple une construction `try-finally` à l'intérieur d'une autre construction `finally`. Il est possible dans ce cas d'avoir une explosion exponentielle de la taille du code.

Modification de la sémantique des byte codes : Lors de la traduction de java vers SMV nous modifions la sémantique de chaque byte code. Nous calculons pour chaque instruction et chaque objet du programme les dépendances. Il faut prendre en compte dans cet ensemble non seulement les objets d'entrée et les objets de sortie mais aussi les objets internes comme la pile d'opérande, le compteur de programme ainsi que le sommet de pile.

Nous définissons pour chaque instruction ses dépendances. Puis nous définissons la sémantique abstraite de toutes nos instructions. La sémantique des byte codes est liée à l'évolution du niveau de sécurité. L'ensemble de la définition de chaque byte code est donnée dans [Bie-00].

Portée des branchements conditionnels : Dans le modèle des dépendances sûres, il existe deux types de dépendances liées aux branchements conditionnels, pour lesquelles on veut vérifier la propagation d'information, les dépendances implicites et explicites. Lorsqu'on le programme entre dans un bloc d'instruction encadré par la portée du branchement conditionnel, le niveau minimal de ce bloc soit au moins celui de la variable évaluée. Ainsi il n'est pas possible de distinguer deux comportements différents. Cette approche nécessite de mémoriser le contexte (niveau de sécurité) du bloc précédent l'évaluation et de restituer ce contexte lors de la sortie du bloc. Il convient donc de faire une évaluation des portées des branchements. L'évaluation de la fin de portée est délicate au niveau du byte code dès que le flot de contrôle n'est plus structuré. Les instructions pouvant modifier le flot de contrôle en byte code Java sont `if` et ses avatars, `goto` et `tableswitch`.

Un graphe représentant un flot de contrôle structuré est un graphe décomposable est sous graphe n'ayant qu'une entrée ou qu'une sortie. La perte d'information lors de la compilation nous oblige à inférer le point de branchement. Si on prend l'hypothèse que le compilateur génère uniquement des graphes conditionnels structurés, une analyse simple détecte le point de jonction. Par graphe structuré, nous entendons qu'il n'y a qu'un point d'entrée : l'instruction conditionnelle et un point de sortie le point de jointure. Cependant cette hypothèse sur la construction correcte par un compilateur doit être relaxée, car de nombreux programmes sont construits directement en byte code pour optimiser la place mémoire. Dès lors il faut s'intéresser aux graphes non structurés.

Notre analyse de point de jonction doit être affinée mais repose toujours sur le même principe : faire dépendre le niveau cumulé du compteur de programme de la ou des conditions de branchement. Par exemple le programme Java suivant : `IF (a\/ not(b)) {THEN T1} ;T2 ; T3 END.` Dans le programme source, il semble que la dépendance s'arrête après T1, cependant dans le code cible ceci n'est plus difficile à inférer. Deux structures apparaissent : les entrées anormales et les sorties anormales. Ces cas sont liés à la conditionnelle composée *a ou non b*. Il est possible que le compilateur génère des boucles se chevauchant (optimisation), en effet la traduction se fait par des raccourcis ce qui implique un graphe non structuré.

Il est possible si le graphe est réductible de le transformer en boucles imbriquées par réplication de code. La réplication modifie le programme original en dépliant le code. Le programme ainsi obtenu est équivalent fonctionnellement mais sa sémantique et sa structure sont modifiées. Ceci pose évidement le problème lorsque l'on veut remonter au code source sur une trace de contre exemple. En transformant notre graphe non structuré en un graphe structuré, la technique utilisée précédemment peut s'appliquer. Nous avons utilisé un algorithme permettant d'analyser la réductibilité d'un graphe décrit dans [Cif-94].

3.1.4. Conclusions et extension

Nous avons utilisé notre prototype sur un ensemble d'applets du domaine bancaire un porte-monnaie électronique et deux applications de fidélité. Les expérimentations ont montré la capacité de notre outil à traiter des exemples réels [Bie-00] et [Bie-02]. L'analyse des interactions construit 51 graphes d'appel différent. Le vérifieur de modèle SMV vérifie la plupart des propriétés en moins de 10 secondes, le long temps de traitement ne dépasse pas 8 minutes. Nous avons ainsi pu détecter plusieurs flux illégaux qui auraient été impossibles à détecter par des audits traditionnels. Il faut maintenant étudier si ces flux sont exploitables pour générer une attaque réelle et obtenir illégalement un accès à des données (confidentialité) ou modifier ces données (intégrité).

Nous avons montré dans ces travaux la possibilité à partir du modèle théorique des dépendances causales sûres comment analyser des programmes Java et en déduire des traces pour analyser des flux illicites d'information. L'automatisation de la production de modèles à partir de programme est nécessaire pour pouvoir traiter des problèmes conséquents. Nous pensons que les techniques d'abstraction et de vérification par un vérifieur de modèle sont l'une des solutions permettant la vérification de l'implémentation correcte de politique de sécurité dans un petit objet mobile de sécurité. Ces travaux peuvent servir de base dans le cadre d'une certification critère commun pour les classes d'analyse de canaux cachés et de vulnérabilité.

Lors de ces travaux, les cartes à puce de type Java n'incluaient pas de capacité de multi-threading. Ainsi certains problèmes soulevés par Volpano n'ont pas été traité dans ces travaux. Cependant il semble que la tendance est au rapprochement des standards Java et Java Card et que désormais le multi-threading soit à prendre en compte.

3.2 Test d'application

Nous avons vu comment éviter d'introduire des fautes dans la conception d'un système. Cependant il n'est pas toujours possible d'utiliser cette technique (code source d'un tiers, routines cryptographique de bas niveau etc.). Dans ces cas, il faut éliminer les fautes résiduelles par le test. Le test est une activité de base dans le développement de logiciel mais dont le coût est très mal maîtrisé et en tout cas toujours considéré comme trop élevé. Afin de réduire le coût de la production de suites de test, il est possible d'utiliser des techniques de synthèse à partir de modèles.

Le processus d'élimination des erreurs est composé de trois étapes, la détection, l'élimination et la correction des fautes. Le mécanisme de détection est généralement le test, qui consiste à réaliser des stimulations sur une implémentation d'un système en relation avec son environnement afin d'en déduire la correction vis-à-vis d'une spécification de référence. Plusieurs cas sont possibles selon que le l'implantation testée est :

- une unité du développement, c'est le test unitaire,
- un ensemble d'unité intégrée, c'est du test d'intégration,

- le système complet tel qu'il sera livré, c'est du test de validation ou test de recette,
- le système après une phase de correction ou de maintenance, c'est du test de non-régression.

Le test fonctionnel (dit aussi boîte noire) ne tient pas compte de la structure du logiciel et le code de l'implémentation n'est pas connu. Ce type de test fait appel à des techniques de test de conformité dont l'objectif est de s'assurer qu'une implantation est conforme à sa spécification. L'ensemble des tests appliqués à l'implémentation peut être structuré sous forme d'un ensemble de cas de test. Un cas de test est un ensemble d'entrées du système, de conditions d'exécution et de résultats attendus. A chaque cas de test est associé un verdict. Un objectif de test est un critère de sélection d'un ou de plusieurs cas de test. L'utilisation d'objectif de test est un moyen de s'abstraire d'éléments constituant un cas de test, *i.e.* de l'identification de l'ensemble des entrées du système, des conditions d'exécution ou de l'identification des résultats obtenus.

