Développement formel d'un vérifieur embarqué de byte-code Java

[Published in *Technique et Science Informatiques (TSI)* **22**, 2003.]

Lilian Burdy, Ludovic Casset, and Antoine Requet

Gemplus Software Research Labs
BP 100, 13881 Gémenos Cedex
{lilian.burdy, ludovic.casset, antoine.requet}@gemplus.com

Résumé. La politique de sécurité de Java est implémentée par des composants de sécurité tels que la machine virtuelle, le vérifieur ou encore le chargeur. Il est de première importance d'assurer que les implémentations de ces composants sont cohérentes vis-à-vis de leurs spécifications. Les méthodes formelles peuvent être utilisées pour apporter la preuve que les implémentations de ces composants sont conformes à leurs spécifications. Dans cet article, nous présentons le développement formel, à l'aide de la méthode B, d'un vérifieur de byte-code Java Card. Le langage complet est traité, ce qui nous fournit des métriques réalistes concernant le coût d'un développement formel. Cette formalisation se termine par l'intégration du code issu du développement dans une carte à puce.

Abstract. The Java security policy is implemented by security components such as the Java Virtual Machine (JVM), the verifier and the loader. It is of prime importance to ensure that the implementation of these components is in accordance with their specifications. Formal methods can be used to bring the mathematical proof that the implementation of these components corresponds to their specification. In this paper, a formal development is performed on the Java Card byte code verifier using the B method. The whole Java Card language is taken into account in order to provide realistic metrics on formal development. This formalization leads to an embeddable implementation of the byte code verifier. We present the formal models, discuss the integration into the card and the results of such an experiment.

Mots-clés. Algorithme de vérification, méthode B, vérifieur de byte-code, méthodes formelles, cartes à puce.

Keywords. Verification algorithm, B method, byte code verifier, formal methods, smart card.

1 Introduction

Les cartes à puce sont souvent considérées comme un élément permettant d'assurer la sécurité d'un système d'information. Ces cartes dotées d'un processeur de différents types de mémoire, protègent les éléments secrets qu'elles contiennent (qu'il s'agisse de données ou d'applications). La sécurité des cartes à puce découle de leur conception : la totalité d'un système carte à puce est contenu dans un unique bloc physique contenant le microprocesseur, la mémoire, les données et les applications, le tout dans 25 mm^2.

Les cartes ouvertes ou *Open Cards* introduisent la possibilité de charger du code exécutable après la délivrance de la carte. Cette possibilité permet à la carte d'offrir de nouvelles fonctionnalités et de proposer de nouvelles applications. Dans ce contexte, il n'y a a priori 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 menacent pas la sécurité de la carte en tant que plate-forme d'exécution. On se doit donc d'assurer que leur exécution ne risque 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 entre plusieurs composants : l'interpréteur, le vérifieur, les bibliothèques 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. Pour 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. C'est donc dans ce contexte que les méthodes formelles ont été utilisées, le but étant d'apporter une preuve mathématique de la conformité de l'implémentation d'un vérifieur vis-à-vis de sa spécification. Notre objectif était de formaliser le vérifieur de byte-code pour le langage Java Card complet (hormis les instructions jsr et ret qui sont traitées séparément), et de montrer qu'une implémentation générée à partir de cette formalisation respecte les contraintes de la carte à puce.

Cet article présente les résultats de l'un des cas d'étude du projet européen Matisse.[1] L'objectif de ce projet est de fournir des méthodologies, des outils et des techniques permettant l'utilisation de méthodes formelles dans un contexte industriel. Le cas d'étude auquel nous avons contribué concerne la spécification formelle et le développement d'un vérifieur de byte-code Java Card.

Le reste de cet article est organisé comme suit : La section 2 décrit les principes de la vérification de byte-code. La section 3 détaille le modèle formel du vérifieur. L'intégration du développement formel avec les éléments développés

[1] Projet européen IST MATISSE numéro *IST-1999-11435*

classiquement est discutée dans la section 4. La section 5 présente des métriques sur le développement et la section 6 conclut.

2 La vérification de byte-code

La vérification de byte-code a pour objectif de s'assurer du respect de contraintes statiques sur le byte-code chargé. Ces contraintes assurent que le byte-code peut être exécuté sans risque par la machine virtuelle et ne peut pas outrepasser les mécanismes de sécurité de haut niveau. La vérification de byte-code est décrite informellement par Lindholm (Lindholm *et al.*, 1996). Elle consiste à effectuer une analyse statique du code mobile chargé, également appelé applet dans le cadre du langage Java. Cette analyse assure que le fichier contenant l'applet est un fichier valide et qu'à l'exécution il n'y aura pas de débordement de pile, que le flot d'exécution reste confiné sur du byte-code valide, que chaque argument d'une instruction est d'un type correct et que les appels de méthodes sont effectués conformément à leurs attributs de visibilité (`public`, `protégé`, `privé`).

Le premier point correspond à de la vérification structurelle alors que les suivants concernent la vérification du typage. Les sous-sections suivantes décrivent plus en détails les propriétés assurées par ces vérifications.

2.1 La vérification structurelle

La vérification structurelle consiste à s'assurer que le fichier chargé est un fichier valide. En fait, cela permet de garantir que le fichier contient bien la description de classes Java comprenant notamment du byte-code interprétable, et que les informations qu'il contient sont cohérentes entre elles. Par exemple, le vérifieur s'assure que toutes les structures sont de la taille appropriée et que les parties référencées existent réellement. Ces tests ont pour objectif de s'assurer que le fichier chargé ne peut pas être mal interprété par le vérifieur de type ou la machine virtuelle.

En dehors des tests structuraux purement dédiés à la vérification du format binaire, d'autres tests, plus en relation avec le contenu du fichier, sont effectués. Ces tests assurent par exemple qu'il n'y a pas de cycle dans la hiérarchie d'héritage des classes, ou que les méthodes finales ne sont pas surchargées.

Dans le cas de Java Card, les tests structuraux sont plus importants que dans le cas de Java. Cela vient du fait que le format du fichier CAP[2] utilisé pour stocker les paquetages Java Card a été conçu pour une installation simple et une édition de liens simplifiée (Sun, 2000). Par exemple, la majorité des références vers d'autres composants est donnée sous la forme de décalages dans le composant. La vérification structurelle s'assure que ces décalages sont confinés au composant.

[2] CAP pour Converted APplet, est le nom du format des fichiers chargés dans les cartes à puce Java Card.

Un fichier au format CAP, spécifié par Sun, est constitué de onze composants qui contiennent chacun des informations spécifiques du paquetage Java Card. Par exemple, le composant *Method* contient le byte-code de chaque méthode et le composant *Class* contient les informations sur les classes comme les références vers la superclasse et les méthodes déclarées. De plus, l'algorithme de vérification que nous utilisons nécessite la présence d'un composant supplémentaire qui est ajouté grâce au mécanisme d'extension de Java Card. Ainsi, avec ce composant additionnel, nommé *Proof component*, nous facilitons le processus de vérification tout en restant compatible avec la spécification Java Card. Nous avons donc douze composants à vérifier pour nous assurer de la validité structurelle du fichier.

Dans le cas de Java Card, nous distinguons les vérifications structurelles internes et externes. Les vérifications internes sont celles qui peuvent être effectuées sur un composant. Un exemple de ces vérifications est de vérifier que le composant *Class* est ordonné selon la hiérarchie des classes. Les vérifications structurelles externes correspondent aux tests assurant la cohérence entre les différents composants du paquetage ou avec les autres paquetages. Par exemple, un de ces tests consiste à vérifier que les méthodes déclarées dans le composant *Class* correspondent à des méthodes existantes dans le composant *Method*.

2.2 La vérification du typage

Cette vérification est effectuée méthode par méthode et doit être faite pour chaque méthode du paquetage, c'est-à-dire pour chaque méthode contenue dans le composant *Method* du fichier CAP à vérifier. Une description générale de cette vérification, mettant en évidence les points délicats, est donnée dans (Leroy, 2001).

La partie vérification du typage assure qu'aucune conversion de type interdite d'après les règles de typage du langage Java Card n'est effectuée par le programme. Par exemple, un entier ne peut pas être converti en référence sur un objet, les conversions de type ne peuvent être effectuées qu'en ayant recours à l'instruction `checkcast` qui s'assure que le changement peut effectivement être effectué. De même, les arguments passés en paramètre à une méthode doivent être de types compatibles avec ceux attendus par la méthode.

Comme les types des variables locales ne sont pas explicitement stockés dans le byte-code, il est nécessaire de retrouver le type de ces variables en analysant le byte-code. Cette partie de la vérification est la plus compliquée, et la plus coûteuse à la fois en temps et en mémoire. En effet, cela nécessite de calculer le type de chaque variable et de chaque élément dans la pile pour chaque instruction et chaque chemin d'exécution possible.

Dans le but de rendre une telle vérification possible, la spécification fournie par Sun est assez restrictive sur les programmes qui sont acceptés. Seuls les programmes dont le type de chaque élément de la pile et de chaque variable locale est le même quelque soit le chemin pris pour atteindre cette instruction sont acceptés. Cela nécessite en particulier que la taille de la pile à une instruction donnée soit la même quel que soit le chemin menant à cette instruction.

```
static void m(boolean b) {
        if (b) {
                int i = 1;
        } else {
                Object tmp = new Object();
        }
        int j = 2;
}
```

Fig. 1. Un exemple de méthode Java

La figure 1 montre un exemple de méthode Java. Les instructions correspondant à cette méthode ainsi que les types inférés par le vérifieur sont donnés à la figure 2. Afin de ne pas alourdir le schéma, la figure 2 présente une version simplifiée ne prenant pas en compte l'initialisation des objets.

Les instructions du byte-code travaillent sur des variables locales numérotées (ici, une seule variable, $\mathtt{v}_0$) et une pile d'exécution. Dans cet exemple, la variable v_0 du byte-code correspond au paramètre de la fonction (le booléen `b`). La méthode définit d'autres variables locales, cependant comme celles-ci ne sont pas utilisées en même temps, le compilateur peut ne déclarer qu'une seule variable dans le byte-code.

Pour chaque instruction, la seconde colonne fournit le type inféré de cette variable locale, ainsi que des éléments présents sur la pile d'exécution. En entrée de la méthode, la variable $\mathtt{v}_0$ contient une valeur de type `int` correspondant au paramètre booléen `b`. Ensuite, selon la branche du `if` considérée, cette variable contient un entier ou une référence sur un objet. Le label `endif` étant accessible depuis les deux branches du `if`, il est nécessaire d'unifier les types de la variable $\mathtt{v}_0$ pour ces deux branches. Dans ce cas, le plus petit surtype commun des entiers et des références correspond à la borne supérieure du semi-treillis utilisé : top.

2.3 Etudes formelles sur la vérification de byte-code

De nombreux travaux ont été conduit sur la vérification de byte-code Java. La plupart de ces travaux ne s'intéressent qu'à la formalisation de la vérification de type, occultant la vérification structurelle qui est pourtant partie intégrante de la vérification de byte-code Java.

Un des modèles formels les plus complets de la machine virtuelle Java est celui de Qian, (Qian, 1999). 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, (Coglio *et al.,* 2000), les auteurs fournissent également 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

	Types Inférés		
.method public static m(Z)V			
.limit stack 2	v_0	pile	
.limit locals 1			
iload_0	int		
ifeq else	int	Int	
iconst_1	int		
istore_0	int	int	
goto endif	int		
else:			
new java/lang/Object	int		
dup	int	Object	
astore_0	int	Object	Object
invokespecial java/lang/Object/<init>()V	Object	Object	
endif:			
iconst_2	top		
istore_0	top	int	
return	int		
.end method			

Fig. 2. Byte-code Java et informations de typage associées

spécifications et ensuite d'extraire le code correspondant au vérifieur en utilisant l'outil Specware.[3]

Dans le projet Bali, Push prouve une partie de la machine virtuelle Java avec le prouveur Isabelle/HOL[4], (Push, 1998). En utilisant les travaux de Qian, elle fournit les spécifications d'un vérifieur et prouve alors sa correction. Elle définit également un sous ensemble de Java, μjava, et prouve des propriétés sur ce sous ensemble, (Push *et al.*, 2000). Plus précisément, ils formalisent le système de typage et la sémantique de ce langage à l'aide du prouveur de théorème Isabelle. Dans un travail plus récent, Nipkow présente la spécification formelle du vérifieur de byte-code Java Card en Isabelle, (Nipkow, 2000). 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 (Rose *et al.*, 1998) a été prouvé correct grâce à sa formalisation et à sa preuve en Isabelle, (Klein *et al.*, 2000). Un algorithme similaire portant sur un langage dédié aux cartes à puce a également été prouvé correct grâce à une modélisation à l'aide de la méthode B, (Requet *et al.*, 2000).