L'oracle est l'entité donnant un verdict à la fin de l'exécution d'un cas de test. Si l'oracle est automatique, il doit pouvoir interpréter les spécifications qui doivent dès lors être cohérentes et complètes par rapport aux tests exécutés. Ceci se traduit généralement par un modèle formel exécutable de l'application accompagné d'une fonction de comparaison.

L'expérimentation consiste à faire interagir l'implémentation sous test. L'environnement est simulé par un testeur qui exécute en série des tests élémentaires appelés cas de test, cette série de test est appelée suite de test. Nos travaux ont porté sur la génération de suites de test à partir d'objectifs de test.

3.2.1. Techniques de test : positionnement

Les possibilités offertes par les techniques de génération automatique de suites de test sont intrinsèquement dépendantes des langages de modélisation auxquels elles s'appliquent. Si l'objectif d'un logiciel est de réaliser un traitement de données il faut que le langage de spécification permette de construire un modèle de ces données. Un certain nombre de langages formels construisent des abstractions des données et des comportements. D'une manière générale on peut décomposer le test de conformité en trois grandes approches. Il s'agit de :

- La synthèse de tests de conformité dans le contexte des systèmes asynchrones. Ce domaine s'est essentiellement développé dans le contexte des protocoles de télécommunications depuis une dizaine d'années. L'outil TGV de l'Irisa est représentatif de cette approche, fondée sur un modèle sous-jacent de système de transition. L'extension STG [Jez-98] prend en compte le traitement symbolique de données et UMLAUT intègre un environnement UML et a été utilisé sur des applications carte à puce.
- Le test de conformité se préoccupe essentiellement des aspects comportementaux (le contrôle, par opposition aux données). Le raisonnement formel sur les données fait appel à des manipulations symboliques, comme par exemple dans l'outil CASTING [Van-98] de l'Irisa. Dans le domaine des spécifications ensemblistes, les premiers travaux ont débuté par Dick et Faivre autour de VDM [Dic-93]. Les travaux du Laboratoire d'Informatique de Franche-Comté, qui visent à générer des cas de tests fonctionnels à partir d'une spécification logico-ensembliste B, figurent parmi les travaux les plus complets dans ce domaine. Dans ces travaux, l'utilisation de la notation B se situe au niveau le plus haut, sans raffinement. Cette technique est d'ailleurs expérimentée dans le domaine de la carte à puce.
- Le cadre des modèles synchrones est particulièrement adapté pour formaliser l'activité de test, les différents objets concernés (modèle, oracle, test) pouvant être décrits dans un cadre uniforme et modulaire. Les travaux autour du test pour le langage LUSTRE sont bien représentatifs de l'approche comme l'outil Lutess [Dub-99] qui sélectionne les structures

d'entrée et des valeurs afin de créer une distribution aléatoire. L'outil Lurett repose sur le même principe mais avec des types plus riches prenant en compte des valeurs entières et réelles.

Dans le cadre des cartes de type Java Card, les applications écrites en Java sont des serveurs réactifs assimilables à des systèmes synchrones. Cependant les cartes de type SIM évoluent vers un type de fonctionnement dit pro actif assimilable à un comportement asynchrone. Les applications, pour des raisons de sécurité, utilisent des mécanismes d'authentification basés sur des comportements mais manipulent des données confidentielles. Ce domaine d'application doit à la fois traiter les comportements et les données.

3.2.2. Le contexte Java et UML

La spécification de ces applications doit se faire en intégrant quatre paramètres a) le surcoût de modélisation lié à la génération de cas de test doit être limité, b) elle doit pouvoir être réalisée par des architectes ou des programmeurs industriels, c) elle doit pouvoir être réutilisée dans le cadre des certifications de type Critères Communs d) elle doit offrir la possibilité de spécifier des architectures orientées objet.

Pour que les deux premiers critères soient remplis il faut masquer le coté formel par un habillage *ad hoc*. Nous proposons donc d'utiliser UML pour ce faire ce qui permet de remplir les deux derniers critères. Dans une approche orientée objet, UML apparaît comme le meilleur candidat pour réaliser un modèle semi-formel (au sens des critères communs). Il intègre les avantages d'autres langages de spécification orientés objet tout en offrant des similitudes avec des langages formels. Cependant, les modèles UML ne permettent ni de vérifier des propriétés ni de générer des suites de tests. Il est donc nécessaire dans le cadre de notre processus de test d'identifier une transition entre UML et les langages formels utilisables.

Parmi les outils disponibles, l'outil UMLAUT apporte une couche UML au-dessus de TGV utilisé pour la génération de cas de test pour des systèmes à transitions étiquetées à entrée/sortie (IOLTS). L'outil UMLAUT/TGV permet de générer automatiquement un test abstrait à partir d'un objectif de test. Cet outil a néanmoins certaines limitations. D'une part, il est nécessaire que les paramètres des méthodes, dans les objectifs de tests, soient instanciés pour que l'outil puisse générer un test abstrait. D'autre part, l'outil ne génère qu'un seul test abstrait par exécution à partir d'un objectif de test. Nos travaux ont porté sur la méthodologie de spécification de modèle UML testable [Mar-01] et l'abstraction des objectifs de test en schéma de test [Bou-00a], [Bou-00b] et [Bou-03]

3.2.3. Modèle UML testable

Le modèle UML nous utilisons deux vues : le diagramme de classe et les automates de type statechart. Dans la mesure où TGV ne travaille que sur un système clôt, il faut que la spécification reflète cette caractéristique. Il faut donc inclure l'environnement dans la spécification afin qu'il n'existe aucune variable libre. Le modèle UML inclut donc le système sous test et des acteurs représentant l'environnement.

Le système complet est donc déduit de la composition du comportement du système et des acteurs. L'automate représentant chaque acteur doit donc appeler les méthodes de l'implémentation sous test. De même l'acteur doit pouvoir recevoir le retour asynchrone des méthodes appelées et des paramètres associés. Il faut donc créer artificiellement des méthodes appelables par la spécification afin de transférer ces paramètres.

Il y a donc une altération obligatoire du modèle due à la technologie utilisée. Dans le cadre d'une utilisation industrielle nous oblige à maintenir deux modèles si le modèle de conception est utilisé pour générer le squelette des objets. Les statechart doivent ensuite être complété par des pré et post conditions. La sémantique des actions dans UMLAUT est actuellement décrite en Eiffel.

3.2.4. Génération d'objectif de test à partir de schéma de test

L'efficacité d'une stratégie de test de conformité à base d'objectifs de tests repose en partie sur la pertinence des objectifs de tests. La génération d'objectifs de tests est intrinsèquement dépendante du logiciel testé. Les objectifs doivent potentiellement permettre de couvrir la totalité des comportements possibles y compris les comportements malveillants. Il faut donc décrire les scénarios associés au test de chaque fonction, c'est-à-dire définir les séquences d'appels de méthodes. Ensuite en l'absence de synthèse de données, il faut choisir les valeurs des paramètres des méthodes. Ce choix est basé sur une hypothèse d'uniformité des domaines de valeur des paramètres. Cet ensemble de valeur est réalisé par une analyse partitionnelle qui définit pour chaque sous domaine une valeur de donnée. Nous générons ensuite un ensemble d'objectifs de test.

Les outils et techniques utilisés dans ce processus de génération d'objectifs de test et de synthèse de tests ont été en grande partie développés par l'IRISA [Jer-99] ou sont issus d'une collaboration entre GEMPLUS et l'IRISA [Bou-01]. Ils contiennent notamment :

- un générateur d'objectifs de tests, buildTP,
- un outil de transformation de modèles UML : UMLAUT,
- un outil de génération de cas de tests abstraits : TGV,
- un outil de concrétisation de cas de test abstraits : Aut2java.