Des travaux précédant celui décrit dans cet article ont également été conduits à l'aide de la méthode B, sur la formalisation d'une machine défensive, c'est-à-dire une machine virtuelle effectuant des tests dynamiques qui se raffine en

[3] Le site internet de l'outil Specware est : http ://www.kestrel.edu/home/techtransfer.html

[4] Le site internet du prouveur de théorème Isabelle, http ://www.cl.cam.ac.uk/Research/HVG/Isabelle/index.html

un vérifieur et un interpréteur, (Casset, 1999), et sur la formalisation puis l'implémentation d'un vérifieur de byte-code simplifié à douze instructions, (Casset, 2001). Un travail similaire sur le vérifieur simplifié a été effectué avec le prouveur de théorème Coq[5] par Bertot. Bertot prouve la correction de l'algorithme de vérification et génère une implémentation en utilisant le mécanisme d'extraction de Coq, (Bertot, 2000).

2.4 Adaptation à des appareils embarqués

Effectuer une vérification de byte-code complète nécessite une importante capacité de calcul et de mémoire. Différents systèmes et solutions sont proposés pour permettre la vérification sur des appareils fortement contraints comme les cartes à puce. Ces solutions se basent sur un pré-traitement hors de la carte de l'applet à vérifier. Comme la vérification de typage est la plus consommatrice de ressources, les solutions adaptées aux appareils embarqués ont pour but de réduire la complexité de l'inférence de type afin d'accélérer son traitement et de la rendre moins dispendieuse en ressources.

Deux approches sont proposées :
- la normalisation de byte-code,
- le proof-carrying code (PCC) ou des techniques apparentées.

Les sous-sections suivantes présentent ces différentes techniques. La technique du proof-carrying code est plus amplement décrite car c'est celle que nous avons choisie de développer pour notre vérifieur de type.

Normalisation de byte-code La normalisation de byte-code est l'approche proposée par le vérifieur pour cartes à puce de Trusted-Logic[6], (Leroy, 2001). Cela consiste à normaliser l'applet qui doit être vérifiée de telle manière qu'elle soit plus simple à vérifier. Plus exactement, l'applet est modifiée de telle façon que :
- chaque variable locale ait un et un seul type,
- la pile soit vide à chaque destination de saut.

Cela a pour conséquence de réduire grandement les besoins en mémoire puisque le vérifieur n'a pas besoin de garder les informations de typage pour chaque instruction, mais seulement pour chaque variable dans la méthode vérifiée. Les besoins en puissance de calcul sont également réduits, puisque seulement un calcul de point fixe simplifié doit être effectué. Cependant, comme le code même de l'applet est modifié, sa taille et par conséquent son occupation mémoire peuvent se trouver théoriquement modifiées.

Vérification de byte-code allégée Proposée et présentée par Necula en 1997, (Necula *et al.*, 1997), cette solution consiste à ajouter une preuve de la sûreté

[5] Le site internet de l'assistant de preuve Coq, http ://coq.inria.fr
[6] Le site internet de la société Trusted Logic : http ://www.trusted-logic.fr/

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.

Une adaptation de cette technique au langage Java a été proposé par Rose, (Rose, 1998). Cette adaptation est désormais utilisée par la KVM de Sun Microsystem, (Sun, 2000). Dans ce contexte spécifique, la preuve représente des informations additionnelles de typage correspondant au contenu des variables locales et des éléments de la pile pour chaque destination de sauts. La figure 3 décrit le contenu de la preuve pour l'exemple de la méthode de la figure 1. Notons que seules les informations de typages correspondant aux labels `else` et `endif` doivent être transmises. Ces informations de typage correspondent aux résultats du calcul de point fixe effectué par un vérifieur complet hors de la carte. Une fois ces informations envoyées avec le code de l'applet, le consommateur de code n'a plus qu'à vérifier la correspondance. Cette dernière vérification est une passe linéaire sur le code de l'applet qui vérifie la validité de chaque information de typage avec le code à vérifier.

	Preuve	
	v_0	pile
.method public static m(Z)V		
.limit stack 2		
.limit locals 1		
iload_0		
ifeq else		
iconst_1		
istore_0		
goto endif		
else:		
new java/lang/Object	int	
dup		
astore_0		
invokespecial		
java/lang/Object/<init>()V		
endif:		
iconst_2	top	
istore_0		
return		
.end method		

Fig. 3. Byte-code Java et sa preuve associée

Comparée à la normalisation de byte-code, la vérification allégée nécessite d'enlever les instructions `jsr` et `ret` des programmes. Ces instructions, utilisées pour l'implémentation de sous-routines internes à une méthode peuvent être supprimées par duplication de code, ou par l'utilisation d'instructions `goto` et `tableswitch` comme proposé par Freund (Freund, 1998). Ce choix de pré traitement correspond à la méthode employée pour la KVM de Sun.

La vérification allégée demande de plus un emplacement temporaire de stockage en mémoire (EEPROM) pour stocker les informations de typage ajoutées. Cependant, la vérification allégée effectue la vérification de type en une seule passe, linéaire, du code. Enfin, la preuve n'étant pas modifiée pendant la vérification, celle-ci peut-être stockée en mémoire EEPROM, libérant la mémoire RAM pour d'autres usages.

3 Modélisation d'un vérifieur de byte-code à l'aide de la méthode B

Dans cette section, nous présentons la manière dont a été modélisé le vérifieur de byte-code à l'aide la méthode B (Abrial, 1996). Nous nous intéresserons plus particulièrement à l'architecture des modèles B qu'il a fallu mettre en place mais aussi aux différents avantages qu'a pu apporter le développement formel de ce logiciel.