L'idée d'introduire un niveau d'abstraction supplémentaire à TGV vient du constat fait par H. Martin [Mar-01] que de nombreux objectifs de test sont assez similaires et qu'il est possible de synthétiser par des schémas de test par différentes abstractions. Il est possible de réaliser une abstraction sur les paramètres, les méthodes en les regroupant sous un même label, ainsi que toutes les instances d'un même objet. L'outil buildTP construit un schéma de test à partir d'une suite d'opérations correspondant à un ensemble de méthodes de l'application et un ensemble de paramètres. Le schéma est ensuite déplié pour générer des objectifs de tests. La fonction ensuite de TGV est d'identifier toutes les opérations devant préalablement être appelée pour permettre de calculer les initialisations requises par les objectifs de tests.

La traduction des suites de tests abstraites en suites de test concrètes peut se faire automatiquement en deux étapes successives avec Aut2Java. Premièrement, le modèle de l'application est un modèle abstrait au sens où il fait abstraction de certains comportements et certains traitement de données. Une des fonctions consiste à réintégrer ces paramètres au niveau des appels de méthodes afin d'obtenir des cas de tests exécutables. Dans une seconde étape, l'outil traduit les cas de test dans le langage cible. Il analyse chaque cas de test abstrait pour construire une séquence d'appel à des méthodes d'un driver de test. Chaque méthode passe en paramètre les valeurs de données en entrée ainsi que les résultats attendus en sortie. A chaque appel de méthode, le driver de test appelle la méthode de l'application testée avec les données en entrée, reçoit les données émises par la carte et vérifie la conformité entre les résultats attendus et les résultats obtenus.

Nous avons identifié plusieurs limites en utilisant la technologie UMLAUT/TGV associée à nos outils buildTP et Aut2Jav. La spécification des objectifs de test est réalisée dans la syntaxe des transitions exprimées dans le modèle LTS utilisé par TGV. Il est impératif que les objectifs de test soient intégrés dans le formalisme UML en utilisant des diagrammes de séquence. Dans notre méthode nous réalisons une analyse partitionnelle sur les données pour ne gérer que les flots de

contrôle. Il est nécessaire de raisonner à la fois sur les données et sur les contrôles pour obtenir un processus unifié de test en générant automatiquement les valeurs en entrée en fonction des valeurs particulières extraites de la spécification.

Ces travaux ont été poursuivis dans le projet RNTL COTE par un prototype complet intégrant différents outils dont l'utilisation de Casting pour générer les données d'entrée. Le principe de l'outil buildTP a été repris dans la conception de l'outil TOBIAS par le LSR [Bon-01].

3.2.5. Conclusion sur la méthode

L'objectif était de pouvoir étendre l'ensemble des techniques de test appliquées sur les modèles afin de les transposer à l'approche par composants qui semble être une des possibilités de programmation pour les futures générations de carte à puce. Ceci devait nous permettre, avec un volume de travail industriellement acceptable, de définir des tests à un niveau d'abstraction plus élevé et d'obtenir ainsi une couverture externe des tests plus étendue que celle obtenue par des processus classiques. Les techniques de tests élaborées pendant ce projet sont définies avec un niveau d'abstraction en adéquation avec celui du modèle, rendant ainsi la méthode globale et ses applications indépendantes de la technologie employée pour implémenter et distribuer les composants.

L'utilisation d'UML pour l'analyse et la conception d'applications objets et de composants est une réalité industrielle. Néanmoins les gains de productivité dans le cycle de développement peuvent être grandement améliorés si la phase de test s'intègre dans un cadre plus global d'utilisation d'UML. Un point important pour l'industriel est d'apporter une vue plus unifiée du développement logiciel et donc une potentielle amélioration de productivité.

La génération automatique de suites de tests est une solution qui s'impose depuis des années mais qui dans la pratique se révèle assez peu utilisée. Ce travail a pour ambition de lever les derniers freins à son application en assemblant différentes technologies dans un environnement unifié. Le choix d'UML permet de construire à partir des diagrammes et des informations présentes dans ses diagrammes des modèles suffisamment précis et non ambigus pour appliquer des techniques formelles maintenant bien appréhendées. Il est également bien accepté par les développeurs pour les spécifications des architectures statiques. L'utilisation d'UML supportée par UMLAUT offre donc un bon compromis entre les besoins techniques et les contraintes industrielles.

La génération de suites de tests dans l'industrie de la carte à puce se base entre autres sur l'expertise des testeurs dans le domaine, ce qui correspond d'un point de vue théorique à des modèles de fautes (base de données sur la caractérisation des fautes). Une des premières conséquences est que le test d'application carte à puce se base essentiellement sur des objectifs de test. Ces objectifs ne couvrent pas toute l'application réduisant ainsi le coût moyennant un risque quantifiable. Pour que la génération automatique de suite de tests soit transférable dans l'industrie de la carte à puce il faut au minimum qu'elle permette aux testeurs de pouvoir continuer à utiliser ces modèles de fautes. En effet, ces modèles de fautes font partie des acquis de l'entreprise et assurent un certain niveau de qualité des produits développés. La formalisation des modèles UML par UMLAUT autorise les testeurs à définir eux-mêmes leurs objectifs de test. UMLAUT-TGV fait la synthèse de ces tests abstraits et permet de générer les cas de tests concrets. Dans le projet COTE, l'intégration dans un atelier UML des drivers de test ainsi que l'environnement Junit permet de transformer ces cas de tests en exécutables et obtenir directement le rapport d'exécution pour la génération de la documentation finale.

Deux points étaient importants : l'optimisation de la génération des objectifs de test et l'intégration des données. Le processus d'identification des objectifs de test se fait manuellement à partir de la spécification informelle du plan de test. La génération automatique des objectifs de test avec TOBIAS est un pas important pour l'automatisation par expansion des cas d'utilisation et/ou l'assistance au testeur dans les phases de gestion des combinatoires entre les actions. Certaines optimisations de TOBIAS semblent faisables à court terme (extension de notions de groupes, abstraction sur les méthodes…). D'autres optimisations nécessitent d'autres développements comme par exemple la reconnaissance automatique de suites de test déjà incluses pour d'autres fonctions sous test.

Pour l'intégration des données, il n'a pas été possible d'analyser l'apport de CASTING pour différentes raisons (types supportés trop simples, pas de support d'appel de méthode dans les pré conditions…). L'utilisateur doit encore se contenter d'une analyse partitionnelle sur les données pour choisir les valeurs pertinentes à fournir à TOBIAS. Un premier effort dans le projet a été fourni sur les notions de couvertures de test à partir de schémas. Ces notions sont souvent utilisées pour contrôler l'efficacité des processus de tests. Elles restent un élément essentiel pour augmenter la confiance que peut avoir un responsable qualité dans un processus de test de logiciel.

D'autres modèles que ceux en UML peuvent être utilisés pour spécifier des applications pour Java, notamment comme JML. Le portage de cette méthode et d'une partie des outils pourrait être un bon complément pour ces techniques. Ceci pourrait être utile dans les domaines applicatifs de la carte à puce où UML ne s'est pas imposé et où, malgré tout, l'automatisation de la génération des suites de test reste d'actualité.

3.2.6. Une alternative : JACK

Nous avons vu l'utilisation de technique de test pour la validation d'application Java. Nous pouvons aussi appliquer les techniques d'évitement de faute au niveau applicatif comme nous l'avons fait au niveau du système d'exploitation. Il existe d'autres possibilités comme la conception correcte à partir de spécification formelle et la génération automatique de code pour Java ce qui a été étudiée dans le cadre du projet RNTL BOM. Nous proposons aussi de regarder le langage JML (Java Modelling Language) de l'université d'IOWA permet de spécifier à l'aide de pré et post conditions le comportement d'une application Java. JML est un langage de spécification de classes Java sous la forme d'annotations décrivant le comportement des méthodes par des pré et post conditions ainsi que les propriétés devant être respectées. Les invariants de classe correspondent à des propriétés sur les variables de classes, les pré conditions associées aux méthodes expriment les propriétés nécessaires à l'appel de ces méthodes et les post conditions décrivent le comportement des méthodes par les propriétés qu'elles garantissent après leur exécution. Quelques mots clés et quelques constructions logiques ont été rajoutés à la syntaxe Java laissant ainsi JML accessible à la communauté des développeurs Java.

Cependant il n'existe pas d'outil industriel pouvant être utilisé dans une chaîne de développement permettant de vérifier si le code Java correspond bien à sa spécification détaillée. Nous avons développé une preuve de concept d'un tel outil : Jack [Bur-02b]. Cet outil prend en entrée un programme Java (un ensemble de classe) annoté avec des commentaires JML et produit à l'aide d'un générateur d'obligation de preuve, des théorèmes pouvant être traités par un outil automatique de démonstration (prouveur) comme celui intégré dans l'Atelier B. Un visualiseur retranscrit dans une vision Java les théorèmes à démontrer permettant ainsi une bonne visualisation de la couverture des obligations de preuve.

Nous savons que dans un environnement industriel où la pression pour une productivité accrue des développeurs est prédominante, les chefs de projet et autres décisionnaires ne sont pas prêts à un saut technologique risqué, à une nouvelle phase d'apprentissage pour leurs développeurs. Cette technologie pourrait pourtant être adoptée naturellement et non par obligation (*i.e.* certification, cahier des charges…) en apportant des solutions rigoureuses et adaptées. Par exemple, l'utilisation de programmes annotés associé à un outil cachant le caractère mathématique des concepts manipulés est une solution permettant l'adoption par les utilisateurs de cette technologie. Cacher les concepts nécessite de fournir une vue adaptée des théorèmes, aider l'utilisateur dans sa compréhension en indiquant clairement les cas en cours de traitement, d'améliorer le taux de preuve automatique et de suppléer la preuve interactive par une activité de test.

Les liens entre la preuve et le test ont été explorés depuis quelques années par les communautés scientifique et industrielle. De nombreux auteurs ont constaté qu'une activité de preuve permettait de montrer l'absence d'erreurs, alors que l'activité de test cherche à mettre en évidence leur présence. Notre projet vise la complémentarité de la preuve et du test au niveau des tests unitaires, pour des logiciels associés à une spécification, afin d'atteindre un niveau élevé de confiance, tout en diminuant l'effort de test. Le processus commence par une phase d'utilisation d'outils automatiques de preuve et de test. Cette activité qui ne réclame pas ou peu d'intervention humaine se réalise à faible coût et donne une confiance totale dans l'adéquation entre les parties prouvées et leur spécification. Dans un deuxième temps, une nouvelle phase de test concentrera l'effort sur les parties du logiciel qui n'ont pas été démontrées automatiquement.

Cette activité suppose une intervention humaine, mais à un degré de qualification compatible avec celui des ingénieurs de développement. L'activité de test peut mener à la détection d'erreurs, qui rendent vaine toute activité de preuve sur les parties concernées. Pour les autres parties, elle ne démontre pas l'absence d'erreur mais augmente le niveau de confiance dans le logiciel et permet d'envisager la mise en œuvre d'une étape de preuve interactive, activité plus coûteuse en temps et qui peut requérir du personnel spécialisé.

Cette complémentarité entre les activités de preuve et de test au niveau unitaire a été mise en avant par plusieurs équipes de recherche pour justifier une activité de recherche sur la synthèse de tests à partir de spécifications formelles. Actuellement notre prototype ne gère pas encore la partie test. Ceci représente pour nous un axe de recherche important. Il existe différentes solutions pour générer des cas de test à partir de spécifications JML.

Le caractère exécutable d'une majorité d'applications écrites en JML et l'existence d'un outil qui vérifie dynamiquement ces assertions permettent d'utiliser une spécification JML comme oracle dans le cadre d'un processus de test de conformité. L'outil JMLUnit, développé à l'université d'Iowa, constitue le premier exemple d'un tel outil. A l'instar de TOBIAS, cet outil permet le test combinatoire des méthodes d'une classe à partir d'états initiaux et de valeurs de paramètres identifiés par l'ingénieur de test. L'outil est couplé à JUnit pour l'exécution des tests. L'outil reste néanmoins plus simple que TOBIAS (il ne permet pas de générer des séquences d'appels et offre moins de capacités de structuration des tests et des valeurs). L'outil Korat, développé par l'équipe de D. Jackson au MIT, constitue une autre solution intéressante. Il étend la génération combinatoire aux structures d'objets et permet de générer de manière exhaustive toutes les structures d'objets qui correspondent à une spécification. A notre connaissance, JMLUnit et Korat sont les seuls outils disponibles aujourd'hui pour le test de conformité à partir de spécifications JML.

Diverses techniques de test de conformité ont été développées dans le cadre de spécifications à base de modèles. Aujourd'hui, les outils les plus performants qui se basent sur ces techniques sont CASTING, développé à l'IRISA, et BZ-Testing-tools de l'Université de Franche-Comté. Ces deux

outils partent de spécifications B et utilisent des techniques de programmation logique avec contraintes pour la génération des tests. Leur adaptation à JML nécessite néanmoins un travail substantiel pour prendre en compte la dimension objet de ce langage et permettre la traduction de JML/Java vers un solveur de contraintes. Le projet CASSIS de l'INRIA est une extension de ces travaux par intégration de techniques d'analyse d'atteignabilité et l'amélioration des techniques de preuve.

4. Perspectives

Ces travaux ont montré des avancées concrètes dans le cadre de l'application de méthodes rigoureuses pour la conception et la vérification de systèmes enfouis. Ils s'inscrivent dans le cadre plus vaste du thème de recherche sur la sûreté et la sécurité des OS enfouis que je développe au sein du Gemplus Research Labs. Nous nous intéressons à la notion de composants de confiance (Trusted Component) dans des systèmes (Trust Driven Solution). Dans ce cadre, l'analyse du système doit montrer que la confiance mise dans un composant n'est pas altérée par son insertion dans un monde potentiellement hostile et ce dynamiquement si possible. Il faut donc s'assurer qu'un certain nombre de propriétés de sécurité ne sont pas invalidées par l'environnement du composant. Les travaux menés au niveau de la plate-forme doivent donc être étendus à l'application d'abord en tant qu'application isolée puis en tant que système distribué.

La conception correcte d'un système est plus complexe qu'un composant logiciel comme par exemple notre carte GemClassifier. Il faut adapter les techniques et/ou les combiner comme le montrent Schneider *et al.* [Sch-02] pour pouvoir spécifier à la fois le comportement de chaque composant mais aussi les interactions au travers des communications. Des travaux récents autours de la méthode B sur soit le B évènementiel [Abr-98] soit le raffinement distribué proposent des pistes intéressantes. Si le système est conçu à partir d'éléments tiers d'autres méthodes doivent êtres utilisées comme une approche à base de contrats pour exprimer les propriétés de sécurité que garantit le composant à charger. Dès lors d'autres problèmes se posent sur la possibilité d'assembler de tels composants et de s'assurer que globalement ils assurent la propriété requise.