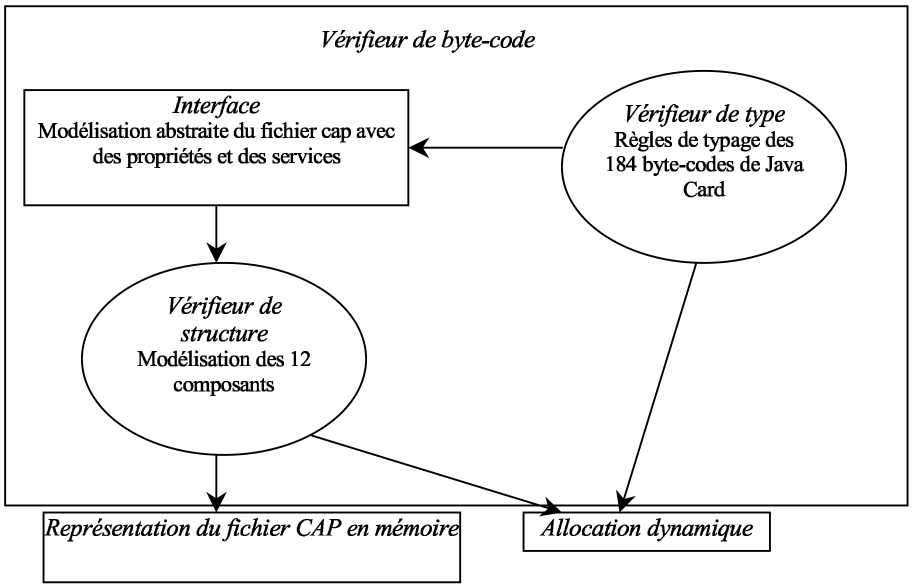

Fig. 4. Architecture générale du modèle B

Comme décrit dans la section précédente, 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 décorrelées,
- 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 avec un traitement particulier selon le type du byte-code.
- du point de vue de la modélisation, comme nous le verrons dans ce chapitre, les modélisations des deux vérifieurs diffèrent dans l'utilisation qui est faite 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 les propriétés que les services dont il a besoin (figure 4).

Le vérifieur de type est complètement modélisé en B sauf ce qui concerne les allocations mémoire. Pour les accès aux composants, une interface contenant les fonctionnalités nécessaires 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. 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 ont été modélisées. 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.

3.1 Le modèle du vérifieur de type

Le vérifieur de type 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 du 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, son byte-code est vérifié de façon linéaire grâce à la technique du proof-carrying code (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 est l'ensemble des composants B (machines, raffinements, implémentations) qui correspondent à la spécification du vérifieur. Il comprend les boucles de haut niveau et la spécification complète de chaque test à effectuer sur chaque byte-code. Le modèle concret est l'ensemble des composants B qui implémentent le modèle abstrait. Il s'appuie sur les services fournis par le vérifieur de structure et une machine de base modélisant la mémoire volatile de travail (RAM). En terme de validation, le modèle abstrait doit faire l'objet de revues pour vérifier sa conformité par rapport aux spécifications informelles, par contre le modèle concret est complètement validé par la preuve comme étant une implémentation correcte du modèle abstrait.

Modèle abstrait La machine de plus haut niveau du vérifieur de type qui définit son interface est très simple puisqu'elle propose seulement une opération qui retourne un booléen. Cette opération est implémentée par deux boucles imbriquées qui appellent une opération qui spécifie le test à réaliser en fonction du byte-code courant. Ceci donne l'architecture B de la figure 5.

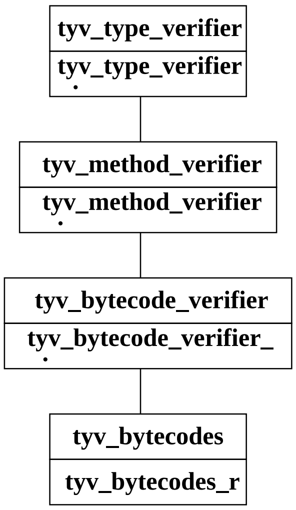

Fig. 5. Architecture du modèle abstrait du vérifieur de type

La machine *tyv_type_verifier* contient une seule opération renvoyant un booléen. Elle est implémentée par une boucle sur toutes les méthodes présentes dans le fichier à analyser. Cette boucle appelle une opération de *tyv_method_verifier* qui elle aussi renvoie un booléen. Cette opération est implémentée par une seconde boucle sur toutes les instructions de la méthode. Cette boucle appelle une opération de *tyv_bytecode_verifier*. Cette opération est implémentée par un traitement par cas suivant la nature de l'instruction traitée (il existe 184 types d'instruction en Java Card). Ce traitement par cas appelle une opération par cas possible. Ces opérations sont décrites de manière creuse dans *tyv_bytecodes* et de manière complète dans son raffinement. Ainsi ces huit composants comprenant quatre machines, trois implémentations et un raffinement forment le modèle abstrait du vérifieur. Cette partie de la spécification comporte peu de propriétés et la preuve permet principalement d'assurer que les boucles terminent et que le typage des variables est correct. Ce modèle est relativement important, le raffinement *tyv_bytecodes_r* fait à lui seul 5000 lignes de B réparties sur 90 opérations.

Modèle concret Le modèle concret implémente le modèle abstrait, et permet d'obtenir la preuve assurant la correction de l'implémentation. L'implémentation se réalise par plusieurs raffinements successifs. Les raffinements sont de deux types :

- le raffinement de structure : les structures abstraites telle que la clause SELECT doivent être converties en structures concrètes telle que la clause IF.
- Le raffinement de données : les données abstraites telles que les séquences doivent être converties en données concrètes tels que les entiers, ceci se fait en donnant un invariant de liaison, dit aussi de collage, qui a pour but de définir la variable abstraite en fonction de la variable concrète.

L'implémentation s'appuie sur des services permettant d'accéder aux informations présentes dans le fichier CAP. Ces services sont décrits dans une machine abstraite servant d'interface. Cette machine a pour intérêt de présenter les propriétés et les services suffisants pour définir et implémenter un vérifieur ; elle représente une première spécification du vérifieur de structure. La seule partie réalisée en machine de base est l'allocation dynamique et la libération de la pile de type. La taille de la pile n'étant pas connue statiquement et dépendant de la méthode analysée, il a fallu mettre en place un mécanisme d'allocation dynamique d'un tableau. Ce point est un peu particulier puisque généralement en B, les données concrètes sont typées statiquement.

Le reste de la construction du modèle concret est plutôt classique. Il est composé dans une première partie des raffinements de structure, avec le raffinement des clauses SELECT par des IF et l'introduction de boucles pour effectuer les tests de compatibilité des signatures ou des types de la pile. Ces raffinements se terminent sur des opérations simples de mise à jour de séquences (pour la pile) ou de fonctions partielles (pour les variables locales). Dans une seconde étape ces données abstraites sont elles-mêmes raffinées pour obtenir un collage final avec un pointeur en mémoire.