4.1 Vérification statique embarquée et système de type

4.1.1. Flux illicites

Le travail fait sur les flux illicites n'est pas transposable dans ce cadre car il repose sur une analyse a priori et donc statique d'une configuration. De plus, le mécanisme de vérification avec un vérifieur de modèle n'est plus adapté aux contraintes de la carte. Il faut donc envisager cette vérification de la non-interférence entre application d'une manière plus dynamique comme par exemple lors du chargement du code mobile dans la carte. Cette propriété de non-interférence entre application se complexifie si le système autorise le multi-threading. Dans un futur très proche, les fonctionnalités offertes par la carte seront très proches des systèmes d'exploitation pour petits systèmes enfouis comme les téléphones portables. Il faut donc se projeter sur les problèmes que poseront demain de tels systèmes. La synthèse de [Fou-01] montre comment il est possible de faire fuir de l'information entre deux applications en utilisant les propriétés de l'ordonnanceur. Auquel cas notre analyse de flux illicites outre son caractère statique sera inadaptée pour les futurs systèmes d'exploitation des petits objets portables. Il faut donc étudier d'autres solutions soit statiques (lors du chargement) soit dynamiques pouvant être embarquées dans un tel système. Des solutions à base de systèmes de type ont été proposées dans la littérature pour résoudre ce problème. Il convient de s'assurer que ces solutions sont applicables réellement car ces solutions fournissent en général des résultats peu précis et peuvent rejeter des programmes sûrs.

4.1.2. Contrôle des ressources

Il s'agit alors de concevoir une sorte de vérifieur d'application sachant traiter des informations supplémentaires. Ces informations peuvent être vues comme un contrat que l'application propose au système hôte. Pour des raisons d'optimisation de l'exécution il n'est pas possible d'insérer des assertions dans le programme en les vérifiant lors de l'exécution. Il est nécessaire de passer par une phase de vérification lors du chargement.. Dès lors, on peut aussi s'intéresser à une propriété difficilement vérifiable actuellement : la disponibilité. Cette propriété est certainement le prochain challenge à relever pour les petits systèmes afin de se prémunir contre des attaques de type déni de service. Ces attaques peuvent bloquer les ressources du système d'exploitation (le processeur, la mémoire, les communications). Des solutions partielles existent (ordonnanceur temps réel mou) pour éviter le blocage du processeur lesquelles peuvent elles aussi être guidées par une analyse préliminaire du programme.

4.1.3. Sécurité des OS

Les mécanismes de sécurité à mettre en place dans les OS enfouis sont pour la plupart identiques à ceux des OS conventionnel c'est-à-dire : authentification des entités, contrôle d'accès aux informations, confinement des applications. Le confinement des applications et le contrôle d'accès ne permettent pas de se prémunir contre des attaques éventuelles. L'exécution d'une séquence légale d'action peut aboutir à une violation de la politique de sécurité (*e.g.* la non-interférence dans un OS avec multi-threading). Un mécanisme supplémentaire de contrôle de séquence d'action peut éviter de telle attaques. Il s'agit dès lors d'une prise en compte dynamique du comportement du système pour détecter des comportements illicites. Un autre point concerne le chargement dynamique d'application. En utilisant les mécanismes classique d'analyse statique (e.g. le vérifieur de byte code) lors du chargement garantit une mise à jour sécurisée. Cependant si on s'intéresse à la mise à jour de fonctions systèmes potentiellement écrites en langage de bas niveau il devient nécessaire d'adapter le mécanisme de vérification et potentiellement de doter le langage avec lequel sont écrites ces fonctions d'annotations vérifiables au chargement (e.g. le système TAL). On voit donc que différentes stratégies peuvent cohabiter pour sécuriser ces systèmes.

Vérification statique : ces différents problèmes peuvent être partiellement résolus par une analyse statique à la volée lors du chargement du code mobile. Les analyses ne sont pas toujours très précises, c'est toujours un équilibre entre la complexité de l'analyse et le temps et/ou l'espace disponible. Ill peut être intéressant d'étudier comment insérer du code vérifiable dynamiquement pour palier une analyse trop simple. Nous avons vu que des systèmes de type peuvent être une solution, pour ces problèmes. Ils sont d'ailleurs à l'origine du mécanisme du PCC (la preuve accompagnant l'application). Ceci peut être une continuation de nos travaux sur le vérifieur à la PCC en étudiant comment modéliser de telles propriétés et construire l'algorithme de vérification correspondant.

4.2 Technique de preuve et de test

Lors du développement de notre vérifieur GemClassifier nous avons vu que les activités les plus coûteuses étaient liées à la preuve interactive. L'interface graphique de l'outil utilisé est très stalinienne et nécessite beaucoup d'effort de la part de l'utilisateur pour comprendre le lemme à démontrer. Il faut soit améliorer l'interface comme nous l'avons fait dans Jack soit remplacer l'activité de preuve interactive par une activité de test structurel. Les travaux sur la recherche de contre exemples présentés par [Mik-02] peuvent aussi être une piste intéressante. Une obligation de

preuve non prouvée automatiquement est soit le fait d'une erreur dans la spécification soit parce qu'il manque des hypothèses pour que prouveur réussisse. Dans ces travaux, les auteurs utilisent un vérifieur de modèle sur un ensemble fini de la spécification afin de rechercher dans la spécification une contradiction. La trace du contre exemple peut être suffisamment précise pour découvrir les améliorations nécessaires à apporter à la spécification.

Nous avons vu en conclusion de nos travaux sur le test d'application à partir d'UML que les modèles créés à partir des vues Statechart par exemple pour générer l'oracle pouvaient être remplacés par une inclusion d'assertion dans le code. Il faudrait étudier les liaisons possibles entre UML et JML et surtout voir comment le solveur de contrainte de CASTING fonctionnant à partir des pré et post conditions pourrait s'adapter dans ce contexte afin de générer des cas de tests. Un autre axe de recherche est de mieux comprendre les capacités d'abstraction du langage JML.

4.3 La certification un vrai vecteur ou mirage ?

La certification pour les niveaux élevés a longtemps été vue comme étant un vecteur important pour la diffusion des techniques formelles. Or aujourd'hui l'état du marché ne suscite plus cet engouement alors que les problèmes rencontrés initialement n'ont pas tous trouvé de solutions. En particulier dans le domaine des classes de test (ATE) des critères communs nous sommes bien loin d'obtenir les taux de couverture requis. Les techniques proposées ici dans le cadre de la génération à partir de modèle pourraient apporter une solution. Les analyses de vulnérabilité et de canaux cachés nécessitent des techniques différentes et donc des modèles différents. Tous ces modèles ont un coût et un axe de recherche pourrait à partir d'un modèle initial dériver des vues différentes (ou translater dans une sémantique différente) afin d'effectuer des analyses particulières. Se pose dès lors le problème de la cohérence entre ces vues de mises à jour, etc.