Ainsi dans le cadre du vérifieur de type, il nous a semblé très intéressant d'utiliser la méthode B puisqu'à partir d'un modèle abstrait, nous construisons un modèle concret comprenant plus de 25 machines dont la moitié d'implémentation. La preuve de ce modèle concret seule nous assure sa validité vis-à-vis du modèle abstrait qui n'est que la traduction d'une spécification informelle.

3.2 Architecture B de l'interface

L'architecture du vérifieur de structure est, à notre sens, peu conventionnelle notamment dans l'utilisation qui est faite de la clause `INITIALISATION`.

```
MACHINE
   Interface
VARIABLES
   V
INVARIANT
   P
INITIALISATION
   V :(P)
OPERATIONS
   Lire_V = …
```

```
IMPLEMENTATION
   Interface_i
REFINES
   Interface
IMPORTS
   Composant
INVARIANT
  vérifié = TRUE
INITIALISATION
  res ← verifs ;
  IF res = FALSE
  THEN
    Fatal_error
  END

PROMOTES
   Lire V
```

```
MACHINE
   Composant
VARIABLES
   V,
   vérifié
INVARIANT
   Typage &
   vérifié = TRUE
=> P
INITIALISATION
   V :(Typage) ||
   vérifié := FALSE
OPERATIONS
  res ← verifs =
    res,
    V,
    vérifié :(
      Invariant &
      res =
vérifié) ;
```

Fig. 6. Spécification de l'interface

L'interface du vérifieur de structure est une machine contenant des variables et des propriétés sur ces variables utiles à la vérification de type. Ces propriétés ne sont pas assurées initialement et ce sont principalement les tests internes de chaque composant qui les assurent. Une solution envisageable serait de garder chaque propriété par le fait que le composant qui l'assure a été vérifié structurellement. Le vérifieur de type ayant en précondition que tous les composants ont été vérifiés, il pourrait utiliser ces propriétés. Nous n'avons pas choisi cette solution qui a pour inconvénient de compliquer la preuve en remplaçant toutes les propriétés *P* par *(composant_x_verifié = TRUE => P)*.

La solution que nous avons mise en place utilise la clause INITIALISATION de la machine d'interface qui appelle les opérations de vérification interne de chaque composant. Ainsi nous nous assurons qu'à l'initialisation, en fait après la vérification syntaxique de tous les fichiers, le vérifieur de type mais aussi les tests externes peuvent s'appuyer sur ces propriétés. Nous avons utilisé une opération spéciale Fatal_Error qui a pour effet de terminer la vérification en renvoyant un code d'erreur. Ainsi le vérifieur de type n'est exécuté que si la vérification interne de chaque composant a réussi.

La figure 6 présente le squelette de l'architecture. Une interface propose une variable V avec une propriété P invariante. Dans l'implémentation, l'initialisation appelle une opération qui vérifie la propriété P et renvoie un booléen (cette opération spécifie que son résultat, les variables V et vérifié prennent des valeurs telles qu'elles vérifient l'invariant et que le résultat est égal à la valeur de vérifié). On sait que si ce booléen est vrai alors la propriété est vérifiée. L'initialisation teste alors ce booléen : dans le cas où il est faux, le programme s'arrête, sinon il continue. Ceci fait qu'à la sortie de l'initialisation nous pouvons considérer que la propriété est toujours vérifiée, cette propriété apparaît dans l'invariant de l'implémentation et permet d'appeler les opérations de lecture sur la variable sans propager à travers tout le modèle le fait que le composant soit vérifié.

3.3 Le modèle du vérifieur de structure

Cette section présente le modèle B du vérifieur de structure (figure 7). Cette partie du vérifieur 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. Nous nommons tests externes, les vérifications de propriétés entre composants.

La modélisation d'un composant 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, c'est ce modèle que nous présentons dans cette partie. La machine de plus haut niveau est décrite figure 6 (machine Composant). Elle contient un ensemble de variables abstraites modélisant les informations contenues dans le

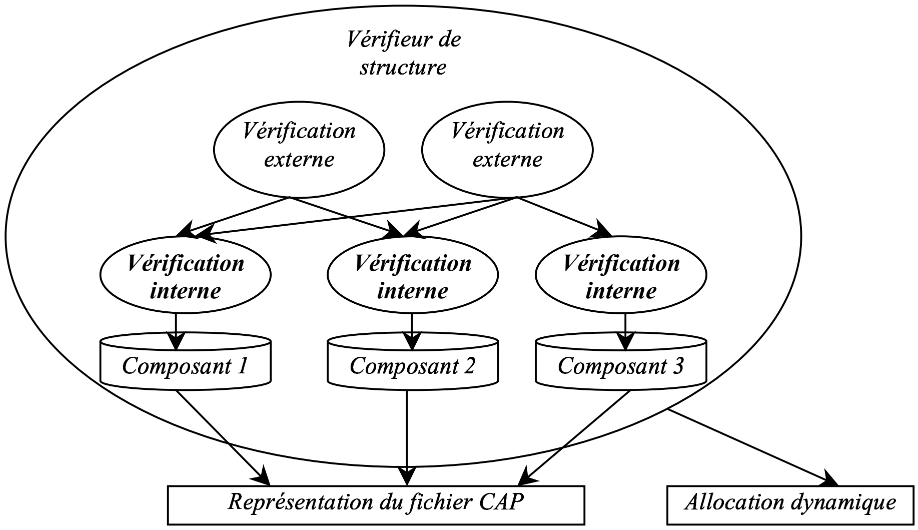

Fig. 7. Architecture du vérifieur de structure

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 a 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 précondition 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.