Cependant, il est possible d'avoir des certifications plus légère en dehors des cadres de ces organismes pour effectuer une certification légère. De nombreux organismes proposent de vérifier des règles génériques de sécurité sur les applications carte à puce comme par exemple s'assurer systématiquement que l'émetteur de la commande est authentifié ou bien que les droits aient été présentés avant d'exécuter un changement dans le cycle de vie de l'applet. Beaucoup de ces règles peuvent se coder simplement et automatiquement au moyen d'assertions JML. Dès lors que ces assertions sont incluses dans le code il faut rajouter la spécification minimale pour pouvoir prouver les lemmes générés. Cette alternative peut remplacer avantageusement l'audit manuel de code de par son automatisation et le degré de confiance que l'on peut obtenir dans la preuve. Certaines propriétés pourront s'exprimer à l'aide de logique temporelle et JML ne serait plus alors le candidat idéal pour ce domaine sauf à étendre le langage par des modalités comme on le voit apparaître dans le langage B.

Conclusions

Les travaux décrits dans ce document ont souvent proposé des solutions pragmatiques à des problèmes rencontrés dans la conception, l'utilisation et/ou le déploiement de solutions à base de carte à puce. Les différents laboratoires de recherche auxquels j'ai eu la chance de participer n'ont pas en effet de vocation de résoudre des problèmes théoriques mais bien de mettre en pratique des technologies issues du monde académique. Notre vocation est de diminuer le risque technologique par le prototypage et l'évaluation de telles technologies. Et en ce sens, nous avons parfois réussi de belle manière en proposant par exemple un modèle générique pour la spécification d'un vérifieur de byte code ou bien la première implémentation dans une carte à puce d'un tel vérifieur spécifié formellement.

D'autres travaux sont restés dans l'ombre et ne méritaient pas publication car plus proches du développement de produits. Nous avons mené plusieurs travaux sur la correction de protocoles, soit de manière préventive (GTV100, TCP/IP, T=CL) soit de manière corrective (ordonnanceur réparti, T=1). Ces modélisations utilisent des langages de description formelle et des vérifieurs de modèles. Nous citerons plus spécifiquement le travail de modélisation de la norme Digital Video Broadcasting en cours de normalisation lors de ce travail (1997) dont Gemplus proposait alors une implémentation : le module GTV 100. Le risque était de mettre en production un système dont la norme pouvait évoluer suite à une erreur. Il nous a donc été demandé de s'assurer de la correction de la spécification principalement lors de défaillance. Nous avons vérifié le comportement d'un système composé d'un serveur (la station hôte) et de deux clients (des modules) tant dans le mode de fonctionnement nominal qu'en présence de fautes pour une propriété donnée. Ce travail nous a parmi de s'assurer que les incomplétudes et erreurs de la norme n'affectaient pas le module en cours de développement.

Les axes de recherche que nous avons développés ici ont souvent fait l'objet de stage de DEA ou de DESS et aussi de deux thèses soutenues. Ceci fut pour nous l'occasion de montrer que les techniques rigoureuses de développement ne sont pas qu'une chimère scolastique mais sont une réalité dans une niche l'industrielle.

Références

[Abr-98] J.-R. Abrial, L.Mussat, *Specification and Design of a Transmission Protocol by Successive Refinements Using B*, in Mathematical Methods in Program Development, Edited by Broy and Schieder, Springer Verlag, 1998

[Avi-01] A. Avizienis, J.C. Laprie, B. Randell, *Fundamental concepts of dependability*, Rapport Laas n°1145, Avril 2001.

[Ban-94] J. Banâtre, C. Bryce, D. Le Metayer, *Compile-time detection of information flow in sequential programs*, LNCS n°875, pp.55-73, 1994.

[Bar-01] G. Barthe, G. Dufay, L. Jakubiec, S. Melo De Sousa, *A Formal Executable Semantics of the Java Platform*, In Proc. of ESOP'01, LNCS n°2028, pp. 302-319, Genoa, April 2001.

[Bar-02] G. Barthe, G. Dufay, L. Jakubiec, S. Melo De Sousa, *A Formal Correspondence between Offensive and Defensive Java Card Virtual Machine*, In Proc. of VMCAI'02, LNS n°2294, pp. 32-45, Venice, January 2002.

[Ber-02] C. Bernardeschi, N. De Francesco, G. Lettieri *Using Standard Verifier to Check Secure Information Flow in Java Bytecode* COMPSAC'02, vol. 1, Oxford 2002.

[Bic-96a] T. Bickard, J.-L. Lanet, *Apport des calculateurs répartis dans la régulation des turbomoteurs*, Real Time Systems & Embedded Systems'96, pp 423-431, January 96, Paris.

[Bic-96b] T. Bickard, N. Hubart, J.-L. Lanet, *Jet Engine Control Systems*, IEEE International Workshop on Embedded Fault Tolerant Systems, September 1996, Dallas, Texas.

[Bie-92] P.Bieber, F. Cuppens. A Logical View of Secure Dependencies, Journal of Computer Security, 1 , pp. 99-129, 1992.

[Bie-00] P.Bieber, J. Cazin, El. Marouani, P. Girard, J.-L. Lanet, V. Wiels, G. Zanon, *The PACAP prototype: a tool for detecting Java Card illegal flow,* in Java on Smart Card 2000, LNCS 2041, pp. 25-37, Cannes Sept. 2000

[Bie-02] P.Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, G. Zanon. *Checking Secure Interactions of Smart Card Applets*, Journal of Computer Security, Vol.10, N° 4, pp.369-398, 2002.

[Bon-01] P. Bontron, O. Maury, L. Du Bousquet, Y. Ledru, C. Oriat, M.-L. Potet, *TOBIAS : un environnement pour la création d'objectifs de tests à partir de schéma de tests*. In ICSSEA'01, December 2001.

[Bör-98] E. Börger, W. Schulte, *Defining the Java virtual machine as platform for provably correct Java compilation*. In L. Brim, J.Gruska, and J. Zlatuska editors, 23rd Int.

Symp. Mathematical Foundations of Computer Science (MFCS) LNCS 1450, pp. 17-35, Brno, Czech Republic, 1998.

[Bör-99] E. Börger, W. Schulte, Initialization problems for java, Software Concepts and Tools, n°20 (4), pp. 175-179, 1999

[Bos-00] G. Bossu, A. Requet, *Embedding Formally Proved Code in a Smart Card : Converting B to C*, In Proc. of ICFEM 2000, pp. 15-22, York, September 2000.

[Bou-00a] L. Du Bousquet, J.-L. Lanet, H. Martin, *An integrated approach for Java Card applets validation*, Internal Report, Gemplus Research Labs, April 2000.

[Bou-00b] L. Du Bousquet, H. Martin, *Automatic Test Generation for Java Card Applets,* in Java on Smart Card 2000, LNCS 2041, pp. 25-37, Cannes September 2000.

[Bou-01] L. Du Bousquet, H. Martin, J.-M. Jézéquel, *Conformance Testing from UML specification*, In UML Workshop, LNI n°7, pp. 43-56, Toronto, Canada, 2001.

[Bou-03] L. Du Bousquet, J.-L. Lanet, H. Martin, *An Integrated Approach for Java Card Applet Validation*, Submitted to ASE'03, Montréal, Canada, 2003.

[Bur-02a] L. Burdy, L. Casset, A. Requet, *Formal Development of an Embedded Verifier for Java Byte Code,* In Proc. DSN 2002, IEEE, pp. 51-56, Washington, June 2002

[Bur-02b] L. Burdy, A. Requet, *JACK : Java Applet Correctness Kit*, in Gemplus Developer Conference 2002, Singapore.

[Cas-99] L. Casset, J.-L. Lanet, *A Formal Specification of the Java Byte Code Semantics using the B method*, Proceedings of the ECOOP'99 workshop on Formal Techniques for Java Programs, Lisbon, June 1999.