L'opération de vérification qui au niveau de la spécification construit les variables représentant le composant ne fait, au niveau de l'implémentation, que parcourir le flot d'octets en vérifiant sa cohérence mais ne construit en aucun cas une représentation interne du fichier CAP. De même au niveau de l'implémentation, les opérations de lecture accèdent directement aux données en mémoire. Ceci nécessite de raffiner les données en même temps que l'on définit l'algorithme de vérification. La figure 8 donne une idée du raffinement d'un composant : la variable abstraite `V` est raffinée, on définit une variable `Info` qui représente la zone mémoire associée au composant et on donne un invariant de collage qui définit l'emplacement de la donnée représentée par cette variable dans cette zone mémoire. L'opération de vérification vérifie la propriété `P` sur la zone mémoire et l'opération de lecture accède directement à la zone mémoire. Dans l'implémentation, nous utilisons un composant CAP (non présenté ici) qui

```
REFINEMENT
  Composant_r
REFINES
  Composant
CONSTANTS
  V_offset
PROPERTIES
  V_offset : word
VARIABLES
  Info
INVARIANT
  Info : word +->
word &
  (vérifié = TRUE
   =>
   V_offset :
dom(Info) &
   V = Info(V_offset)
&
   P(Info(V_offset)))
OPERATIONS
  res ← verifs =
    res,
    Vérifié,
```

```
IMPLEMENTATION
  Composant_i
REFINES
  Composant_r
IMPORTS
  Cap
VALUES
  V_offset = …
PROMOTES
  Lire_V
OPERATIONS
  res ← verifs =
  BEGIN
    l_size ←
lire_size;
    IF V_offset <=
l_size
    THEN
      v ←
lire(V_offset)
      IF P(v)
      THEN
        res = TRUE
      ELSE
        res = FALSE
      END
    ELSE
```

Fig. 8. Raffinement d'un composant

redéfinit la variable `Info` et des opérations de lecture. L'opération de vérification accède aux données via le composant CAP et réalise les tests. La variable `Info` est elle-même raffinée ultérieurement et définie comme une partie de la mémoire où est stockée l'applet. Cet exemple a pour but de donner une idée des raffinements de la vérification de structure. Il faut savoir que lors du développement, il a fallu pour certains composants passer par plusieurs étapes de raffinement (jusqu'à neuf) pour passer d'une représentation abstraite aux données réellement stockées en mémoire.

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.

De ce fait, l'apport de la méthode B est moins flagrant pour un tel développement puisque la preuve n'assure pas l'implémentation correcte d'un modèle restreint à un ensemble défini de machines, mais assure uniquement la correction des invariants de collage vis-à-vis des opérations. Deux erreurs cumulées dans le collage et l'implémentation ne seraient pas détectées. Par exemple, figure 8, une erreur dans la valuation de la constance V_offset ne serait détectée à la preuve. Pour cette raison, et pour des raisons de temps de développement, nous n'avons pas modélisé intégralement le vérifieur de structure, seuls neuf des douze composants ont été modélisés, les trois autres étant implémentés dans des machines de base.

La modélisation des tests externes Le second objectif du vérifieur de structure est de réaliser les tests externes. Ces tests permettent d'assurer la cohérence d'informations partagées entre différents composants. Ces vérifications s'appuient sur les tests internes des composants et utilisent les services fournis par chaque composant.

Tous les tests externes sont modélisés, raffinés et implémentés en B. Ici, il est très intéressant d'utiliser la méthode B car chaque test externe est modélisé de manière abstraite en utilisant la représentation abstraite des informations présentes dans les composants. Cette modélisation est ensuite implémentée, le plus souvent par des boucles, et la preuve assure que cette implémentation souvent complexe est correcte.

À chaque composant est ainsi associé une machine ne contenant que des opérations mettant en œuvre les tests externes permettant d'assurer la cohérence de ce composant vis-à-vis des autres.

4 Intégration dans la carte à puce

Cette section porte sur l'intégration du code formellement développé dans une carte à puce, de la génération du code à partir du modèle formel à son implantation dans la carte.

4.1 Génération de code

L'un des principaux avantages de la méthode B est la génération automatique de code. En fait, le processus de raffinement se termine sur une implémentation écrite en B0 (sous-ensemble du langage B). Un des principaux points d'interrogation, au début de ce projet, était l'efficacité et surtout la taille du code obtenu par traduction.

Nous avons développé un prototype de traducteur en partant des travaux de Bossu (Bossu, 1999). Ce prototype est un traducteur simpliste qui ne fait aucune optimisation de code, il prend juste en compte le type des variables pour les déclarer avec des types C de taille minimum ce que ne fait pas le traducteur actuel de l'Atelier B développé par ClearSy[7] Le code obtenu peut sembler très loin d'un code optimisé pour carte à puce tel qu'il est encore aujourd'hui développé par les implémenteurs carte, mais un peu à notre surprise et grâce aux optimisations du compilateur, nous obtenons finalement un logiciel embarquable avec des tailles de code raisonnables.

4.2 Implémentation sur la plate-forme ATMEL AT 90

La cible sur laquelle a été implémenté notre vérifieur est une plate-forme ATMEL. Nous avons une première version du logiciel qui tient sur 32 Ko. Cette première implémentation nous a permis de démontrer la faisabilité de l'intégration de code issu d'un modèle formel dans une carte à puce. Cette première version du vérifieur contient un vérifieur de type complet excepté les byte-codes gérant les entiers 32 bits et un vérifieur de structure avec uniquement les vérifications internes à chaque composant.

Une deuxième version du vérifieur, plus complète a été intégré sur un composant disposant de 64 Ko de ROM. Dans cette deuxième version, le vérifieur accepte les byte-codes gérant les entiers 32 bits et la phase de vérification structurelle est complète, incluant les tests internes de chacun des composants du fichier CAP et les tests inter-composants. Dans cette dernière version, la taille du vérifieur de byte-code atteint 45 Ko. Cette taille est importante pour une carte à puce, mais il est à noter qu'aucun travail de fond pour l'élimination de tests redondants ou d'optimisation du code n'ont été entrepris. Cela laisse entrevoir des possibilités de réduction de la taille du code.

Nous avons bénéficié pour réaliser cette implémentation du compilateur IAR qui propose un haut niveau d'optimisation du code. Ceci a pour effet négatif de dégrader les performances en terme de temps d'exécution mais notre principal souci était dans un premier temps de « faire rentrer le vérifieur dans une carte ». Dans la section 5, les résultats concernant les temps de réponse du vérifieur sur des applets réelles montre que ces temps, même s'ils peuvent être améliorés, ne sont pas irréalistes.

[7] Le site internet de ClearSy : http ://www.clearsy.com

4.3 Intégration des développements formels dans un cadre informel

Nous n'avons pas modélisé la carte dans son intégralité. Dans notre cas, seul le vérifieur de byte-code a été modélisé à l'aide de la méthode B. Une importante question est de savoir jusqu'à quel point nous pouvont avoir confiance en ce développement formel alors qu'il s'appuie sur des briques non formelles.