[Cas-00] L. Casset, G. Grimaud, A. Requet, *Application of the B Formal Method to the Proof of a Type Verification Algorithm*, Hase 2000, Albuquerque, NM, November 2000.

[Cas-01] L.Casset L., *Formal Implementation of a Verification Algorithm Using the B Method*, Proceedings of AFADL01, Nancy, France, June 2001

[Cas-02a] L. Casset, J.-L. Lanet, *Increasing smart card dependability,* SIGOPS EW 2002, pp.209-212, Saint Emillion, Sept-02

[Cas-02b] L. Casset, *Construction correcte de logiciels pour carte à puce*, Thèse de l'Université de Marseille, Octobre 2002.

[Cas-02c] L. Casset, J.-L. Lanet, *D9: Smart Card Case Study Analysis*, Matisse Project Report V1.1, http://www.matisse.qinetiq.com/

[Cas-02d] L. Casset, *Development of an Embedded Verifier for Java Byte Code using Formal Methods,* In Proc. FME 2002, Copenhagen, LNCS n°2391, pp. 290-309, July 2002.

[Cas-02e] L. Casset, D. Deville, J.-L. Lanet, *On Card byte code verification, the ultimate step*, Java One 2002, Technical session, San Francisco, March 2002.

[CC-99] *Common Criteria for Information Technology Security Evaluation*, Part 1: Introduction and general model. V2.1, August 1999. http://www.commoncriteria.org/cc/cc.html.

[Cif-94] C. Cifuentes, *Reverse Compilation Techniques*. PhD Thesis, Queensland University of Technology. 1994.

[Coh-97] Cohen, R., The Defensive Virtual Machine Specification Version 0.5, [http://www.cli.com/software/djvm]

[Den-76] D.E Denning, *A lattice model of secure information flow, Communicaton of the ACM*, vol. 19, pp.236-243, May 1976.

[Dub-99] L. Du Bousquet, F. Ouabdesselam, J.-L. Richier. *Expressing and Implementing Operational Profiles for Reactive Software Validation*. 21st International Conference on Software Engineering (ICSE), ACM, mai 1999.

[Dic-93] J Dick, A. Faivre. *Automating the generation and sequencing of test case from model-based specifications*. FME'93: Industrial Strength Formal Methods, Springer-Verlag, 1993.

[Fou-01] L. Fournerie, *Multi-threaded programs and security: an outline approach,* Internal Report, Gemplus Research Labs, August 2001.

[Fre-98] S. Freud, J. Mitchell, *A type system for object initialization in the Java byte code language*. In Proc. of OOPSLA 98, 1998.

[Fre-99] S. Freud, J. Mitchell, *A Formal Framework for the Java Byte Code Language and Verifier*, In OOPSLA'99, pp. 147-166, ACM, Denver, December 1999.

[Gir-99] P. Girard, J.-L. Lanet, *New Security Issues Raised by Open Cards,* Elsevier Information Security, vol. 4 n°2, pp. 19-27, June 1999.

[Gog-84] J. Goguen, J. Messeguer, *Unwinding and Inference Control*, Proc. IEEE Symp. Security and Privacy, IEEE Computer Soc., pp. 75-86, 1984

[Gri-99] G. Grimaud, J.-L. Lanet, J.-J. Vandewalle, *FACADE: a Typed Intermédiate Language dedicated to Smart Card*, In Software Engineering ESEC/FSE 99, LNCS n°1687, pp. 476-493, Toulouse 1999.

[Gri-00] G. Grimaud, *Camille: un système d'exploitation ouvert pour carte à microprocesseur*, Thèse de l'université de Lille, décembre 2000.

[Hub-96] N. Hubart, J.-L. Lanet, *A New Fault-Tolerant Distributed Turbojet Enegine Control Computer and its debugging Tool*, CESA'96, pp. 659-663, July 1996, Lille.

[Jer-99] T. Jéron, P. Morel, *Test generation derived from model-checking,* In Computer Aided Verification, LNCS n°1633, 1999

[Jez-98] J.-M. Jézéquel, A. Le Guennec, F. Pennaneac'h. *Validating Distributed Software Modeled with UML*. <<UML>>'98 Beyond the Notation, page 331-340, 1998.

[Joy-03] M. Joye, *Elliptic curves and side-channel analysis*, ST Journal of System Research 4 (1), pp.17-21, February 2003.

[Kle-00] G. Klein, T. Nipkow, *Verified Lightweight Byte Code Verification*, in ECOOP'00, Workshop on formal techniques for Java Programs, pp. 35-42, Cannes, June 2000.

[Lan-92] J.-L. Lanet, *Définition d'un automate distribué pour la régulation d'un turbo réacteur*, Mémoire d'ingénieur CNAM, Cedric, Paris Avril 1992.

[Lan-95a] J.-L. Lanet, *Allocation de tâches dans un système temps réel,* in Proceeding of Real Time Systems'95, pp. 222-231, January 95, Paris.

[Lan-95b] J.-L. Lanet, *Task Allocation in a Hard Real Time Distributed System*. Real Time Systems'95, pp. 244-252, Sklarska Poreba, Poland, Sept.-95.

[Lan-95c] J.-L. Lanet, *Placement statique de tâches dans un calculateur réparti de régulation moteur.* Thèse de l'université Pierre et Marie Curie, 1995

[Lan-96] J.-L. Lanet, *A load Balancing Task Allocation Scheme in a hard Real Time System*, Euro-Par'96, pp. 640-643, August 96, Lyon

[Lan-97] J.-L. Lanet, *Modélisation du protocole DVB*, Internal Report, Gemplus Research Labs, April 97.

[Lan-98a] J.-L. Lanet, P. Lartigue, *The Use of Formal Methods for Smart Cards, a comparison between B and SDL to Model the T=1 protocol*, International Workshop on Comparing Systems Specification Techniques, pp. 3-16, Nantes March 26-27, 1998.

[Lan-98b] J.-L. Lanet, *Using the B Method for Modelling Protocols*, AFADL 98, pp.79-90, Ensma-Lisi, September 98, Futuroscope.

[Lan-00] J.-L. Lanet, *Are smart card the ideal domain for applying formal methods,* Invited speaker**,** ZB 2000, LNCS 1878, pp. 363-373, York.

[Lan-01] J.-L. Lanet, *Cartes à puce et méthodes formelles, une lente intégration…,* TSI, Vol. 20 n°7/2001, pp. 959-964.

[Lan-02a] J.-L. Lanet, *GemClassifier, a formally developed smart card*, Invited speaker, 2nd HCSS Conference, pp. 17-23, Baltimore Maryland, USA, March, 2002.

[Lan-02b] J.-L. Lanet, *The Use of B for Smart Card*, FDL'02, proceeding vol.2 September 24-27, 2002.

[McL-92] J. McLean, Proving Noninterference and Functional Correctness Using Traces, Journal of Computer Security, Vol. 1, pp. 37-57, 1992.

[Mar-01] H. Martin*, Une méthodologie de génération automatique de suites de tests pour applet Java Card*, Thèse de l'université de Lille, Mars 2001

[Mel-03] S. Melo De Sousa, *Outils et techniques pour la vérification formelle de la plateforme JavaCard*, Thèse Inria, février 2003.

[Mik-02] L. Mikhailov, M. Butler, *An Approach to Combining B and alloy*, In Proc. of ZB 2002, LNCS n°2272, pp. 140-161, Grenoble, January 2002.

[Mye-97] A. Myer, B. Liskov, *A decentralised model for information flow control*. In Proc. of the 16th ACM symposium on operating systems principles, 1997.

[Nec-97] G. Necula, P. Lee, *Proof Carrying Code*, in 24th ACM Sigplan Symposium on Principles of Programming Language, pp.106-119, Paris, 1997.

[Nip-01] T. Nipkow, *Verified Byte Code Verifier*, In Proc. of FOSSACS'01, LNCS n°2030, pp. 347-363, 2001.

[Pus-98] C. Push, *Proving the Soundness of the Java Byte Code Verifier in Isabel/HOL*, In Formal Underpinning of Java, OOPSLA'98 Workshop, Vancouver, October 1998.

[Qia-98] Z. Qian, *A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines*, In Formal Syntax and Semantics of Java LNCS n°1523, pp.271-312, 1999

[Qia-00] Z. Qian, A. Coglio, A. Golberg, *Towards a Provably Correct Implementation of the JVM Byte Code Verifier*, In Proc. DISCEX'00, Vol. 2, pp. 404-410, 2000.

[Ros-98] E. Rose, K. Rose, *Lightweight Byte Code Verification*, In Formal Underpinning of Java, OOPSLA'98 Workshop, Vancouver, October 1998.

[Req-03] Requet A., *A B model for ensuring soundness of the Java Card virtual machine*, Science of Computer Programming, 46 (3), pp. 283-306, 2003

[Sch-02] S. Schneider, H. Treharne, *Communicating B Machines*, In Proc. of ZB 2002, LNCS n°2272, pp. 416-435, Grenoble, January 2002.

[Smi-97] G.Smith, D.Volpano, A type-based approach to program security, In TAPSOFT'97, LNCS vol. 1214, pp. 607-621, April 1997.

[Smi-98] G. Smith, D. Volpano, *Secure information flow in a multi threaded imperative language*, In Proc. of the 25th POPL, pp. 355-364, San Diego, January 1998.

[Sta-98] R. Stata, M. Abadi, *A type system for Java byte code subroutines*. In Proc. 25th POPL, pp. 149,-160, San Diego, January 1998.

[Sta-01] R Stärk., J. Schmid, E. Börger, *Java and the Java virtual Machine: Definition, Verification and Validation,* Springer Verlag, Berlin 2001.

[Sto-01] J Strooter More, R. Krug, H. Liu, G. Porter, *Formal Models of Java at the Java Virtual Machine Level – A Survey from the ACL2 Perspective*, In Proceeding of Formal Techniques for Java Programs, S. Drossopoulou editor, 2001.

[Van-98] L. Van Aertryck. *Une méthode et un outil pour l'aide à la génération de jeux de tests de logiciels*. Thèse de doctorat, Université de Rennes 1, janvier 1998.

Bibliographie annexe

Anton J., McDonald J., SPECWARE, *Producing Software Correct by Construction*, Technical Report KES.U.01.3., Kestrel Institute, Palo Alto, California*, USA*, March 2001.

Coglio A., Goldberg A., Qian Z., *Towards a Provably-correct Implementation of the JVM Bytecode Verifier*, In Proc. DARPA Information Survivability Conference and Exposition (DISCEX'00*)*, Vol. 2, pages 403-410, IEEE Computer Society, 2000.

Barbey S., *Test selection for Specification-Based Unit Testing of Object-Oriented Software based on Formal Specifications*. Thèse de doctorat, Ecole Polytechnique Fédérale de Lausanne,1997.

Basin D. Friedrichs S., Posegga J., Vogt H., *Java Byte Code Verification using Model Checking*. In R. Alur and T. Henzinger, editors*.* 11th Int. Conf. On Computer Aided Verification (CAV), LNCS 1633, pp.491-494, 1999.

Behnia S., *Test de modèles formels en B : cadre théorique et critères de couverture*. Thèse de doctorat, LAAS CNRS, 27 octobre 2000.

Binder R.-V., *Testing object-oriented systems – Models, Patterns, and Tools.* Addison Wesley, ISBN 0-201-80938-9, octobre 1999.

Bernot G., Gaudel M.-C., Marre B., *Software testing based on formal specifications: A theory and a tool.* Software Engineering Journal, page 387-405, novembre 1991.

Börger E., Schulte W., *Modular design for the Java Virtual Machine Architecture*. In E. Börger editor, Architecture Design and Validation Methods pp. 297-357, 2000.

Burdy L., Cheon Y., Cok D., Ernst M., Kiniry J., Leavens G. T., Leino K. R. M., Poll E., *An overview of JML tools and applications*, *FMICS'03, 2003; Electronic Notes in Theoretical Computer Science*, Vol. 80.

Corbett J.C., Dwyer M.B., Hatcliff J., Laubach S., Pasareanu C.S., Robby and Zheng H., *Bandera :extracting finite-state models from Java source code*. In 22nd Int. Conf. On Software Engineering, pp. 439-448, Limerick, 2000.

Doyon S., *On the security of Java : The Java Bytecode Verifier*, Mémoire de l'Université Laval, Avril 1999.

Fernandez J.-C., Jard C., Jéron T., Viho C., *Using On-the-fly Verification Techniques for the Generation of Test Suites*. In R. Alur and T.A. Henzinger, editors, Conference on Computer Aided Verification (CAV'96)*,* New Brunswick, New Jersey, USA, LNCS 1102. Springer, August 1996.

Gaudel M.-C., *Testing can be formal too*. Editeurs Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, TAPSOFT'95 : Theory and Practice of Software Development, Volume 915 de LNCS, page 82-96, Aarhus, Denmark, éditions Springer Verlag, 1995.

Hartel P.H., Butler M. J., Levy M., *The operational semantics of a Java secure processor*. In J. Alves-Foss editor, Formal Syntax and Semantics of Java, LNCS 1523, pp.313-352, Springer Verlag, 1999.

Jensen T. Le Metayer D., Thorn T., *Verification of control flow based security properties*. In Symposium on security and privacy, pp. 89-103, Oakland may, 1999

Leroy X., *Java Byte Code Verification: An Overview*, Proceedings of Computer Aided Verification, CAV'2001, LNCS 2102, pp 265-285, Springer-Verlag, 2001.

Myers A. C., Liskov B., *A Decentralized Model for Information Flow Control*, Proceeding of the 16th ACM Symposium on Operating System Principles, Saint- Malo, France, October 1997.

Nipkow T., *Verified Byte code Verifiers,* Fakultät für Informatik, Technische Universität München, 2000. http://www.in.tum.de/~nipkow

Phalippou M., *Relation d'implantation et hypothèses de test sur des automates à entrées et sorties*. Thèse de doctorat, université de Bordeaux 1, septembre 1994.

Posegga J., Vogt H., *Offline verification for java byte code using a model checker*. In Proc. of Esorics, LNCS n°1485, 1998.

Pusch C., *Proving the Soundness of a Java Bytecode Verifier in Isabelle/HOL*, in Formal Underpinnings of Java, OOPSLA'98 Workshop, Vancouver, October. 1998.

Qian Z., *A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines*. In Jim Alves-Foss, editor, Formal Syntax and Semantics of Java, volume 1523 of Lecture Notes in Computer Science, pages 271-312. Springer, 1999.

Shellhorn G., Reif W., Schairer A., Karger P., Austel V., Toll D., *Verification of a formal security model for multiapplicative smart card*s. In proceedings of ESORICS, LNCS n°1895, 2000

Schneider F. B., *Enforceable Security Policies*, Department of Computer Science, Cornell University, USA, 1999.