En réponse, nous pensons qu'un développement formel est utile pour traiter un logiciel complexe, cela permet d'améliorer la qualité du produit et la confiance que nous pouvons avoir en lui. Malgré cela, le domaine des cartes à puces reste un domaine de spécialistes où les contraintes, notamment en ce qui concerne la taille des logiciels embarqués, ne permettent pas de construire des systèmes aussi complexes que dans l'industrie classique. De ce fait, les techniques usuelles de test permettent le plus souvent d'assurer un très bon niveau de qualité. Dans la jeune histoire des cartes, l'utilisation des méthodes formelles n'était jusqu'à présent pas nécessaire pour obtenir des logiciels de très haute qualité.

Actuellement, l'ouverture des cartes sur l'extérieur amène une complexité nouvelle qui devient de plus en plus difficile à appréhender pour le développeur. L'utilisation de techniques mathématiques semble nécessaire pour garder le même niveau de qualité. Partant de cet état de fait, il est raisonnable qu'un vérifieur soit développé formellement en s'appuyant sur des briques de base réalisant la gestion mémoire ou le chargement de code. Le code issu du formel s'intègre facilement avec le code préexistant, l'interaction étant réalisée à l'aide d'interfaces abstraites, que nous nommons machine de base en B, pour ces briques de base comme décrit dans la figure 9.

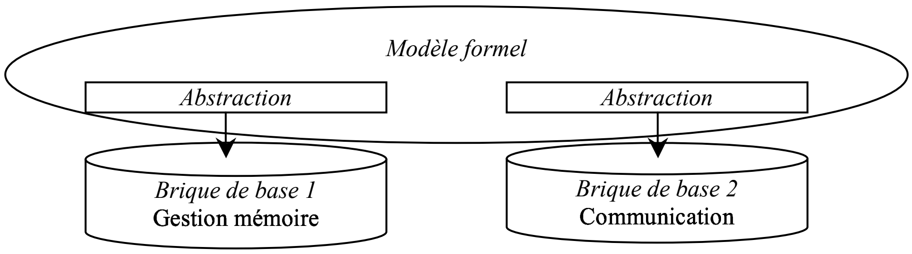

Fig. 9. Intégration des développements formels et informels

4.4 Les limitations des modèles formels

Les modèles formels du vérifieur de byte-code embarqué sont conçus pour produire une implémentation formelle. Dans notre méthodologie de développement, il nous a fallu prendre en compte non seulement le développement formel, 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 compilée 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 bonne marche du vérifieur. Il nous faut alors utiliser d'autres techniques pour nous assurer du bon fonctionnement du vérifieur. La figure 10 décrit les différentes phases du développement, de l'expression des besoins fonctionnels du vérifieur au code embarqué de ce même vérifieur.

En particulier, nous notons le recours à la relecture de spécification et aux tests. 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 recouvrir la phase de relecture, mais aussi 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 de 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 que le code produit en fin de développement correspond aux attentes formulées par les besoins informels.

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 test et la relecture de code.

5 Quelques métriques sur le vérifieur de byte-code

Dans cette section, des métriques concernant le développement formel du vérifieur de byte-code sont discutées. Cela concerne en particulier les données relatives au développement, mais aussi celles sur le code obtenu et les éventuels surcoûts occasionnés par l'approche choisie pour l'algorithme de vérification.

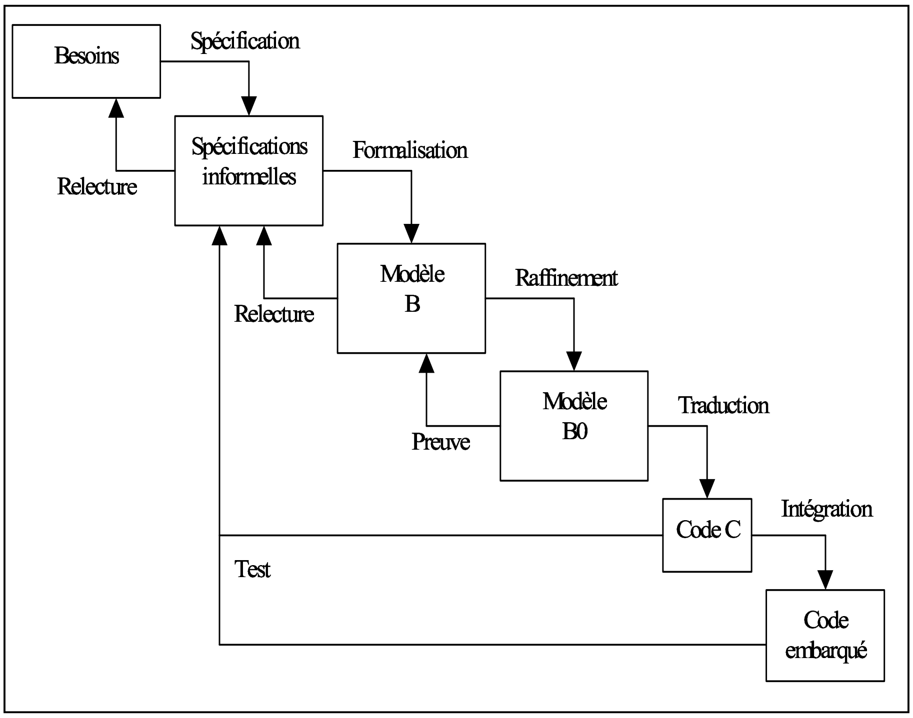

Fig. 10. La méthodologie de développement

5.1 Le développement formel

Le tableau 1 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 machinerie 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.

Le tableau 1 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

Tableau 1. Métriques sur le développement formel du vérifieur de byte-code

	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
Charge de travail (h/m)	8	4	4	16

principale raison a été évoquée à la section 3.1.1 : le traitement par cas effectué par le vérifieur de type produit 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.

5.2 Les métriques sur le code embarqué traduit du développement formel

Le tableau suivant (tableau 2) 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.

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

Tableau 2. Métriques sur la vérification d'applets par le vérifieur de byte-code

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

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.

5.3 Les métriques sur l'ajout du composant de preuve

L'ajout du composant de preuve n'est pas sans conséquence pour le vérifieur de byte-code. Le but de ce composant est de simplifier la partie vérification de type en rendant l'algorithme de vérification linéaire. Pour cela, les informations de typage nécessaire à ce nouvel algorithme sont stockées dans le composant de preuve.

Étant un nouveau composant ajouté au fichier CAP, il doit être également vérifié structurellement. Comme les autres composants du fichier CAP, le composant de preuve possède une structure où sont mémorisées les informations. De même, il référence des éléments d'autres composants. Cette structure nécessite une vérification pour s'assurer que les informations référencées existent réellement. La phase de vérification structurelle est donc légèrement alourdie par l'ajout des tests relatifs à ce nouveau composant. Cependant ce surcoût n'est pas très important. En effet, la vérification structurelle est une étape rapide du processus de vérification (tableau 2), en particulier quand nous la rapportons à la vérification de type.

Le deuxième surcoût est lié à l'occupation mémoire. Il s'agit du surcoût le plus crucial, la mémoire étant une denrée rare sur la carte à puce. Le tableau 3 indique pour quelques applets le surcoût d'occupation de la mémoire EEPROM en fonction de la taille de l'applet.

Théoriquement le surcoût relatif peut être supérieur à 20%. Dans le cas pratique, comme le montrent les exemples du tableau 3, le surcoût est inférieur à 20%. Il dépend de la nature de l'application. Comparé à la technique de Leroy qui ne nécessite qu'un surcoût de l'ordre de 2% (Leroy, 2001), l'approche que nous avons choisi est plus consommatrice en mémoire EEPROM, mais elle optimise l'utilisation de la RAM. Le tableau 3 montre également la consommation en RAM requise pour effectuer la vérification. Nous notons en particulier les

Tableau 3. Le surcoût en mémoire du composant de preuve

Applet	Taille (octets)	Taille du composant de preuve (octets)	% relatif	Consommation en RAM (octets)
Utils	4439	868	19,5	28
pacapint	2375	23	0,9	26
Wallet	2762	188	6,8	29
TicTacToe	4988	336	6,7	36
Applet 1	10394	1303	12,5	61
Applet 2	22554	3791	16,8	90

faibles quantités de RAM nécessaire à la vérification qui restent en dessous de la centaine d'octets pour les exemples proposés.

Les éléments de preuve placés en EEPROM ne sont jamais modifiés : ils sont écrits une fois pour toute. Nous n'avons donc pas de coût lié à la modification de ces valeurs. Une fois la vérification terminée, les informations de typage peuvent être supprimées de la mémoire de la carte. Le surcoût est donc temporaire, durant la phase de vérification où il est nécessaire de disposer de ces informations.

6 Conclusion

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. De plus, une spécification claire et complète de ce que doit faire un vérifieur est disponible. Un point positif est que tous les tests ont été spécifiés sans optimisation. Notre prototype peut alors être utilisé comme une implémentation de référence d'un vérifieur embarqué. Ce sont les bénéfices majeurs que nous retirons de l'utilisation des méthodes formelles. Un autre résultat important est l'expérience acquise pour intégrer du développement formel dans un développement traditionnel déjà existant, ainsi que le coût d'un développement formel et le coût de l'intégration. L'utilisation des méthodes formelles n'est pas gratuite. 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. La raison est que les erreurs découvertes par les développements formels et traditionnels sont similaires. Ainsi, il ne faut pas perdre son temps sur de tels développement mais concentrer ses efforts sur d'autres points comme le vérifieur de type où les apports des méthodes formelles sont plus importants. Cependant, nous montrons aussi qu'une description formelle de chaque composant est à la fois nécessaire et utile, d'abord pour le reste du développement formel qui s'appuie sur cette description, mais aussi pour le développement traditionnel. En effet, la spécification du composant fournie par la description formelle, permet d'éviter de nombreuses erreurs dues aux mauvaises interprétations de la spécification.

Bibliographie

Abrial J.R., *The B Book, Assigning Programs to Meanings*, Cambridge University Press, 1996.

Bertot Y., A Coq formalization of a Type Checker for Object Initialization in the Java Virtual Machine, Research Report, INRIA Sophia Antipolis, 2000.

Bossu G., Traducteur de spécifications B vers le langage C, Rapport de stage, Laboratoire d'informatique de Franche-Comté, 1999.

Casset L., Lanet J.-L., 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.

Casset L., Formal Implementation of a Verification Algorithm Using the B Method, *Proceedings of AFADL 2001*, Nancy, June 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.

Freund S., The Costs and Benefits of Java Bytecode Subroutines, *Formal Underpinnings of Java Workshop at OOPSLA*, October, 1998.

Klein G., Nipkow T., Verified Lightweight Bytecode Verification*, in ECOOP 2000 Workshop on Formal Techniques for Java Programs*, pp. 35-42, Cannes, June 2000.

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

Leroy X., On-Card Byte Code Verification for Java Card, *Proceedings of e-Smart 2001*, Cannes, September 2001.

Lindholm T., Yellin F., *The Java Virtual Machine Specification*, Addison Wesley, 1996.

Necula G., Lee P., Proof-Carrying Code, *in 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 106-119, Paris, 1997. http ://www-nt.cs.berkeley.edu/home/necula/public_html/popl97.ps.gz

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

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

Pusch C., Nipkow T., von Oheimb D., μJava : Embedding a Programming Language in a Theorem Prover, *In Foundations of Secure Computation*, IOS Press, 2000.

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.

Requet A., Casset L., Grimaud G., Application of the B Formal Method to the Proof of a Type Verification Algorithm, *HASE 2000*, Albuquerque, November 2000.

Rose E., Rose K. H., Lightweight Bytecode Verification, in *Formal Underpinnings of Java, OOPSLA'98 Workshop*, Vancouver, October 1998. http ://www-dse.doc.ic.ac.uk/~sue/oopsla/rose.f.ps

Sun Microsystem, *Java Card 2.1.1 Virtual Machine Specification*, 2000.

Sun Microsystem, *Connected, Limited Device Configuration*, Specification 1.0a, Java 2 Platform Micro Edition, 2000.

Lilian Burdy est ingénieur de recherche au sein du laboratoire de recherche logiciel de la société Gemplus. Ses travaux portent, d'une manière générale, sur la mise en oeuvre des méthodes formelles dans le contexte industriel, notamment sur leur application au niveau des langages objets.

Ludovic Casset est consultant de la société ClearSy. Ses travaux portent sur l'industrialisation et l'application des méthodes et des techniques formelles. Ses précédentes expériences ont porté sur les travaux de modélisation de cartes à puce au sein du laboratoire de recherche de la société Gemplus avant de rejoindre la société ClearSy, spécialisée dans l'industrialisation des méthodologies formelles comme la méthode B.

Antoine Requet est ingénieur de recherche au sein du laboratoire de recherche logiciel de la société Gemplus. Ses travaux portent sur l'application des méthodes formelles aux logiciels pour carte à puce.